High-Performance Computing in GeoScience

Data Preprocessing by Domain Decomposition and Load Balancing

Dissertation

zur Erlangung des Grades eines Doktors der Naturwissenschaften

der Geowissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen

vorgelegt von
Dany Kemmler

2009

Tag der mündlichen Prüfung: 16.03.2009

Dekan: Prof. Dr. Peter Grathwohl

1. Berichterstatter: Prof. Dr. Paul D. Bons, Tübingen

2. Berichterstatter: Senior Lecturer Dr. Sven Helmer, London

In memory of my beloved grandma, who passed away too soon. Thank you to Matthias, for his love, care and never-ending support, and also to my family and all the countless good souls everywhere who believed in me, encouraged me and helped me, so that this thesis could become reality one day ...

Foreword and Acknowledgements

When my supervisors and the administrative people at the School of Geosciences here in Tuebingen are asked what part of my thesis would be the most important one, they definitely will reply: "Everything from page 1 on." In my opinion, this is just half the truth! They might be right in regard to the pure "facts" and the "hard work" that is always hidden behind such a big and long Ph.D. project.

But, to be honest, in my eyes this short chapter here is really the most important part. Such a thesis could never succeed without the help of countless people in the background who give support, offer their help, have an open ear for all kinds of problems and who do not hesitate to give encouragement, especially at those times when everything looks quite hopeless and the seemingly endless amount of work appears to exceed the capacity of one human being.

It is impossible to name them all; each is a single jigsaw piece in the whole project called "Ph.D. thesis." It doesn't matter in what way the support was offered. Everybody is equally important. Therefore, thank you to you all!

Nevertheless, I explicitly would like to thank some of these helpful souls here in personal by name. The order is not important; I have to start somewhere ...

First of all, there is my "better half," *Matthias*, who never stopped believing in me, even at times when I was completely convinced that working towards a Ph.D. was the most stupid idea I had ever had in my whole life. His loving support and continuous encouragement through this seemingly endless time of writing my dissertation is unbelievable. Thank you, honey! Without you this work would never have been finished. Thank you for believing in me and being there for me all along!

As you can see, this thesis is written in English, although I'm German. I have no English-speaking roots nor am I a linguistic talent. But when you peruse this text it reads smoothly and looks perfectly fine. Well, this is the hard work of *Ginny Dittrich*! I can't thank her enough for her endless patience in correcting my "colloquial wish-wash" and somehow converting the whole into the nice and readable English thesis that you now hold in your hand. She was my "Language Dschinn" and I can count myself more than lucky that she agreed to help me with my English!

Although it is "their job," I definitely would like to thank both my Ph.D. supervisors, *Paul Bons* in Tuebingen and *Sven Helmer* in London. There was a time when it looked as if I could finally "bury" the prospect of ever finishing my Ph.D. thesis. With their help and support it was possible to finish this work, despite administrative problems and unlucky circumstances which cropped up. They have been very helpful, have given good advice, have been very fair and had very good ideas. I experienced a great deal of support from both of them. They were not required to help me, but offered me a hand and made my dream of a Ph.D. come true in the end.

Another person whom I can't thank enough is *Wolfgang Frisch*. He always had an open ear. Whenever I thought there was no way out of mountain ranges of problems of all kinds, he always

had a solution or an idea and a word of comfort. Without his endless support and encouragement I would never have been able to finish this thesis. And for me a little dream came true when it turned out that I would be the very last Ph.D. candidate he would be testing before finally retiring from a long academic life. I hope I didn't disappoint him in the end ...

One person unfortunately is no longer able to celebrate the termination of my thesis with me. This is my beloved grandma who passed away way too soon. She was the very first one to encourage me to step forward and start such a "big project" and to work towards my Ph.D. She was always on my side even when countless others thought I was completely out of my mind. She was so proud and was looking forward to the day when I finally would step up to her door and ring her bell with my Ph.D. certificate in my hand. Wherever she is now, this thesis is dedicated to her and I hope she is proud and very happy. Thank you grandma; I miss you!

In addition to the people mentioned above, there are also *Steffen* and *Hadmut* who helped me whenever the technical equipment refused to work as I wanted it to. They have been my "computer emergency squad" and I always could count on them. Then there is of course my family, who first seemed a little bit "surprised" about my "stupid" decision to go for my Ph.D., but never gave up hope and believed in me. And last but not least I also am grateful to the fantastic group of people around Paul Bons and Wolfgang Frisch who have always been very nice to me, have been supportive whenever I needed help or just someone to talk to, and who accepted me as if I had always been a normal and full group member. They provided a very pleasant and friendly working environment and I enjoyed being there.

Most likely I have forgotten a million other helpful souls. In case someone is missing his or her name here, I'm very sorry, but to all of you a big thank you and my gratefulness for everything you all did for me so that I finally could hand in this piece of work that represents my Ph.D. thesis.

Höchstleistungsrechnen in den Geowissenschaften – Datenvorverarbeitung mittels Gebietszerlegung und Lastbalancierung

Computer sind heute ein fester Bestandteil in unserem täglichen Leben und haben zu einem extremen Fortschritt in den letzten Jahrzehnten beigetragen. In Forschung und Wissenschaft hat der Einsatz von Computersimulationen die Beantwortung vieler wichtiger und spannender Fragen erst ermöglicht. Ohne solche Simulationen können bestimmte, komplexe Sachverhalte gar nicht angegangen werden, weil z.B. der physische Zugang fehlt oder die betrachteten Zeiträume zu lang sind. Die Untersuchung von Prozessen in vielen 100 km Tiefe in unserer Erde, das Verhalten von radioaktivem Müll, der in einem in Bentonit umschlossenen Metallcontainer eingelagert wird, und den dann stattfindenden Prozessen über viele 1000 Jahre oder präzise Wettervorhersagen auf dem Niveau, wie wir es heute gewohnt sind, gehören zu diesen Sachverhalten.

Aber auch trotz des enormen Fortschritts im Sektor der Computertechnik ist es heute immer noch nicht möglich mit Standardcomputern sehr rechenintensive Probleme zu lösen. Hier ist der Einsatz von Spezialhardware von Nöten, die mehrere 10 Mio. Euro kostet. Erst durch die Nutzung von diesen Höchstleistungsrechnern sind z.B. die heutigen Wettervorhersagen überhaupt erst möglich. Aber auch wichtige Fragestellungen in vielen anderen Gebieten der Wissenschaft können nur mit Hilfe solcher “Number Cruncher” angegangen werden.

Für Computersimulationen erstellt man allgemein immer erst ein Modell oder Abbild der Realität, mit dem dann die Antworten auf bestimmte Fragen berechnet werden sollen.

Bei der Entwicklung von Software, die innerhalb solcher Modelle die vorhandenen Daten verarbeitet und Berechnungen vornimmt, ist es wie mit der Entwicklung eines Formel-1 Wagens: Letztlich handelt es sich, aus der Sicht eines Laien betrachtet, immer noch um ein Auto; tatsächlich ist aber jeder einzelne Formel-1 Wagen eine genau angepasste Einzelanfertigung, an der viele, verschiedene Ingenieure mitentwickelt haben. Genauso verhält es sich mit Software, die auf Höchstleistungsrechnern einmal laufen soll. Für den Laien ist es nichts anderes als ein bisschen Software in C oder Fortran, doch tatsächlich handelt es sich bei diesen Programmen um exakt abgestimmte Spezialsoftware, die genau auf die jeweils unterliegende Hardware angepasst sein muss. Viele einzelne Punkte im Parallelisierungsprozess sind wichtig. Aber nur die Gesamtheit aller notwendigen Einzelschritte ergibt letzten Endes ein lauffähiges, genau auf die Maschine ausgerichtetes Höchstleistungscomputerprogramm.

Typischerweise zeichnen sich die Modelle, die mit den Höchstleistungsrechnern berechnet und simuliert werden sollen dadurch aus, dass sie eine enorme Menge an Eingabedaten bewältigen müssen. Eine gezielte Vorverarbeitung dieser Datensets mit oft mehreren 10er Millionen Eingabedaten, ist ein unabdingbarer Schritt, um im späteren Verarbeitungsprozess, die eigentliche Problem- oder Modellparallelisierung erst angehen und durchführen zu können. Nur wenn die Eingabedaten optimal auf- und vorbereitet sind, kann die unterliegende Spezialhardware sinnvoll für die spätere Modellberechnung genutzt werden.

Trotz der hochwertigen Ausstattung der Höchstleistungsrechner sind auch physikalische Grenzen gesetzt was z.B. die Speichergröße oder die Taktfrequenzen betrifft. Je nach Typ der zu Grunde liegenden Hardware sind verschiedene Beschränkungen vorhanden. Kein Großrechner gleicht notwendigerweise dem anderen, da es keine klassischen Rechner von der Stange sind, wie man sie auf jedem Schreibtisch findet.

In der Regel zeichnen sich Höchstleistungsrechner dadurch aus, dass sie sehr viele extrem schnelle Verarbeitungseinheiten, also Prozessoren besitzen, die untereinander in einer gewissen Weise vernetzt und vor allem auf enorme Rechengeschwindigkeiten bei der Verarbeitung von Gleichungssystemen ausgelegt sind. Die höchsten Verarbeitungsgeschwindigkeiten werden dann erzielt, wenn jeder einzelne Prozessor möglichst ungestört nonstop vor sich hinrechnen kann. Jede Unterbrechung, egal aus welchem Grund, treibt die eigentliche Gesamtverarbeitungszeit nach oben. Daher ist es ein erklärtes Ziel bei dem Einsatz solcher Multiprozessorsysteme, die Unterbrechungszeiten und -mengen so gering wie möglich zu halten.

Auf der anderen Seite werden die Modelle, die berechnet werden sollen, normalerweise durch enorm komplexe partielle Differentialgleichungssysteme dargestellt, die auf den vorhandenen Eingabedaten operieren. Wenn nun mittels eines Höchstleistungsrechners solch ein Problem/Modell gelöst werden soll, müssen zum optimalen Einsatz eines solchen Multiprozessorsystems, die vielen Millionen Eingabedaten sinnvoll auf die einzelnen Verarbeitungseinheiten verteilt werden, so dass die unterliegenden mathematischen Gleichungen und Rechenschritte immer noch korrekt und sinnvoll sind. Man kann also nicht einfach hingehen und die vorhandenen Datensets beliebig auseinanderreißen um sie gezielt zu verteilen. Hier sind ausgefeilte Algorithmen von Nöten, die die Eingabedaten ganz bewusst so in Untersets zerlegen, dass die Gesamtmodellberechnung immer noch korrekt bleibt.

Dabei muss darauf geachtet werden, dass die Zerlegung des Ausgangsproblems derart passiert, dass die Daten optimal auf die einzelnen Verarbeitungseinheiten des Grossrechners verteilt werden. Die Datenverteilung darf nicht dazu führen, dass einige Prozessoren sehr viel arbeiten müssen, während andere die meiste Zeit untätig sind. Nur bei einer vernünftigen sogenannten Lastbalancierung, ist eine optimale und schnellst mögliche Berechnung mittels der Spezialsoftware und den vorhandenen Eingabedaten möglich. Die Lastbalancierung sorgt dafür, dass die Rechenlast gleichmäßig auf die einzelnen Prozessoren verteilt wird.

Bei der Vorverarbeitung dieser großen Eingabedatenmengen müssen viele temporäre Zwischenergebnisse sowie Hilfsvariablen im Speicher gehalten werden. Nicht nur die eigentliche Menge an Eingabedaten muss vernünftig in schnellen und optimal angepassten Datenstrukturen gespeichert werden, sondern auch diese zusätzlichen, weiteren Datenmengen. Obwohl bei den heutigen Großrechnern zwar gewöhnlich bereits schon schnellere und größere Speicher zum Einsatz kommen, ist trotzdem auch deren tatsächliche physikalische Grenze irgendwann einmal erreicht. Ohne den Einsatz einer Datenstruktur, die nicht nur auf die Datenmengen und deren Speicherung für die spätere Weiterverarbeitung angepasst sein muss, sondern gleichzeitig auch auf die optimale Nutzung und Anpassung der unterliegenden Hardware des Großrechners abgestimmt ist, wird direkt automatisch kostbare Rechenleistung durch z.B. unnötige, zusätzliche Verarbeitungsschritte bei der Modellberechnung verschenkt. Die Laufzeiten gehen unnötigerweise in die Höhe und im schlimmsten Fall ist sogar eine Berechnung des Modells durch die Eingabedatenflut erst gar nicht möglich.

Bei einer sinnvollen Vorverarbeitung ist es daher unabdingbar, all diese wichtigen Aspekte optimal miteinander zu vereinigen, auch wenn bestimmte Punkte beinahe widersprüchlich zu einander sind. Nur ein perfekt abgestimmtes Zusammenspiel zwischen Datenstruktur, Lastbalancierung und Gebietszerlegung ermöglicht letzten Endes eine gute Grundlage für eine spätere schnelle und effiziente Parallelisierung des Gesamtproblems.

In der vorliegenden Arbeit ist genau dies das Ziel: bestehende große Eingabedatensets sollen nicht nur perfekt so in Unterprobleme zerlegt werden, dass die mathematischen Gegebenheiten stets ihre Korrektheit behalten, sondern es soll gleichzeitig auch der Kommunikationsoverhead zwischen

den vorhandenen späteren Prozessoren zum Zusammenfügen der Einzel- und Zwischenergebnisse minimiert werden, um dadurch die Gesamtrechenzeit zu verkürzen. Ausserdem ist es wichtig auf dem eingesetzten Höchstleistungsrechners durch eine schnelle und platzsparende Datenstruktur den Speicher so optimal wie möglich zu nutzen. Durch diese genau auf das individuelle Problem abgestimmte Vorverarbeitung der gegebenen Daten ist eine optimale parallele Weiterverarbeitung der Eingabedaten zu einem späteren Zeitpunkt erst möglich.

Um die einzelnen Aspekte zu verdeutlichen, die notwenig sind dieses Ziel zu erreichen, wird der Leser schrittweise an die vorhandenen Teilprobleme heran geführt.

Zu Beginn der Arbeit wird eine kurze, sehr allgemeine Einführung in das Gebiet des Höchstleistungsrechnens gegeben, die die Notwendigkeit für eine gewisse Sorgfalt aber auch das Problembewußtsein erst einmal aufzeigt. Mittels eines einfachen und prägnanten Beispiels zur Berechnung einer Wettervorhersage, sowie von zwei historischen Ausgräbern, werden auf einfache und allgemeine Weise die Probleme im Sektor des Höchstleistungsrechnens dargestellt.

Das auf dieses als "Appetitanreger" gedachte erste Kapitel, dann folgende, gibt einen kleinen, sehr allgemeinen Überblick auf die Arbeit in Englisch sowie den Inhalt der diversen Kapitel im speziellen. Die Arbeit ist grob in 2 Teilbereiche gegliedert: einen sehr allgemeinen, aber einführenden Teil, in dem für Personen, die nicht unbedingt aus dem Gebiet der Informatik kommen, erst einmal die notwendigen Grundlagen gelegt werden, um den eigentlich wichtigen aber späteren Teil der Arbeit zu verstehen. Im zweiten Teil der Arbeit wird auf die oben bereits diskutierte Problematik der Gebietszerlegung, der Lastbalancierung, sowie der Auswahl der geeigneten Datenstruktur genauer eingegangen und eine ausführliche Diskussion zur später implementierten Lösung gegeben.

Der Einführungsteil beschäftigt sich zu Beginn mit dem Problem der Diskretisierung anhand einer Finiten Elemente Modellierung. Anschaulich wird schrittweise das Vorgehen bei einer solchen Diskretisierung sowie die evtl. auftretenden Probleme eingegangen.

Im darauf folgenden Kapitel 3 werden die verschiedenartigen aber dennoch wichtigen Aspekte im Bezug auf Computer allgemein aber auch im Hinblick auf die spätere Parallelisierung des Gesamtproblems genauer beleuchtet. Anhand von sehr realitätsnahen Beispielen werden auch dem fachfremden Leser immer wieder teils sehr abstrakte oder auch komplizierte Sachverhalte verständlich dargestellt.

Kapitel 4 beschäftigt sich schon etwas spezieller mit der Problematik, wie ein ursprünglich sequentiell ausgelegtes Computerprogramm gezielt auf eine spätere parallele Portierung vorbereitet werden kann und welche Probleme dabei entstehen können. Das zu Grunde liegende Modellierungsprogramm, das in der vorliegenden Arbeit zur Verwendung kam, nennt sich GeoSys/RockFlow und basiert auf dem Einsatz von Finiten Elementen.

Ab dem Kapitel 5 hat der Leser nun genügend Rüstzeug an der Hand um in die tatächlichen Sachverhalte der Arbeit einzusteigen. In diesem Kapitel werden gezielt die Probleme bei einer notwendigen Gebietszerlegung auf den bestehenden Eingabedaten diskutiert, notwendige theoretische Grundlagen bestehender Algorithmen und Grundwissen allgemein auf diesem Gebiet vorgestellt, sowie zwei weit verbreitete Werkzeuge, namens *Jostle* und *Metis*, die in dieser Arbeit benutzt wurden, eingeführt und genauer diskutiert.

Kapitel 6 ist das eigentliche Kernstück der Arbeit und behandelt nicht nur ausführlich die Implementierung verschiedener in Frage kommender Datenstrukturen, in denen die enormen Eingabedatensets später gespeichert werden sollen, sondern es werden auch wichtige Grundlagenalgorith-

men sowie eine ausführliche Diskussion der notwendigen Verarbeitungsschritte in der Vorverarbeitung vorgestellt. Wichtige weitere Werkzeuge werden ebenfalls besprochen sowie das implementierte Kommandozeilen-Inferface der fertig implementierten Vorverarbeitungssoftware.

Die Arbeit wird abgeschlossen mit einer gezielte Durchleuchtung der vorhandenen Ergebnisse im Hinblick auf die Leistungsfähigkeit der möglichen Datenstrukturen und eine Begründung für die letztliche Entscheidung zur aktuell implementierten, finalen Datenstruktur in der Vorverarbeitungssoftware. Ausserdem werden die beiden integrierten Gebietszerlegungsroutinen und ihre erzielten Ergebnisse ausführlich miteinander verglichen. Das Kapitel wird beendet mit einem Ausblick auf verschiedenste, mögliche Erweiterungen und Verbesserungen zur momentan implementierten Version der Vorverarbeitung – auch im Hinblick auf eine bessere Handhabung durch Nutzer, die nicht notwendigerweise die Arbeit auf kommandozeilenbasierten Plattformen gewohnt sind.

Die Arbeit wird abgerundet durch einen sehr ausführlichen Anhang. Dort gibt es ein Abkürzungsverzeichnis aller im Text eingeführten und verwendeten Abkürzungen, so dass der Leser im Falle eines Problems dort sofort nachschlagen und Hilfe finden kann. Desweiteren werden beispielhaft die Eingabedatensets vorgestellt sowie die später durch das neu geschriebene Vorverarbeitungsprogramm erzeugten Ausgabedaten für die weitere parallele Verarbeitung auf dem Großrechner. Ein Listing aller zur Zeit implementierten Kommandozeilen-Parameter in der aktuellen sowie zukünftigen Version der Software, sowie der Aufruf des Vorverarbeitungsprogramms auf der Kommandozeile ist ebenfalls vorhanden.

Der Anhang endet mit einem viele Seiten umfassenden Literaturverzeichnis sowie einem recht ausführlichen Stichwortverzeichnis, wo der Leser im Falle eines Problems oder zum schnellen Auffinden bestimmter Sachverhalte nachschlagen kann.

Contents

Motivation: About Weather Forecasts and Excavations ... 1

1 Abstract and Structure of the Thesis **5**
1.1 Abstract . . . 5
1.2 Structure of this Thesis . . . 6

2 From a Real-World Problem to a Computable Model **9**
2.1 The Process of Discretization . . . 9
2.1.1 The Difference between Discretization, Model and Modeling . . . 11
2.1.2 The Basic Units: Vertices and Elements . . . 12
2.2 From a Discrete Model to its Software Representation . . . 20

3 Basic Terms and Concepts of Parallelization **25**
3.1 The Basic Units of a Computer . . . 25
3.2 The Different Levels of Parallelization . . . 26
3.2.1 Parallelism at Bit Level . . . 27
3.2.2 Parallelism at Instruction Level . . . 27
3.2.3 Parallelism at Block Level . . . 31
3.2.4 Parallelism at Task or Process Level . . . 32
3.2.5 Parallelism at Job or Program Level . . . 33
3.3 Different Types of Parallelism . . . 33
3.3.1 Spatial Parallelism . . . 33
3.3.2 Temporal Parallelism . . . 33
3.3.3 Data Parallelism . . . 34
3.3.4 Functional Parallelism . . . 34
3.4 Parallel Computer Architectures . . . 34
3.4.1 *Flynn's* Taxonomy . . . 34
3.5 Single-Processor Systems . . . 35
3.5.1 SISD Computer Architectures . . . 35
3.5.2 SIMD Computer Architectures . . . 36
3.6 Multiprocessor Systems (MPS) . . . 40
3.6.1 MISD Computer Architectures . . . 40
3.6.2 MIMD Computer Architectures . . . 41
3.6.3 Shared Memory (SM) Multiprocessor Systems (MPS) . . . 44
3.6.4 Distributed Memory (DM) Architectures . . . 46
3.6.5 Distributed Shared Memory (DSM) Architectures . . . 47
3.6.6 Hybrid Architectures . . . 48
3.6.7 Computer Cluster . . . 48

3.6.8 Grid Computing and Distributed Computing . . . 51
3.6.9 Example of a MIMD Architecture . . . 52
3.6.10 Summary of Processor Architectures . . . 55
3.7 Interconnection Networks . . . 55
3.8 Load Balancing and Domain Decomposition . . . 56
3.9 General Programming Techniques for Parallel Systems . . . 57
3.9.1 MPI . . . 57
3.9.2 OpenMP . . . 58
3.9.3 Hybrid Programming – Combining MPI and OpenMP . . . 59
3.10 General Parameters for Performance Analysis . . . 59
3.10.1 Runtime Approximation for Sequential Programs . . . 59
3.10.2 Runtime Approximation for Parallel Programs . . . 60
3.10.3 Speedup . . . 62
3.10.4 Efficiency . . . 63
3.10.5 Scalability . . . 64
3.10.6 Utilization . . . 65
3.10.7 Example of Performance Analysis . . . 65
3.11 Summary and Conclusion . . . 67

4 From Sequential to Parallel Modeling - With Emphasis on GeoSys/RockFlow 69
4.1 Scientific Modeling - A Short Introduction . . . 69
4.2 The Program Package *GeoSys/RockFlow* . . . 71
4.2.1 Why Choose GeoSys/RockFlow? . . . 71
4.2.2 A Closer Look at GS/RF . . . 72
4.3 The General Process of Sequential (Computer) Modeling . . . 74
4.4 Transition to a Parallel Modeling Program . . . 79
4.5 Summary . . . 86

5 Domain Decomposition and Load Balancing 89
5.1 Importance of Domain Decomposition and Load Balancing . . . 89
5.2 A ”Quick Glimpse” into Theory . . . 99
5.2.1 Simple Partitioning Methods . . . 101
5.2.2 Inertial Partitioning Method . . . 101
5.2.3 Spectral Partitioning Method . . . 102
5.2.4 The KL Partitioning Method . . . 102
5.2.5 Multilevel-KL Partitioning Method . . . 103
5.3 The Process of Data Preprocessing . . . 103
5.4 Jostle . . . 107
5.4.1 The Chaco Input File Format . . . 107
5.4.2 How Does Jostle work? . . . 108
5.4.3 Usage of Jostle . . . 108
5.5 Metis . . . 112
5.5.1 How Does Metis Work? . . . 113
5.5.2 Usage of Metis . . . 113
5.6 Comparison and Future Improvements . . . 116
5.7 Summary . . . 117

6 Implementation of the Domain Decomposition Wrapper Application 119
6.1 Overview . . . 119
6.2 How to Choose the Adjacent Neighbors . . . 120
6.3 The Problem of Choosing the Appropriate Data Structure . . . 125
6.4 Linked Lists . . . 128
6.4.1 The Basics of Linked Lists . . . 128
6.4.2 Types of Linked Lists . . . 129
6.4.3 Basic Operations on Linked Lists . . . 130
6.4.4 Implementation of Linked Lists . . . 132
6.4.5 Linked List in the Preprocessing of the Model Data . . . 135
6.5 Array Structures . . . 137
6.5.1 The Basics of Arrays . . . 137
6.5.2 Multidimensional Arrays . . . 141
6.6 Bit Arrays . . . 145
6.6.1 Introduction – Problem of Accessing Single Bits Directly . . . 145
6.6.2 Congruences - Introduction into Modular Arithmetic . . . 146
6.6.3 Bit Arrays under Implementational Aspects . . . 148
6.7 Dynamically Allocated Arrays . . . 158
6.7.1 Introduction to Dynamically Allocated Arrays . . . 158
6.7.2 The Necessity of Dynamically Allocated Arrays . . . 159
6.7.3 The Usage of Dynamically Allocated Arrays in the Wrapper . . . 161
6.8 Valgrind . . . 170
6.8.1 Memcheck – The Valgrind Memory Profiling Tool . . . 170
6.8.2 Using Valgrind's Memcheck in Practice . . . 171
6.8.3 Validation of the Wrapper using Valgrind . . . 176
6.9 Input Data Processing . . . 176
6.10 Preprocessing of Model Data . . . 178
6.11 Post Processing and Outputting of Model Data . . . 180
6.12 Statistic Functions . . . 182
6.12.1 Statistic Information Offered by Jostle . . . 182
6.12.2 Statistic Information Offered by Metis . . . 185
6.13 Testing and Debugging . . . 187
6.14 Timer Functions . . . 187
6.15 The Commandline Interface of the Wrapper Program . . . 189
6.15.1 The `--help` Commandline Option . . . 191
6.15.2 The `--input` Commandline Option . . . 192
6.15.3 The `--partitions` Commandline Option . . . 193
6.15.4 The `--jostle` Commandline Option . . . 193
6.15.5 The `--metis` Commandline Option . . . 194
6.15.6 The `--parmetis` Commandline Option . . . 194
6.15.7 The `--ddcfile` Commandline Option . . . 194
6.15.8 The `--rfifile` Commandline Option . . . 195
6.15.9 The `--output` Commandline Option . . . 195
6.15.10 The `--statistic` Commandline Option . . . 195
6.15.11 The `--test` Commandline Option . . . 196
6.15.12 The `--verbose` Commandline Option . . . 197

6.15.13 The --debug Commandline Option . . . 197
6.15.14 The --weight and --noweight Commandline Option . . . 198
6.15.15 General Usage of the Wrapper Program . . . 199
6.16 Independence of the Wrapper . . . 200

7 Discussion of the Results, Outlook and Future Improvements 201
7.1 *Metis* or *Jostle*? - A Detailed Comparison between the Two Tools . . . 201
7.2 Implemented Data Structures in the Wrapper - Comparison and Discussion . . . 209
7.2.1 General Aspects of Performance Analysis . . . 209
7.2.2 Introduction to the *Big O Notation* . . . 210
7.2.3 Advantages and Drawbacks of Linked Lists . . . 214
7.2.4 Advantages and Drawbacks of Arrays in General . . . 218
7.2.5 Comparison of Linked Lists versus Standard Arrays . . . 219
7.2.6 Advantages and Drawbacks of Bit Arrays . . . 221
7.2.7 Advantages and Drawbacks of Dynamically Allocated Arrays . . . 222
7.2.8 Comparison of Bit Arrays versus Dynamically Allocated Arrays . . . 223
7.2.9 Summary and Concluding Discussion . . . 224
7.3 The Work Conditions during the Project . . . 225
7.4 Changes and Improvements between Wrapper Version domdec_v03_2 to domdec_v03_3 . . . 227
7.4.1 Handling of Input Files . . . 227
7.4.2 Dynamical Adjustment and Adaption of Array Size during Reallocation . . . 228
7.5 Future Improvements and Expansions of the Wrapper . . . 228
7.5.1 Parmetis Expansion for Parallel Domain Decomposition . . . 228
7.5.2 Support of Weights for Vertices and Edges . . . 230
7.5.3 A Graphical User Interface for the Wrapper . . . 230
7.5.4 Web Interface for the Wrapper . . . 231
7.5.5 Comfortable Installation on Various Unix/Linux-Based Platforms . . . 233
7.5.6 Construction of Classical Unix/Linux Man Pages . . . 234
7.5.7 Automatic Coupling of the Wrapper . . . 234

Appendix A – Acronyms 236

Appendix B - Different Output Examples of the Wrapper Program 241
Appendix B.1 - Example Output Files . . . 241
Appendix B.2 - Examples of Statistic Output of the Wrapper . . . 247
Appendix B.3 - Examplary Mapping File between Elements and Domains . . . 251

Appendix C – Help Text of Domdec V3.2 and V3.3 253

Appendix D – Test Data Sets 257

Appendix E – Bibliography 262

Appendix F – Index 271

List of Figures

1 Classical scientific method with and without numerical simulation 2

2.1 Simple groundwater contamination scenario due to environmental pollution by NAPL 10
2.2 Simple one-dimensional grids consisting of nodes and one-dimensional elements (= lines) 12
2.3 Simple two-dimensional elements, partly axisymmetric, are built by connecting a set of nodes 13
2.4 Simple three-dimensional elements, each consisting of a different set of nodes and possibly showing axisymmetry like the cube 13
2.5 Discretization of a simple one-dimensional problem domain 14
2.6 Variations in level of the water table before and after pumping water out of a well . 14
2.7 Discretization of a pumping scenario with a coning effect for a simple 2D grid . . . 15
2.8 Variable placement of nodes in a 2D problem domain where the field variables are expected to change rapidly 15
2.9 Discretization of a simply-shaped three-dimensional problem domain 15
2.10 Usage of symmetry effects to minimize mesh size 16
2.11 Usage of symmetric effects to reduce model dimension during discretization 17
2.12 Incorrect (left) and correct (right) nodal placement at the interface of two different materials 17
2.13 Gaps between elements as well as overlapping edges of elements are not permitted 18
2.14 Detached vertices and incorrect element shapes are not permitted 18
2.15 3D model of the geology of the Jordan Valley [Chen] 21
2.16 The long way from a 3D model to a mathematical representation 22
2.17 The finite element representation must be mapped into a matrix-vector product . . 22
2.18 The main steps in the whole process of translating a 3D model into a computable format using a standard computer system 23

3.1 The *von Neumann* architecture from 1946 26
3.2 Command pipeline with 5 interlocking independent microinstructions 28
3.3 Assembly line with 4 assembly stages for automobiles 29
3.4 A stream of very long instruction words is assigned to the different functional units within the VLIW microprocessor 31
3.5 Flynn's taxonomy of computer architectures 35
3.6 The structure of a traditional sequential computer architecture 36
3.7 Single Instruction Multiple Data (SIMD) architecture with shared memory 37
3.8 Single Instruction Multiple Data (SIMD) architecture with distributed memory . . 37
3.9 Multiple Instruction Multiple Data (MIMD) architecture with shared memory . . . 41
3.10 Multiple Instruction Multiple Data (MIMD) architecture with distributed memory . 42

3.11 An example of a simple symmetrical multiprocessor architecture 43
3.12 An example of a simple asymmetrical multiprocessor architecture 44
3.13 Architecture of a generic shared memory multiprocessor system 45
3.14 Architecture of a generic distributed memory multiprocessor system 47
3.15 Hierarchical architecture of the NEC SX-8 vector supercomputer 49
3.16 Medium-sized NEC SX-8 model 49
3.17 Small cluster which is built from off-the-shelf components 50
3.18 System configuration of each of the processing nodes of the *Earth Simulator* 53
3.19 Partial view of the operating room of the *Earth Simulator* 54
3.20 Overview of the *Earth Simulator Center* in Yokohama, Japan 54
3.21 2,890 km of cables connecting the 5,120 processing nodes of the *Earth Simulator* . 55
3.22 Activity plot of a parallel program during execution on 8 processors 61

4.1 Simple sequential program flow of a software package using the finite element method 76
4.2 Transition from a FE model presentation into a matrix-vector-product in the main processing stages of a sequential modeling application 77
4.3 What steps are necessary to compute a finite element model on a supercomputer? . 80
4.4 The simplified steps from a real-world model to its computation on a parallel computer 81
4.5 From a mathematical representation of the original real-world problem to a matrix-vector product with elements of the subdomains of the overall problem 81
4.6 The main steps in the process translating a mathematical representation of a model into a form that can be processed on a parallel computer 82
4.7 Zooming into an originally sequential modeling software which main processing part is be parallelized 84
4.8 Simple parallelization using MPI in the main stage of the GS/RF modeling software 85
4.9 Matrix-vector products resulting from a subdivision of a FE presentation of a model are computed as independent MPI processes and their result vectors are merged into one final one using the MPI command `MPI_All_Reduce` 85
4.10 A matrix-vector product is subdivided into different smaller subsystems, each constituting one single MPI process. The final result is merged from the different solution vectors of the smaller matrix-vector products. 86
4.11 Simplified overview of how to break down a global matrix-vector-product into different independent subsystems for a simple MPI parallelization approach 87

5.1 Highly differing computation times of 4 processing nodes 90
5.2 Quite similar computation times of 4 processing nodes 91
5.3 Quite similar computation times of 4 processing nodes, but with an interruption of the computation/work process at node no. 1 92
5.4 Activity plot of a parallel program during execution on 8 processors 93
5.5 Accumulation of the computation, execution and idle times during an execution of a parallel program running on 8 processor nodes 93
5.6 A FE presentation of a model and one, exemplary subdivision of it into several smaller subdomains 95
5.7 A simple finite element model consisting of 20 nodes 96
5.8 Problem of how to assign certain nodes at the borders to one of the 4 subdomains of a simple exemplary model 96

5.9 The white boundary node b_k will be used by both subdomains in their computations when processing the data of the adjacent elements e_1, e_2, e_5 and e_6 97
5.10 The main steps in the process translating a mathematical representation of a model into a form that can be processed on a parallel computer 98
5.11 A mathematical representation of a simple model with 24 elements and 20 nodes . 104
5.12 A simple mathematical model representation on the left and its corresponding adjacency graph to the right 106
5.13 A simple adjacency graph consisting of 7 vertices representing 7 elements in its former mathematical representation 109
5.14 A simple adjacency graph consisting of 7 vertices with the corresponding values of the Jostle data structures 111

6.1 A mathematical representation of a simple model with 24 elements and 20 nodes with element no. 10 as the starting point for finding its adjacent elements 120
6.2 Elements no. 3, 11 and 12 are in the immediate vicinity of element no. 10 121
6.3 All immediate neighbors of element no. 10 connected over its leftmost vertex in the triangle 122
6.4 All immediate neighbors of element no. 10 connected over its upper and its rightmost vertex in the triangle 122
6.5 All immediate neighbors of element no. 10 which are connected over mutual vertices 123
6.6 Two simple model representations, each consisting of differently shaped elements . 124
6.7 A section of a simple linked list 128
6.8 A simple singly linked list consisting of 4 list elements 129
6.9 An empty singly linked list with just an “empty reference” 129
6.10 A simple doubly linked list consisting of 4 list elements 129
6.11 A singly circular linked list consisting of 4 list elements 130
6.12 A doubly circular linked list consisting of 4 list elements 130
6.13 A fourth list element is to be inserted into a singly linked list 131
6.14 A list member is taken out of a singly linked list which automatically decreases in size after the deletion 131
6.15 A virtual memory with a size of 2000 bytes, organized in 4-byte blocks on a computer with a 32-bit architecture 133
6.16 An example of how a short singly linked list could be stored in the virtual memory 135
6.17 Excerpt of a computer memory ranging from memory address 924 to 940 137
6.18 Excerpt of a computer memory in which 365 different daily temperature values are stored consecutively 139
6.19 An excerpt as an example of a memory where a simple 2D array consisting of 6 elements with 2 rows of 3 elements each is stored 143
6.20 An excerpt as an example of a memory where a simple 1D array consisting of 6 elements is stored 144
6.21 Position of bit number 32,435 within 1 megabyte of data 145
6.22 Face of a clock with 12 digits 146
6.23 Face of a clock with 24 digits 146
6.24 Difference in memory consumption using 4-byte, 1-byte and 1-bit data structures, respectively 152
6.25 Truth tables for the 3 Boolean operations AND, OR and NOT 152
6.26 Truth tables for the 3 Boolean operations AND, OR and NOT 153

6.27 Adding two binary numbers, X and Y . . . 153
6.28 A simple bit array starting at bit 0 . . . 154
6.29 A bit array organized in bytes and blocks of word size on a 32-bit architecture . . . 154
6.30 A binary representation of the value n showing the two necessary indices for accessing its bit in a bit array . . . 155
6.31 Performing a division of 32 without a remainder on a binary representation of a number n . . . 156
6.32 Determining the correct bit position in a bit array for a binary value of 133 . . . 157
6.33 A node-to-element assignment using standard arrays and dynamically allocated arrays at the same time . . . 162
6.34 Example data structure holding the node information per element of a simple input file consisting of 20 vertices and 24 elements . . . 178
6.35 Example of the data structure that holds for each of the elements the subdomain number it was assigned by one of the domain decomposition tools . . . 180
6.36 Example data structure that holds the assignment of the subdomain for each node in a model using DAAs . . . 181

7.1 Domain decomposition of a 1D model consisting of 11 nodes and 10 linear elements 204
7.2 Domain decomposition by *Jostle* of a 1D model consisting of 11 nodes and 10 linear elements . . . 204
7.3 Domain decomposition by *Metis* of a 1D model consisting of 11 nodes and 10 linear elements . . . 204
7.4 Domain decomposition into 4 subdomains by *Jostle* of a model with 20 nodes and 24 elements . . . 205
7.5 Domain decomposition into 4 subdomains by *Metis* of a model with 20 nodes and 24 elements . . . 205
7.6 Comparison between the domain decomposition results of *Jostle* and *Metis* for a real model that consists of different geometrically shaped elements . . . 207
7.7 Comparison between the domain decomposition results of *Jostle* and *Metis* on a real model that consists of 2 mio. elements . . . 208
7.8 The O notation gives an upper bound for a function to within a constant factor: $f(n) = O(g(n))$ . . . 211
7.9 The Ω notation gives a lower bound for a function to within a constant factor: $f(n) = \Omega(g(n))$ . . . 212
7.10 The Θ notation bounds a function to within constant factors: $f(n) = \Theta(g(n))$ . . . 213
7.11 An example of the structure of one of the input data files that have been used . . . 261

List of Tables

3.1 Taxonomy of Parallel Computing Architectures after *Flynn* 35

6.1 A matrix that is called the "magic square," as its rows, columns and also diagonals all have the same sum 142
6.2 Standard units of data sizes after *IEC* 150
6.3 Listing of all available options and switches of `domdec_v03_2` 191

7.1 Exemplary table with results obtained after subdividing differently sized input files using the domain decomposition tool *Jostle* 206

Motivation

What have weather forecasts, excavations and parallel computers in common?

Not even 150 years ago the thought of being able to carry out some hundreds of arithmetic operations within just one second seemed unimaginable. With the dawn of the computer era this was no longer a big problem. However, scientists and engineers now were thinking about carrying out a few thousands of operations each second. And after thousands it was then millions, billions and even trillions (assuming the English system of denominations with one billion = 10^9, one trillion = 10^{12} and one quadrillion = 10^{15}).

One simple question arises in this connection: Where does this "lust" of constantly growing hunger for computing power come from? Why is it that we never can have a computer or processing unit that finally is fast enough? Why are one billion operations per second today enough but tomorrow we already want one trillion operations per second?

It is the basic method of science to first observe, then theorize and finally test one's findings or theory thoroughly. In engineering we have the same process: Designing a prototype, generally on paper, then building and testing it, and the result might be a new product.

As a result of consistently improving computer techniques and faster processing units and hardware it is nowadays less expensive to carry out sophisticated computer simulations than to build a series of prototypes for numerous experiments. The traditional manner of designing and prototyping in engineering is now increasingly being replaced by computer simulation and computation. Additionally, with much more computing power, we can now suddenly simulate phenomena in a way that seemed impossible a couple of decades ago, like precise weather calculations or research regarding our universe. Fig. 1 depicts the old, traditional scientific method and the contemporary one, where numerical simulation plays an important role.

Numerical simulations are becoming an increasingly important tool for scientists, especially in fields where physical experiments cannot be used to test the theories because they are too expensive or too time-consuming, because they might be unethical, or because they are simply impossible to perform. Most likely none of us will ever travel to the center of the earth. Although feasible, no geologist would ever waste a second on the thought of drilling a deep borehole every 10 meters for setting up a suitable hydrogeological model or for defining the structure of an oil reservoir. It is simply too expensive.

One billion or even one trillion operations each second - that surely might sound excessive. The following interesting and descriptive example found in [Pacheco] not only illustrates quite arrestingly that one trillion operations is rather modest but also shows a very simple problem of mankind: the more knowledge we acquire, the more complex our questions will become.

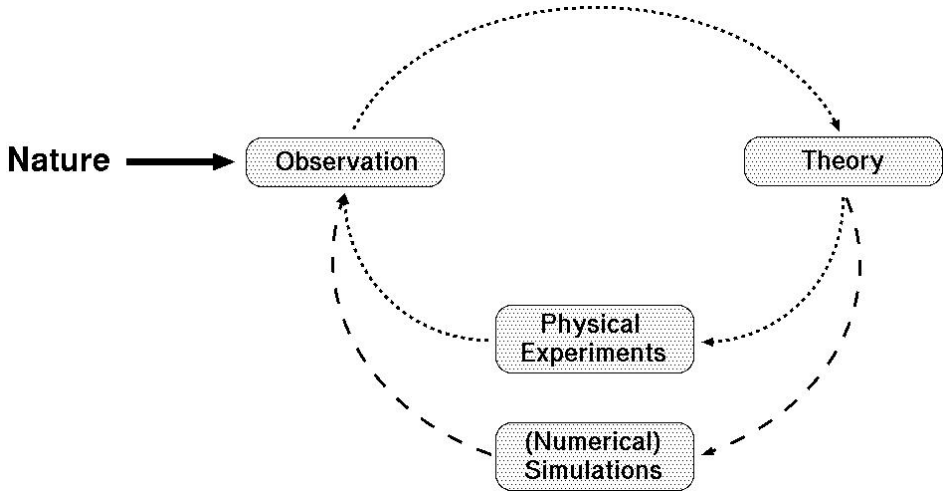


Figure 1: Classical scientific method with and without numerical simulation

Let us suppose we wish to predict the weather over the United States and Canada for just the next two days, and, further, that we want to model the atmosphere from sea level to an altitude of 20 kilometers. Our goal is to make a prediction of the weather only every hour for the next two days.

The standard approach to this type of problem is to cover the region of interest with a grid and then predict the weather at each vertex of the grid. Since the area of the United States and Canada is about 20 million square kilometers, and assuming we are using a cubical grid in which each cube measures 0.1 kilometers on each side, we will need at least

$$2.0 \times 10^7\, km^2 \times 20\, km \times 10^3\ cubes\ per\ km^3 = 4 \times 10^{11}\ grid\ points.$$

If it takes 100 calculations to determine the weather at a typical grid point, then in order to predict the weather one hour from now, we will need to make about 4×10^{13} calculations. Since we want to predict the weather every hour for 48 hours, our total of calculations will be

$$4 \times 10^{13}\ calculations \times 48 \approx 2 \times 10^{15}\ calculations.$$

If our computer can execute one billion (10^9) calculations per second, this will take about

$$2 \times 10^{15}\ calculations/10^9\ calculations\ per\ second =$$
$$2 \times 10^{16}\ seconds \approx 23\ days!$$

This simple example shows that the calculation is hopeless if we can only carry out one billion operations per second. If, on the other hand, our computer is able to perform one trillion (10^{12}) calculations per second, the result of our weather calculation will be ready for us within half an hour. In this case we'll actually be able to make a complete prediction of the weather for each of the next 48 hours.

It is quite simple to slightly modify this little example so that even one trillion operations per second will no longer be sufficient. For example, instead of a weather forecast for just the United

States and Canada we are now interested in a 48-hour weather prediction for the whole world. Now our area in question grows from 2×10^7 to about 5×10^8 square kilometers. The required computation time for this slightly different problem would increase from formerly half an hour to currently about 13 hours and additionally, our first 12 predictions would be useless.

But solving such problems regarding computational speed is not the only aspect to be considered. Developing greater computational power is most likely associated with vastly greater storage requirements. Even if we construct a computer capable of performing billions of trillions of operations each second, if it has access to only a few million words of memory, this computer is likely to be of little use to us.

Generally, such big computation-rich problems like weather prediction perform many routines where they repeatedly calculate rather simple equations; furthermore, these equations quite often do not necessarily all depend on each other. [Pacheco] again illustrates a nice simple analogous example to such a problem:

Suppose Peter is a Roman contractor who specializes in excavation. His single laborer, Paul, can excavate 1000 cubic feet a day. However, there is a huge surge in demand for his services when it is reported that Attila the Hun is going to pay a visit to Rome the following week. Peter figures that in order to meet the demand, he needs to excavate about 100,000 cubic feet a day for the next week. As Peter is clever, he solves his problem by simply hiring 99 more men, increasing his workforce from 1 to 100. (Incidentally, Attila didn't sack Rome after all! Tradition has it that Pope Leo I met with him and convinced him to leave Italy.)

If we now carry over this example to our computing problem, the analogy should be clear: Paul, the single laborer, is our processor and memory. The 100,000 cubic feet a day symbolize a great challenging problem or a mathematically and/or computationally demanding problem. Thus, our solution is clear: We should obtain more laborers, i.e., more processors and memory modules, to solve our problem. And this is what a parallel computer simply is: a "collection of computers" with multiple processors that can work together on solving a problem.

Unfortunately, matters are not that simple. The analogous situation for processors is much more complicated, as we haven't said *how* our processors will work together! Presumably, Peter might just have told each of his men explicitly where to dig and handed them a shovel. He wouldn't have been able to accomplish his job by simply hiring a bunch of unskilled laborers without giving them precise instructions! Or he could have considered additionally hiring a few skilled laborers who could supervise a smaller group of unskilled laborers and who would report to him every evening, thus taking the pressure off Paul of being "visible" everywhere all day long.

Coming back to our weather example, we won't be able to buy a "California-weather-predicting" computer or a "jetstream-predicting" processor anywhere on the market. Everything will depend on us, the programmers of the computers, and on the processors. We have to ensure that each processor or computational unit receives the appropriate instructions on how to perform its calculations properly. Furthermore, given the fact that processor *A* might receive clear instructions on how to predict the weather in Los Angeles, it still may need to communicate with processor *B* or *C*, which calculate the weather of nearby locations, to exchange important and necessary information.

The purpose of this little introduction is to make it clear that the existence of fast and powerful computers or parallel machines alone does not solve the current problems of our world. Rather, we, as programmers, scientists and engineers, have to take care to use the computers and hardware in such a way as to get the most benefit out of them. This results, for example, in finding a

suitable problem size by breaking a major problem into subproblems and distributing them in an appropriate way among several computing nodes, similar to Peter's hiring some supervisors who then take care of smaller workgroups of laborers. Our models should be adequate in size yet still precise enough for a good interpretation at the end of the model simulation and, at the same time, simplified enough to obtain a result without requiring endless computation times. (It is clear that Peter would always try to reduce the number of supervisors to a very small amount but still hire enough to have all unskilled laborers well directed and guided.)

This thesis deals neither with exhaustive weather forecasts nor Roman excavations. Rather, it focuses on a small part in the whole process of parallelizing big computation-rich problems. Especially in the geosciences there is a vast variety of computation-intensive problems like weather forecast, groundwater flow, geothermal reservoir usage and chemical impact of landfills on the countryside through waste water and nuclear repository modeling, just to name a few.

Although data preprocessing and problem- or model-subdivision might seem only of minor importance to the problem at first glimpse, one can easily see at second glimpse that porting model simulations in geoscience onto powerful parallel computers is not possible in the whole chain of necessary procedures without the seemingly trivial steps involved in preprocessing. The more carefully such tasks are accomplished, the better the outcome as a result of parallel computing. To show this fact is the goal of this thesis. The upcoming chapter gives a short overview of the content of the thesis and provides a listing of the individual chapters and their specific topics.

Chapter 1

Abstract and Structure of the Thesis

1.1 Abstract

The popularity and availability of computers is a simple fact of life in most of today's world. Computer simulation is a quick and relatively inexpensive alternative to physical experimentation in the scientific realm. Computer operations are meanwhile performed increasingly rapidly; one trillion operations per second are not anything extraordinary these days and are necessary in the field of automotive engineering, weather forecast or applied geology, for example. An interdependency of speed, power and storage space means that if one of these attributes is insufficient or impaired, it will limit the efficacy of the others. Data preprocessing and problem subdivision play a vital role in making real-world problems "computable."

High-performance computing is not only a cornerstone of modern life, but also a compelling topic in itself. The task of descretizing real-world problems and processing models so that these problems can be solved by computers involves obtaining finite amounts of data from real-world problem domains and replacing them by grids consisting of inter-connected nodes and elements which will serve to model the problem in question on a computer. Altering the size of the grid, its number of nodes, etc. to determine the optimal structure to simulate a certain problem on a computer is the ultimate goal here. The more carefully discretization is accomplished, the more successful the outcome of the model processing will be.

Real-world problems and the computation of their models can pose major challenges to software programmers. Some models are complex enough to require special hardware and call for multiprocessors and supercomputers to compute models. Such complicated models demand specialized algorithms and methods. Parallelization, which takes on many forms and acts at various levels, is a key concept here; part of the aim of parallelization is to allow calculational tasks to be performed simultaneously. Operating at the microinstructional level within the microprocessor, it aims at minimizing the idle time imposed on the processor by the external hardware.

The goal of minimizing runtime as much as possible must be kept in mind when selecting the architecture for particular problems. On the other hand, communication overhead must also be kept to a minimum to take maximum advantage of the computing environment chosen.

As the models of real-world problems are simplified mathematical representations of given real-world scenarios, sophisticated tools have to be employed to subdivide the problem domains into manageable "chunks" that can be distributed on specialized parallel computers with a certain number of processing nodes. There is always a trade off between communication overhead, runtime

or memory consumption. Depending on the given situation, the final solutions for the ultimate realization might vary greatly.

1.2 Structure of this Thesis

This thesis is an interdisciplinary work and touches on various fields in computer science but also on aspects in the earth sciences. As the main focus of the thesis is more computer-science related where often mathematical, technical or theoretical aspects prevail, it is also necessary to introduce certain terms and the basics to those readers who are not equipped with such knowledge, before "diving" into the essentials. On the other hand, often useful examples in this thesis mainly come from different fields in the geosciences where the rather technically oriented audience might not feel completely "at home." By starting out with more basic but general chapters and slowly moving towards the more specialized topics, readers either lacking a profound background in computer science or those who have just a little knowledge in geologically related matters have a chance to follow the text without being left behind. Although this approach will add to the number of pages of this thesis, the extra length will hopefully pay for itself in improved comprehensibility.

For the aforementioned reasons, the thesis is intentionally structured so that each single chapter deals with one major aspect of the work and starts out with basic introductory sections that lay the foundation for the readers not familiar with the topic to be discussed in that part.

In case someone is already familiar with the terms and introductory aspects, he or she can easily skip the first sections and directly proceed to the more challenging sections. The topics of the chapters themselves are structured in the same way. At the beginning of the thesis the rather "simple" aspects are discussed and described. They build the necessary framework for the later chapters, which then focus mainly on the important and essential parts of the work, like, for example, implementation or domain decomposition.

As a figure or a deliberate example saves a thousand words, the author uses many figures to illustrate important issues rather than describing them in lengthy and "boring" text passages. Also, Peter and his employee Paul, who both have already been introduced in the motivation chapter, will accompany the reader for quite some time through this text, always helping to present complicated matters in a coherent manner. In case the reader gets lost at some point while reading the thesis, references throughout the text will help him or her to easily locate the appropriate pages or the basic concepts in the introductory chapters. Additionally, a full index will be provided to quickly find certain topis within the flow of the text.

The following gives an overview of the content of the individual chapters in this thesis:

- **Motivation** - This first, short chapter can be thought of as an "appetizer" elucidating why, in general, high-performance computing is an interesting topic in itself and where it is already an integral part of our daily life.
- **Chapter 1** - This chapter gives a very brief overview of the work of the thesis itself without diving into the specifics too deeply. Furthermore, a detailed listing of the structure of the thesis is presented.
- **Chapter 2** - In this chapter a basic explanation of discretizing real-world problems is provided as well as a short description of how to process scientific models until they can be solved on a computer.

- **Chapter 3** - This is the longest introductory but very basic part of the thesis. It doesn't help to talk about parallel computing and all the necessary steps related to the parallelization process if typical basic terms, architectures of the parallel machines and certain techniques are not known or understood. In this way also non-computer scientists can gain a better and clear understanding of the various aspects and challenges in connection with the different steps of parallelization. As this thesis cannot serve as a general textbook on the many different but important topics introduced here in this chapter, each section is terminated by recommendations of literature related to the matter discussed in the section.

- **Chapter 4** - In this chapter, the basics of modeling in general are touched on as well as an exemplary modeling software package called Geosys/Rockflow. The chapter is concluded by a brief introduction to the rough structure of a sequential modeling program in general and the necessary changes when switching to parallel modeling.

- **Chapter 5** - The problems of domain decomposition, load balancing and communication aspects are intertwined; changing one of them will directly affect the two others. Therefore, this chapter starts out with a short introduction to the theory of domain decomposition before explaining all necessary aspects of load balancing, domain decomposition and communication issues by using practical examples. Furthermore, two common and widely used tools for subdividing existing models, which have also been implemented in the wrapper program, are described and compared to each other.

- **Chapter 6** - This chapter is clearly the longest but also the most important one, dealing with the implementation issues in all their facets. It is the core of the thesis. One major block deals with the problem of memory consumption. Depending on the implemented data structures, direct effects on the usage of memory result. This can lead to problems, depending on the size of the input data of the models. Three different types of data structures are introduced and described as a possible solution to the preprocessing of this model input data. An additional but optional statistical output helps in assessing the produced data. Over a transparent simple-to-use commandline interface the user can preprocess all his or her data without knowing any of the tiny details of preprocessing, which are completely hidden.

- **Chapter 7** - This last chapter describes the outcome of preprocessing. Here the achieved results are compared in dependence of several factors. The first major part deals with the domain decomposition tools *Jostle* and *Metis*. The second part concerns the three different data structures that were examined and partially implemented and tested in the wrapper code. Their advantages as well as their drawbacks are discussed and compared. A last part deals with future improvements and ideas that could be implemented in a future version of the code.

- **Appendix** - The appendix at the end of the thesis comprises several parts. It consists amongst other things of a listing of all abbreviations used throughout the text, a few exemplary files used as input for the wrapper and its resulting output, the user manpages that are displayed when applying the software developed during the thesis, as well as an index with most of the terms and topics introduced in this text. The interested reader will also find a long listing of relevant literature with complete references and sources.

Chapter 2

From a Real-World Problem to a Computable Model

Modeling complex coupled transient and dynamic processes found in geo-systems requires increasingly high-resolution models for a more precise and qualified validation and evaluation. The more exactly the investigation of the given problem can be accomplished, the better the utilization, management and planning of the given geo-resources can be undertaken.

The above statement is definitely correct and sounds very convincing and reasonable, but before the scientist is able to start this modeling process to find a solution to it on a computer as a final step, the problem as it exists in the outside world itself has to be transformed in such a way that it can be worked on at all. The important steps on the way from a real-world problem as it presents itself to the researcher in daily life to its final representation which is used finally in the chain of processes to compute a possible solution on a computer is the focus of this introductory chapter.

In the meantime, 3D computer models have become quite common in daily scientific life and many professionals and experts deal with them in the course of their work. Unfortunately, although many of them are used to working with the models themselves, many have no real idea how they are produced and why a careful and thorough transformation is very important to accomplish a successful computation.

In this chapter, the reader is taken by the hand from the very moment of problem discretization at the beginning of the problem description up to the point of the final model processing on a computer with a special program. Computational and scientific modeling is a wide field in itself and many aspects have to be considered when working with such sophisticated modern models. Therefore, this chapter can only "scratch at the surface" of this interesting research area, but at the end of it, the reader will have enough understanding of the important basics of computer models and their processing to proceed to the upcoming chapters, in which these basics then are needed.

2.1 The Process of Discretization

When dealing with typical real-world problems in all areas of geology, the scientists and researchers normally examine continuous problems. This means the variables and parameters that

are regarded can increase and decrease to any possible value in between the possible minima and maxima of the specific problem. Unfortunately this results in an unlimited number of values. A typical but simple problem, as can be found in the field of applied geology, for instance, is shown in Fig. 2.1.

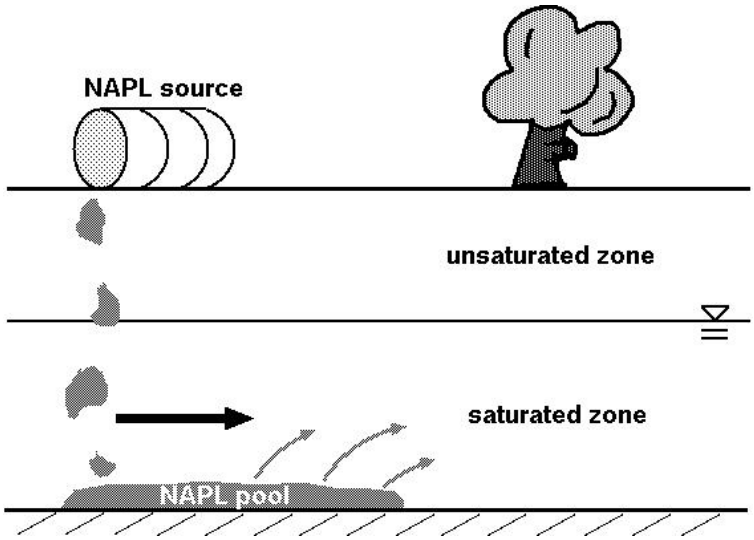


Figure 2.1: Simple groundwater contamination scenario due to environmental pollution by NAPL

In the scenario depicted in Fig. 2.1, a barrel with a pollutant liquid, in this example a NAPL (non-aqueous phase liquid), is poured out on the ground surface. The toxic material finds its way down through the different layers in the ground until, at some point, it reaches the groundwater table and the water-saturated zone. Depending on whether this NAPL source was a LNAPL (light NAPL) or a DNAPL (dense NAPL), different possibilities arise as to what is going to happen next. In case of a LNAPL, the material will most likely swim on top of the aqueous phase and thus possibly be transported far away from the originating source of contamination. Such LNAPLs could be for example, certain carbonhydrates like gasoline or oil which could have percolated out of an old (oil) tank. In this way long-term sources of pollution can exist without being detected. Their plume can be carried away and spread out far from where the tank is located.

If, instead, the liquid is a DNAPL, it sinks down, as it is not soluble in water and also has a higher density than water. The organic parts start to dissolve very slowly over time, varying from a few 100 ml/l up to some 1000 ml/l in decades or centuries. Depending on the setting where this happens, even very dangerous side-products can result from chemical reactions. Sometimes the DNAPLs start to sink into the ground and to settle in the bedrock. Their pollution potential is very high and they can also be a long-term source of serious environmental pollution. Depending on the chemical, very few countermeasures are possible, like, for example, pump-and-treat or reactive systems.

No matter what material is actually polluting the environment, it is definitely clear in the above scenario that the pollutant is spreading everywhere around the location in which it was poured out. This is also true in case a poisonous plume begins to develop or when the NAPL reaches the bedrock and starts to percolate into the pore space between the grains of the native rock.

2.1.1 The Difference between Discretization, Model and Modeling

When now examining such a situation as depicted above in reality, the scientist working on this case just might have a little information about the true amount of pollution from a couple of boreholes, if he or she is lucky. There might also be some results from pumping tests or other investigative methods available, depending on the case itself. But in the end, when modeling such a scenario, one goal could be to produce a complete 3D model showing, for example, the plume of the pollutant fully spread out in the ground, but one could also be interested in finding out about the timely matters of the pollution and not the actual 3D shape of the plume.

How close the final model is to the real-world situation depends strongly on the amount of data available, how these data are treated during the modeling process, etc. As it is simply impossible to measure every single micrometer in such a situation like the one with the plume, beginning at the surface and following the plume and pollutant all the way down to the layer where it has finally settled or to where the pollution ends, or to follow the pollution in all possible directions, the scientist has to discretize the problem.

Discretization means that one has to extract or obtain a finite and discrete amount of data from a continuous set of information. As continuous data sets are always uncountably infinite sets, it is impossible to process and solve such problems in reasonable times and with existing memory space. Only at this level of abstraction, by the discretization of a given problem, can significant models be established and computed at all.

It should be noted that one should not confuse the terms model, modeling and discretization. When we have a certain situation in real life, like the one depicted in Fig. 2.1 previously, we need to discretize the spatial information, which would be the polluted ground, as it is impossible to work on every single millimeter in such a scenario. We need a digital representation of this area, where the pollution has taken place. One could picture this as some kind of “snapshot” of the actual scenario at a certain time. We transform our 3D world into a digital representation that only consists of discrete data.

The process of discretization transforms the real 3D world scenario into an artificial, less detailed representation. The level of simplification which signifies how much of our original information will be lost during this transformation process, depends on the type or method of the modeling process that will follow.

In the real-world scenario we have not only “visible” information like the size of the plume, or the type of the pollutant, etc., but everything that happens there underlies certain physical laws. Some information we cannot directly see but we know from chemistry how a compound might react with other substances, that gravity does not stop all of a sudden and that rocks have typical characteristics, for instance.

A model is a representation of a system which one uses to describe, for example, the fact that the plume is spreading out in the scenario depicted in Fig. 2.1. But the model could be the simple 3D representation of the current state of pollution. There could also be just a chemical model examining the chemical reactions that are taking place but not considering the time factor or the material surrounding the chemicals, etc. Thus, a model in science is a physical, mathematical or logical representation of a system of entities, phenomena or processes. It is a simplified abstract view of the complex reality. Depending on the focus one has, the model could always look different.

Modeling now is the process of generating such abstract, conceptual, graphical, mathematical, etc. models. To do so one can use a diversity of tools, a collection of methods or techniques and theories to finally set up a model he or she is interested in. If someone is already content having a perfectly detailed 3D "picture" of, for example, such a pollution scenario, he or she would have a graphical model, which can be obtained by the process of discretization. If another person is rather interested in the processes which are hidden in such a scenario, he or she would most likely not concentrate on a appealing 3D representation but rather try to find the best-suitable functions and equations to describe such processes.

Therefore, we can have many different modeling techniques resulting in different types of models and thus, the modeling itself also varies.

One common modeling technique is modeling using finite elements (FE). This method is based on spatial information. There are many other modeling techniques, like artificial neuronal networks or cellular automata which rely on completely different information. As the modeling software which is be transformed from a sequential to a parallel version (see also chapter 4, starting on page 69), is based on the FE technique, the further steps and actions necessary for this type of modeling will be regarded.

2.1.2 The Basic Units: Vertices and Elements

When someone would like to deploy the FE method for modeling, the underlying model has to be set up in a certain way. When discretizing a given real-world problem for the later FE model, its problem domain is replaced by a *grid* or *mesh*, which is a collection of *nodes* (also called *vertices*, singular: *vertex*) that are connected to each other in a certain way over so-called *edges*. How the nodes are combined depends mainly on the dimension of the problem domain. Through the connection of the vertices, different geometric objects result which can be axisymmetric or not. These geometric shapes are called *elements*.

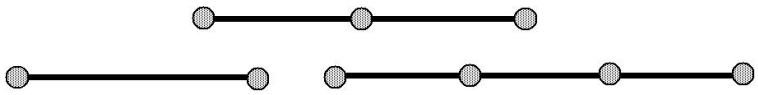


Figure 2.2: Simple one-dimensional grids consisting of nodes and one-dimensional elements (= lines)

Fig. 2.2 depicts three very simple 1-dimensional grids consisting of nodes, which are the gray dots in the figure, and elements, which are simply lines in 1D. In the case of a 1-dimensional grid the connecting lines between its vertices, the edges, are at the same time also the elements. Unfortunately, this often leads to some confusion. They are the same because just one dimension is involved.

The elements that evolve in a 2-dimensional mesh are already a little bit more complex in comparison to the simple lines in 1D. The 2D elements can be of either shape; they can be also axisymmetric but needn't be. Some examples of such 2D elements are shown in Fig. 2.3.

The geometric elements that can be found in a 3D scenario show an even higher complexity than those in a 1D or 2D scenario (there can be also 4D, 5D or 11D scenarios). A very simple 3D element is, for example, a cube which results when connecting 8 equally spaced nodes. This geometric shape is completely axisymmetric, but there are also element shapes possible that do not show any symmetry at all. Some exemplary 3D elements are depicted in Fig. 2.4.

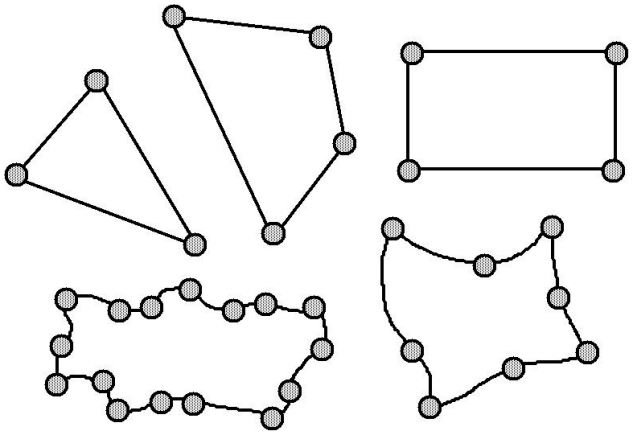


Figure 2.3: Simple two-dimensional elements, partly axisymmetric, are built by connecting a set of nodes

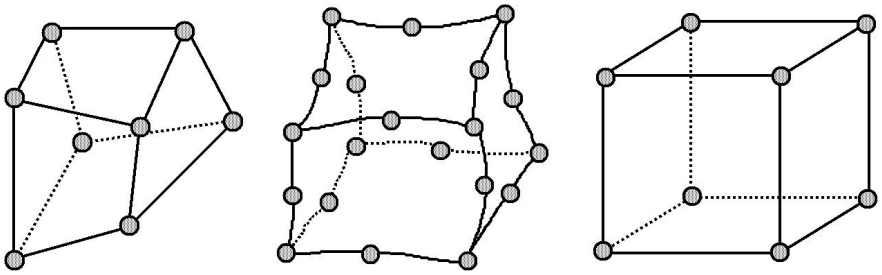


Figure 2.4: Simple three-dimensional elements, each consisting of a different set of nodes and possibly showing axisymmetry like the cube

An element should not be confused with an edge! In a 1D scenario, as depicted in Fig. 2.2, the elements and the edges happen to be the same because we have just one dimension we are working in, but in all other cases an edge is something completely different than an element. The element consists of nodes which are connected by edges. All of this together forms the element.

The elements within such a grid may be of any size; furthermore, within the same mesh the elements may differ in not only size but also shape. In Fig. 2.5 a very simple discretization of a simple 1D problem domain is shown. The necessity of different lengths of the single elements (here: lines) within this grid can be easily seen.

Normally the problems to be modeled in geoscience are more complex and have at least two dimensions. A typical and daily example as can be found in hydrogeology is depicted in Fig. 2.6. Generally, the water table starts sinking when taking water out of a production well, and the effect of *coning* can be observed. When ceasing the water extraction, the water level will normally slowly rise back to its original level. An observation well is often used to examine the differences in the water levels while pumping.

When modeling such a pumping scenario, mainly the observed differences in the aquifer are of interest and especially here, in this example, the effects of coning. A discretization of this pumping and coning problem can be seen in Fig. 2.7, where the original scenario is reduced to a simple 2D mesh. As the water table shown in the figure is normally variable, one has to make sure that such information is also represented when necessary in one way or another in the resulting model.

In Fig. 2.8 a slightly different discretization of the pumping scenario is shown. Here, a 2D grid is still used, but now the elements vary in size and shape depending on the placement of the nodes

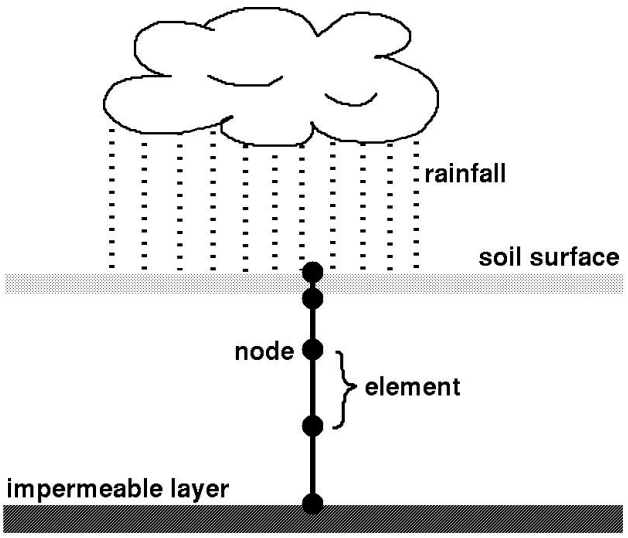


Figure 2.5: Discretization of a simple one-dimensional problem domain

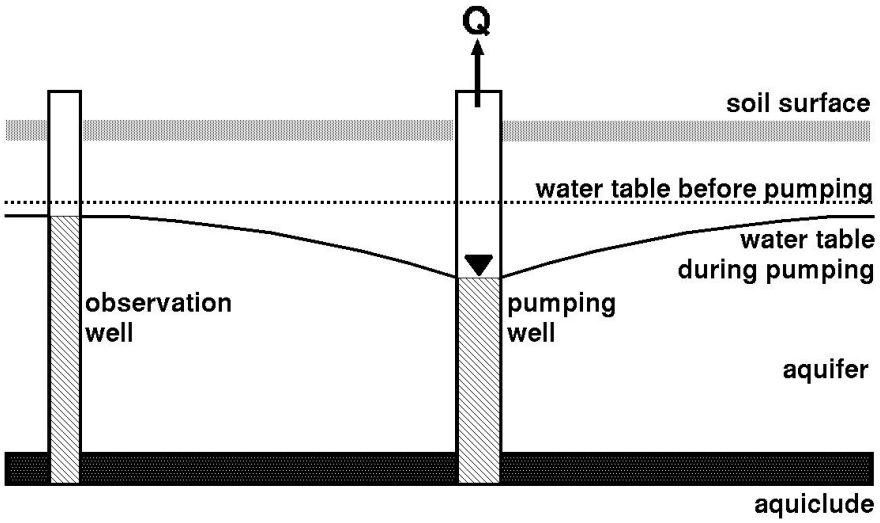


Figure 2.6: Variations in level of the water table before and after pumping water out of a well

within the grid. As the rate of change in value most likely increases the closer one gets to the pumping well, it is more reasonable to reduce the size of the elements where the parameters are going to change faster in the problem domain. This way the modeling and thus, the outcome, is more exact later on when processing the existing data.

In many cases though, 3D grids are used to model a certain problem domain, and the existing forces operate normally in all dimensions. Depending on the geometric shape of the elements used during the discretization of the problem domain, all kinds of 3D grids can be built which might even result in a pseudo-realistic appearance of the resulting model. In Fig. 2.9, a very simple 3D grid is shown for demonstration purposes.

Depending on the method which is used later on during the modeling and computation process, the vertices and elements are assigned parameters, information or certain values that have to be stored and which are also needed in the later processing stages. The smaller the elements and the more closely the grid points are located to each other, the more values, etc., can be processed.

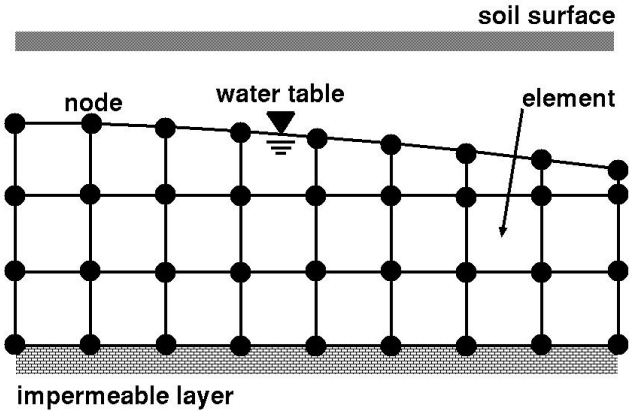


Figure 2.7: Discretization of a pumping scenario with a coning effect for a simple 2D grid

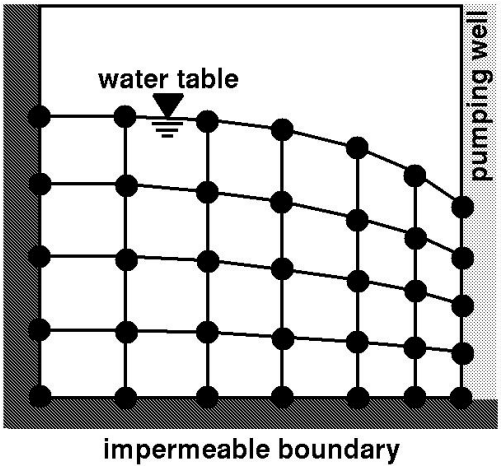


Figure 2.8: Variable placement of nodes in a 2D problem domain where the field variables are expected to change rapidly

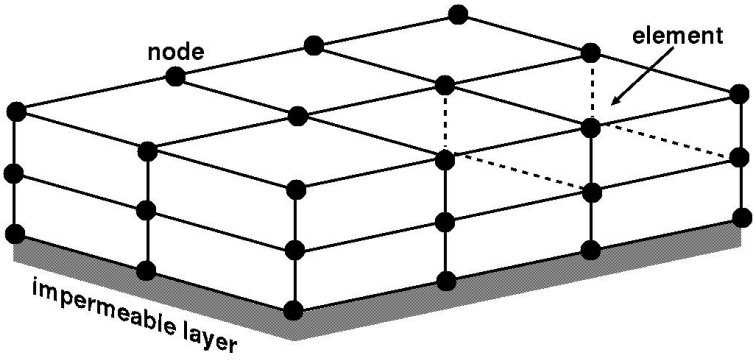


Figure 2.9: Discretization of a simply-shaped three-dimensional problem domain

The resolution of the model is increasing (the actual number of parameters does not change). This results in finer and more exact models with a more realistic outcome in comparison to meshes where the grid is very simplified and coarsely structured. Unfortunately the more sophisticated and finer the grid is (which generally means a much higher node and element number), the greater is not only the storage space needed for the model but also the increase in computation time.

There is no typical rule of thumb as to what the perfect size of the grid or useful number of vertices would be, as much depends on the scenario and the problem to be computed as well as the geometrical shape of the elements and the grid, etc. Just by refining the grid itself, by increasing or decreasing the size and its node number, for example, one can finally figure out the most convenient grid structure for a specific problem and its modeling that yields the best and most useful solution to a given problem scenario.

Quite often simple but effective tricks can be used to minimize the number of nodes and elements and thus disk storage without losing important information. A simple example was already shown previously in Figs. 2.6, 2.7 and 2.8. In Fig. 2.6 the change in the water table towards the pumping well and the coning effect appeared all around the well; in the picture as a 2D model, therefore, it could be observed on the left- and right-hand side of the well. In Figs. 2.7 and 2.8, just one side was discretized and not both, as the effect of symmetry can be used effectively. It is completely indifferent and of no importance to the problem whether one considers the left side of the well or the right side here. In reality the coning problem will appear 360° all around the pumping well but normally it will be the same in all of the 360° directions. In this way, already a great deal of data storage can be avoided without sacrificing precision or accuracy. Fig. 2.10 shows another example of an effective usage of symmetry effects.

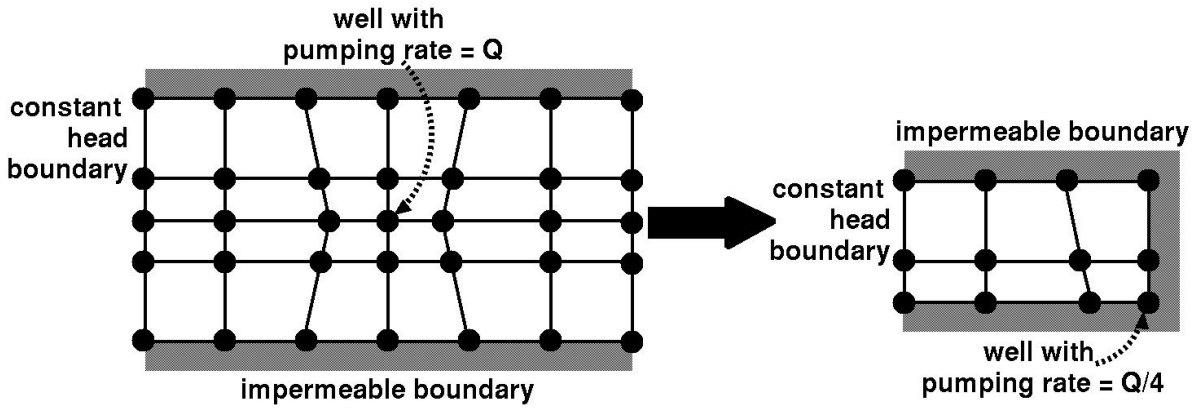


Figure 2.10: Usage of symmetry effects to minimize mesh size

In Fig. 2.10, only a quarter of the original problem size has to be modeled and processed later on, also reducing the computing time most likely to a quarter of the processing time of the whole mesh shown in Fig. 2.10. The actual time saving depends on the modeling method and myriad other aspects but discussing these might lead too far at this point here. A very detailed and nice description of modeling can be found in [Istok], for instance.

What can be applied to 2D models is of course also valid for 1D and 3D models and their discretization. A simple but effective example is shown in Fig. 2.11.

In the example shown in Fig. 2.11, the same composition for the ground is assumed in 360°: A sandy layer is covered by a silty layer on top. Instead of now modeling the whole block around the well, just a “thin slice” is cut off the 3D model which is used for the computation later on. This is possible by performing a transformation of coordinate system from a typical Chartesian coordinate system with x, y and z to a polar coordinate system which relies on angles, a radius and z. And again it is completely indifferent which direction is chosen for selecting the “slice” for modeling.

The success and the final outcome of a model processing depends not only on the modeling method

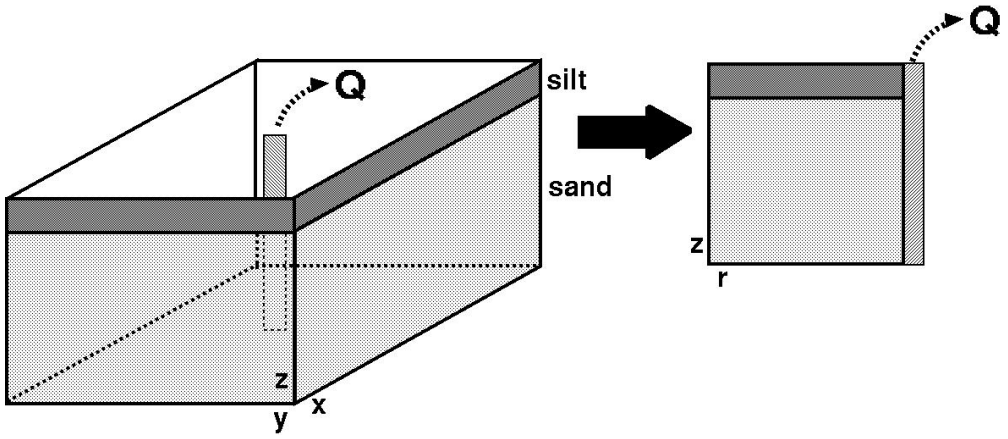


Figure 2.11: Usage of symmetric effects to reduce model dimension during discretization

and the structure of the model itself but also on the care and diligence exercised when discretizing the problem domain. A few but very important rules have to be obeyed when breaking down the problem domain into discrete nodes and their connections which shall be used later on in a FE model. The following figures will give a quick overview of correct and incorrect discretizations. (Some of the restrictions might not apply when discretizing for a different type of modeling technique.)

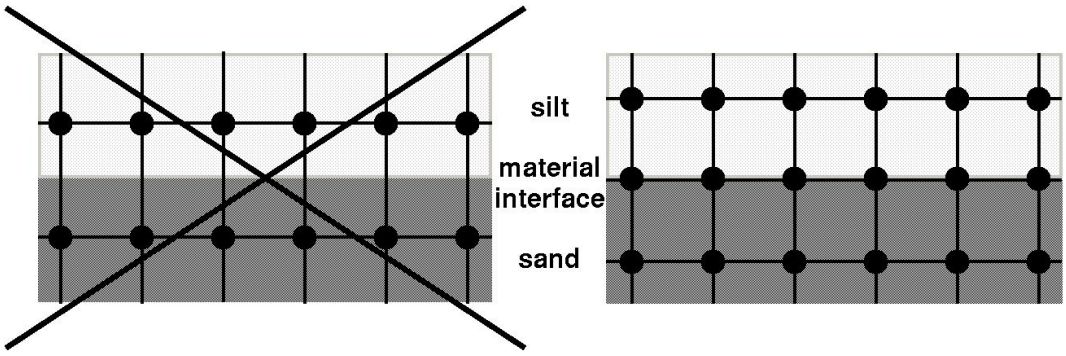


Figure 2.12: Incorrect (left) and correct (right) nodal placement at the interface of two different materials

In Fig. 2.12 the importance of following naturally existing borders and interfaces is shown. On the left-hand side of Fig. 2.12, the nodal placement is not on the exact border between silt and sand. On first glimpse this doesn't seem much of a problem. Later on during the modeling process, each of the elements that are located along this border line have to be assigned one single value, either sand or silt. But when the interface between these two materials is now, for instance, precisely in the middle of such an element, which material property should the element be given? Sand? Or rather silt? There is no "both;" only one concrete value can be chosen. Therefore the right-hand side depicts the correct placement of the vertices in the grid. The material interface is now exactly between the elements. This way the element has now one specific value; it is either sand or silt, and during the computation can be treated correspondingly.

In Fig. 2.13 two other dangerous mistakes are shown. On the left-hand side of the figure a simple grid is shown which contains a gap. Such "holes" in the mesh should be completely avoided in any case. This means not only that there will be inconsistencies in the model itself later on, but the

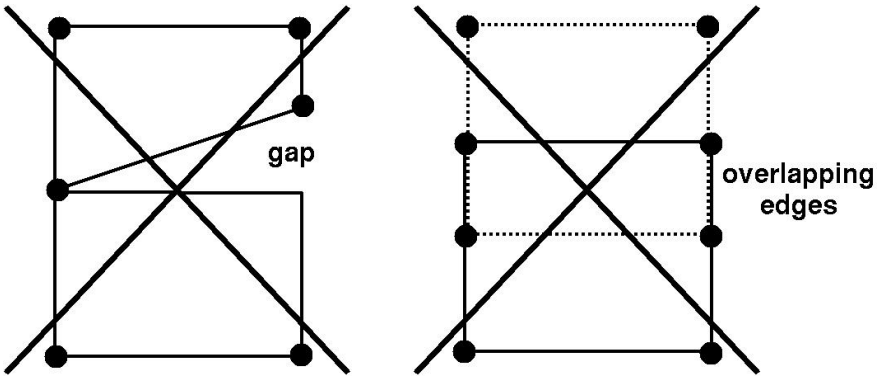


Figure 2.13: Gaps between elements as well as overlapping edges of elements are not permitted

same applies here as already indicated for the material interface. In case the modeling algorithm has precautions implemented for such gap situations it must assume some kind of value for these gaps. But which property or parameter would now be the correct one? Normally the algorithms should check for such inconsistencies and then issue an error message when a situation like this is detected. A proper outcome of the model computation with faulty meshes containing gaps is normally no longer given.

It is possible to model certain “holes” in a model, for example, containing a well. But generally, such areas are replaced by only one big element that represents the parameters or values that are connected with this hole. Such holes , however, are something different from true “gaps” in the model, where simply the necessary information is missing or the discretization process was not done properly.

On the right-hand side of Fig. 2.13 there is another unfortunate nodal placement. Here the elements overlap. Remembering the fact that elements do possess certain important values and information, the computing algorithm now again has the problem of assigning the correct value to the overlapping area. Given the exemplary case in which the upper element represents sand and the lower element silt, which of the two materials would now be present in the overlapping zone?

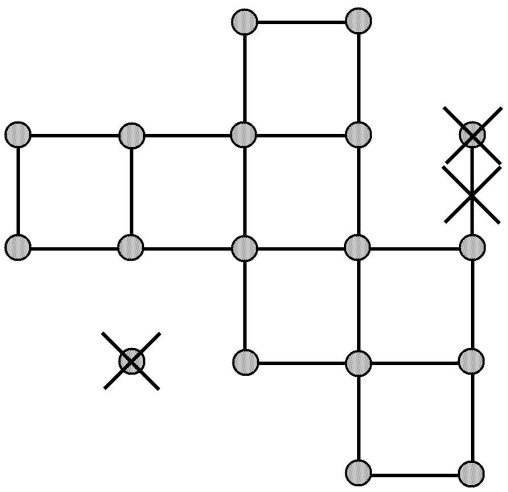


Figure 2.14: Detached vertices and incorrect element shapes are not permitted

Two other sources of error are shown in Fig. 2.14. Sometimes it can happen during the discretiza-

tion process that the algorithm suddenly places a single vertex outside the grid. But there is no element that it is attached to. The vertex holds its correct geometrical coordinates that are generally used when discretizing, but as no material property can be connected to this single node over an element, its existence does not make any sense at all. It is just another source of error. Furthermore, meshes or the models later on have a certain domain geometry as well as initial and boundary conditions. Such a single misplaced node would not comply to these, and in case the algorithm trapped such situations, the outcome of the model processing would most likely be incorrect.

It is possible to mix different geometrical shapes for the elements within one mesh. But in such cases the forms permitted are shapes that are typical for the dimension chosen in the grid. In this exemplary mesh in Fig. 2.14, squares are chosen as 2D elements. By attaching the single node to the right by a line, a 1D element is now used in a 2D grid. This mixture in element dimensions is not allowed.

Usually, current specialized software discretizes a given problem domain correctly where this "wisdom" of how to discretize the domain of a real-world problem is already implemented within the discretization algorithm. Nevertheless one should always be aware that software might also not be completely faultless!

As a well-suited closure for this section the following gives a short listing of the main checkpoints one should always keep in mind when discretizing a problem domain for a FE model to minimize the sources of error and to maximize the outcome of the model computation:

- The size and shape of the elements in a mesh are determined primarily by the size and shape of the problem domain itself, the number of different types of materials within the model and of course by the number of nodes in the mesh. In problems that already have a naturally complex geometry, for instance, an irregular depth of a bedrock, or through the geologic structure itself, for example, the presence of faults, many elements will be required for the realistic modeling of this problem domain. In problems with a simple geometry, like a shallow alluvial aquifer which is underlain by a regularly layered horizontal bedrock, fewer elements might be required instead.
- If the problem domain contains curved boundaries or interfaces that are to be modeled, different types of elements should be used in comparison to the case in which the boundaries and interfaces consist of straight lines or planes.
- Elements should be generally smaller in size in those parts of the mesh where the field variable is supposed to change rapidly. This results out of the fact that the nodes are placed closer together in such areas.
- Each element in the grid is defined by two or more nodes. These nodal coordinates determine the size and shape of the element. (For this reason the node numbers for each element are normally listed.)
- The material properties in a model are specified using the elements. Therefore each element will be assigned definite values, which might lead to the fact that some elements can have identical values.
- If possible, the simplest type(s) of element required to characterize a particular problem should be used. This usually means employing linear bar elements in 1D problems, linear, triangular or rectangular elements for 2D problems and linear parallelepiped elements for

3D problems. However, one should not hesitate to use more complex elements, especially when curved boundaries or interfaces are encountered.

- Edges of adjacent elements should never overlap.
- There should be no "gaps" in between the elements of the mesh.
- Material properties are assumed to be constant within an element, but they are allowed to vary from one element to the next. Therefore elements should not overlap an interface between two different types of material.
- As the shape of the elements affects the accuracy of the resulting model solution but also the computation time and storage space, highly distorted elements should be avoided whenever possible. Depending on the method and algorithm used for the model computation, the element shape influences the size of the time step that is required to obtain a stable solution.
- In case the problem geometry requires a change in element shape and size, one should not attempt to switch to a different element size and shape abruptly but use, if possible, a gradual transition from one type to the other, maybe even over a transition area in the grid.
- By using the advantage of symmetry in the problem domain, one should try to reduce the number of elements and nodes in the mesh. When doing so, it should be kept in mind that the boundary and initial conditions, the material properties and the domain geometry must all display this symmetry for such an approach.

2.2 From a Discrete Model to its Software Representation

Generally, actual and daily problems which researchers and scientists are working on nowadays often comprise challenging mathematical equations and algorithms for their solution. It might be possible for an average desktop machine or out-of-the-box computer to be able to compute such ambitious models. Unfortunately our real-life problems are very complex, and the more accurate and precise the solution, the more demanding the requirements for the processing environment. Some of the models are so complicated and complex that special hardware is needed to find a solution at all, like, for example, the Earth Simulator, which will be briefly introduced in subsection 3.6.9 of chapter 3.

For the computation of the solution of these models, generally big equation systems, matrices or complex differential equations have to be solved. Unfortunately, the mathematical expressions used normally depend on each other or share variables or intermediate solutions. One can't just sit down and simply "chop" these equation systems or matrices into single suitable pieces like a cream cake. This would directly lead to unusable results or completely false equation solutions and iterations of the problem.

Let us take, for example, a 3D model of a real existing landscape like the one shown in Fig. 2.15. It depicts the geological 3D model of the Jordan Valley with all its faults, elevation, etc. Such good-looking copies of reality are built as described previously from a system of grid nodes in a mesh that are connected by edges and, thus, form elements. Depending on the problem which has to be solved in regard to the model, the single nodes or elements within the grid are combined with different information and important data. Here one could imagine all kinds of information like, for

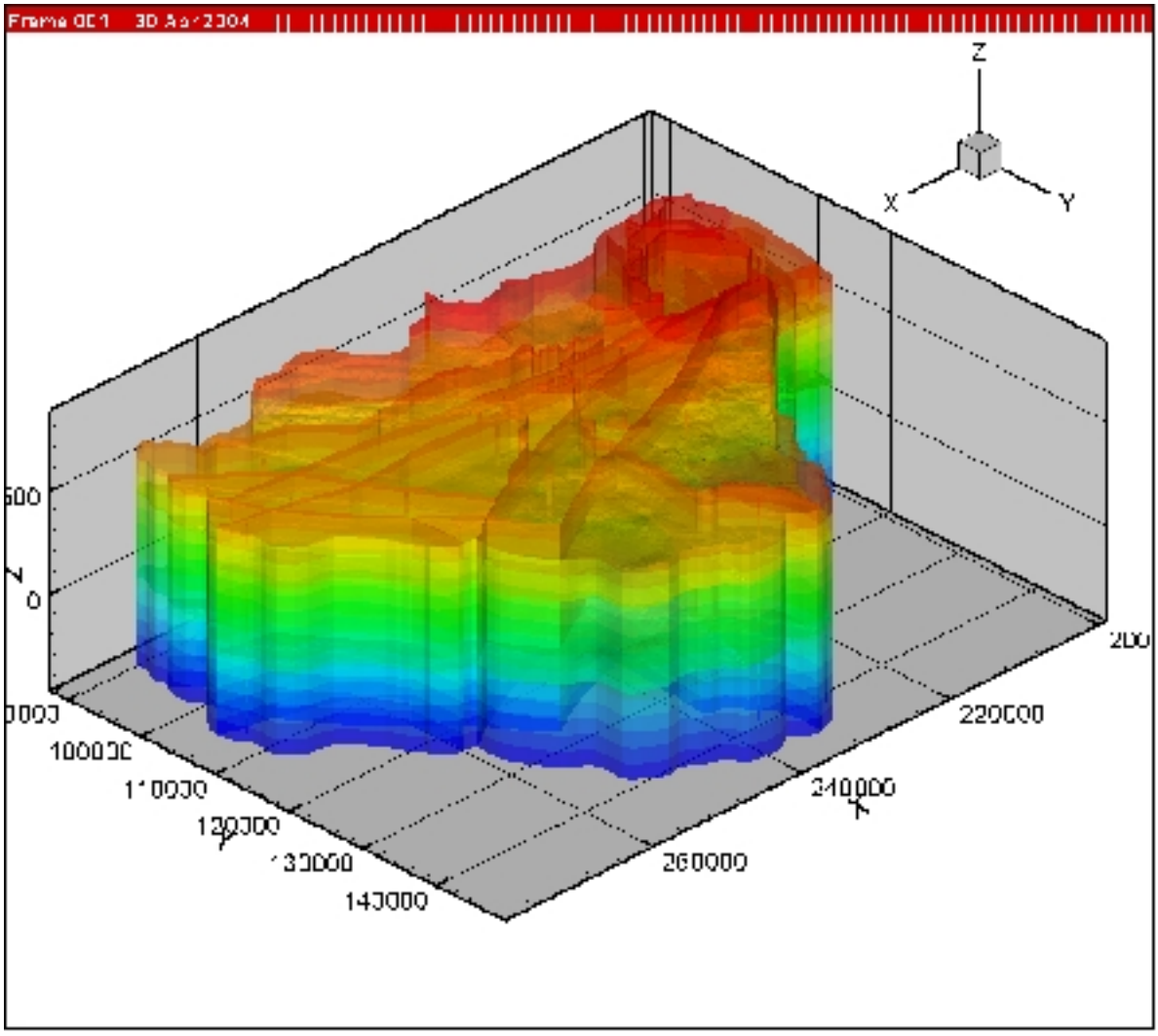


Figure 2.15: 3D model of the geology of the Jordan Valley [Chen]

example, groundwater levels, the material that was found when drilling, elevation, fluxes or flow directions, just to name a few.

For such meshes to be computed in a program, they have to be broken down and be read in onto the processing machine, depending on how they are to be further processed there. But it is imperative that no relevant (node) information is lost during this preprocessing. The existing mesh data is normally used to build the matrices or equation systems which then have to be solved by the software to find a solution to the model. In the same way that the nodes in the grid are connected to each other to set up such a 3D model like the one of the Jordan Valley shown in Fig. 2.15, the relevant data attached to these grid elements is connected later on through the matrices and equation systems.

In Fig. 2.16 the different but still simplified steps that have to be taken are shown, starting out with a useful 3D model of our real world to a final, representative mathematical model, like, for instance, a *finite element* representation. Unfortunately we are still not at a point where the information hidden in such models can be processed. This next but very important step is represented in Fig. 2.17.

When processing numerical problems on a simple desktop system or a normal "out-of-the-box" computer, the responsible programmer or software engineer just has to know the correct "transla-

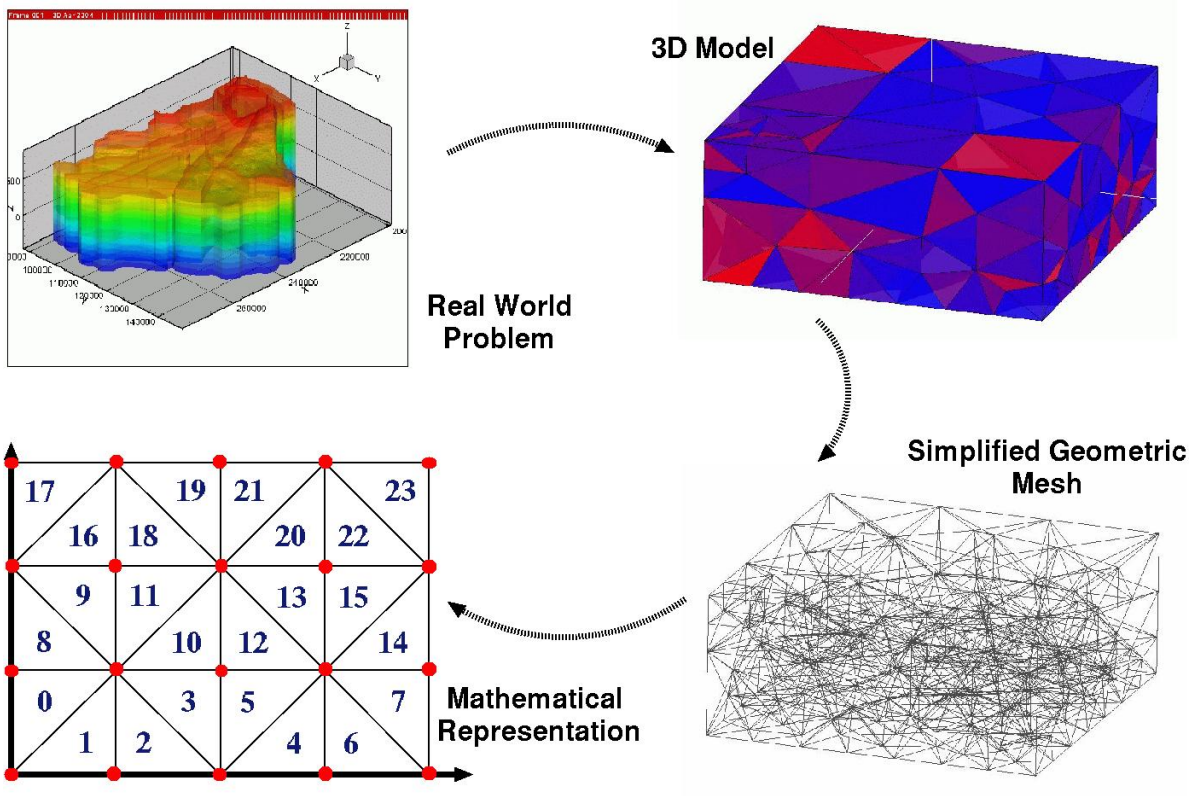


Figure 2.16: The long way from a 3D model to a mathematical representation

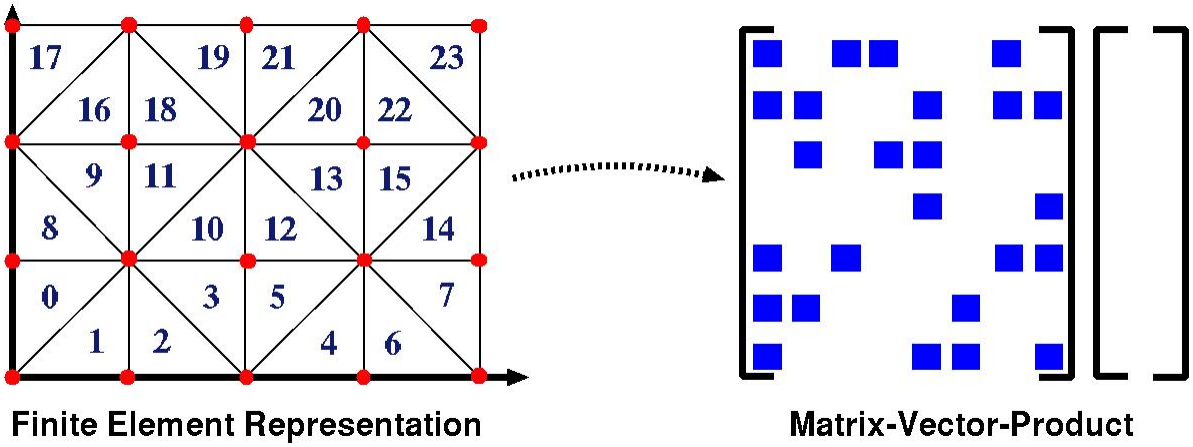


Figure 2.17: The finite element representation must be mapped into a matrix-vector product

tion" between the necessary steps, beginning with the real-world problem at the very beginning of the process, until he or she finally ends with the matrix-vector product as shown symbolically on the right-hand side of Fig. 2.17. When reaching this stage in the "model translation process," such a mathematical construct, like a big equation system, for example, can now be broken down into code and solved when running the software. Fig. 2.18 summarizes the main steps of this whole process, when working with a standard computer which has just one single processor.

If specialized hardware for the computation of the models or the underlying equation systems is needed, the previous paragraphs should have made it clear that it is simply impossible to just tear these models apart into suitable, handy pieces. The data is distributed within the matrices

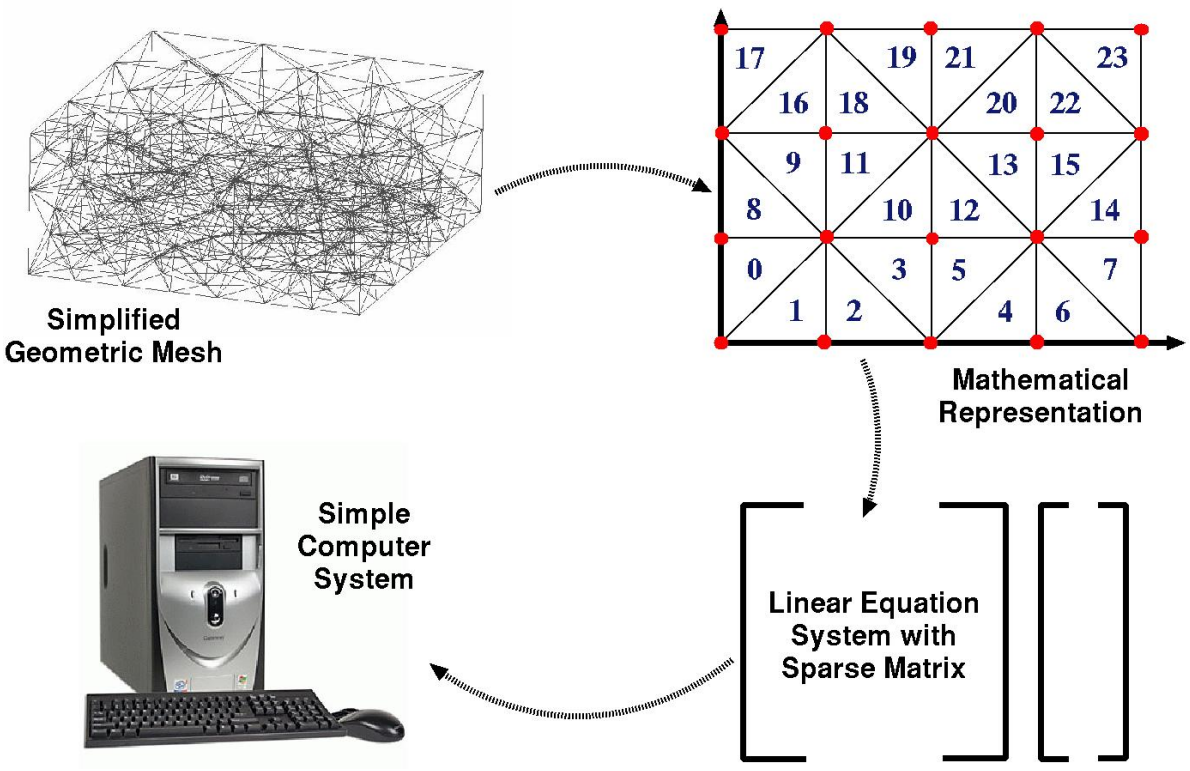


Figure 2.18: The main steps in the whole process of translating a 3D model into a computable format using a standard computer system

and cannot simply be cut into suitable "chunks." In case a supercomputer- or multiprocessor-environment is necessary for the computation of the model solution, further specialized algorithms and methods are necessary.

This will be the main focus of chapter 5. Here, important aspects will be discussed that have to be taken care of when the need for a subdivision of such complex models emerges. Also tools helpful in such a case will be introduced briefly. But before diving into quite specialized issues already, the next chapter will lay a simple foundation for the construction of a computer in general and give a short introduction to the various facets of parallel computing in general.

Chapter 3

Basic Terms and Concepts of Parallelization

When talking about "parallelism" it quite often happens, even among computer scientists, that basic concepts and technical terms of the field of parallelization are used in an inappropriate way. Often programmers or engineers express what they "feel" or "think" about a certain topic in regard to parallel aspects, but frequently such terms and fundamental concepts get "mixed up."

With a solid foundation and a correct classification of the basic terms and aspects in regard to parallelization which are presented here as briefly as possible, the reader of this thesis has a good starting point for the further discussion in the upcoming chapters.

3.1 The Basic Units of a Computer

In [Duden] one can read that in 1946 *John von Neumann* already suggested a concept of how to design a universal computing machine to meet all requirements imposed by commerce, science and engineering. Even today almost all modern computers still show this basic structure of so-called *von Neumann architectures*. The *von Neumann design* comprises the following 5 fundamental units, which are depicted in Fig. 3.1:

- Input unit
- Output unit
- Arithmetic logical unit (ALU) with accumulator
- Memory
- Control Unit

The different units are attached to each other over so-called buses or bus systems. These are very fast network connections which are realized in hardware. Furthermore, external devices like a keyboard, a monitor or a printer are installed in order to use this very simple computer effectively.

The single units of the *von Neumann* architecture already speak for themselves: The input unit is used to feed data and programs into the computer. The output unit is necessary to access in some way the results of the programs or routines which are processed on the machine. In the memory,

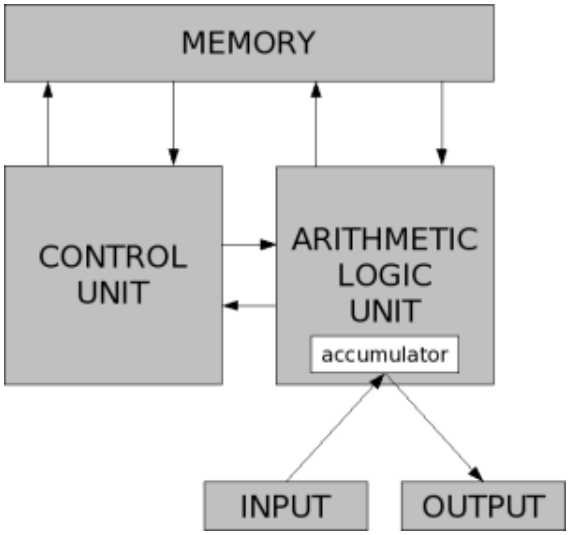


Figure 3.1: The *von Neumann* architecture from 1946

programs and data are stored. Therefore a *von Neumann* computer can also be called a *stored-program computer* interchangeably. Finally, the two last remaining units are the most important ones. The ALU is simply comparable to a "calculator" that itself uses something like a "short term memory," namely the accumulator (= very fast registers).

The second significant unit is the control unit. It coordinates the handling of data and programs. It keeps track of the progress of processing programs, hands over necessary data to the ALU for mathematical computations, makes sure that keystrokes will be recognized quickly or that the mouse can be clicked. This effect is shown directly on one's attached displaying device.

The ALU and control unit together are referred to as the central processing unit (CPU), as this is where everything finally connects. The CPU takes care of the proper administration inside the machine so that all units work together.

It is clear that nowadays much more specialized hardware is helping the CPU accomplish all these tiny little administrative tasks. Still, without a processor, all this special hardware and such devices as graphic cards, network interfaces and pointing or clicking devices, just to name a few, would be worthless. The processor is the "heart" or "brain" of the computer.

Interested readers with regard to basic computer hardware, internal processes within a computer and fundamental computer science basics can have a look at [Ortmann]. This book is a good start for newbies, easy to read and quite entertaining at the same time.

3.2 The Different Levels of Parallelization

Before talking about the different levels of *parallelization* the term itself should be defined. The way in which it is understood in the following sections is as performing certain operations simultaneously, which is in parallel. Sometimes this is also referred to as *parallel computing*.

When talking about parallelization, one approach could be to determine the different levels at which parallelization can be found. The following subsections show where parallelism can be determined and clearly set the limits among each of the levels. The lowest level of parallelization

is at the bit level, with which this section starts out. From there on the parallelization level becomes more and more abstract, up to the job or program level.

3.2.1 Parallelism at Bit Level

As is commonly known, the smallest unit in computer science in which "useful information" can be stored is one byte, consisting of 8 bits (a bit being the customary abbreviation for *binary digit*). If a processing unit now wants to work on one data byte, it has two possibilities of transmitting this information into the CPU: Either one has just one single data bus capable of transporting 1 bit at a time, which means the byte arrives at the CPU after 8 clock cycles (one for each bit), or, one installs 8 data buses which can transmit 8 bits in parallel at one clock cycle, whereby the CPU has an idle time for just one clock tick instead of 8.

The early computers were also known as 8-bit computers. Their data buses could transmit only 8 bits at a time. Meanwhile the data bus width has increased successively to mainly 64 bits! Hardware techniques have been developed which even interlock the transporting of the bits to double the possible throughput again at just one single clock cycle.

3.2.2 Parallelism at Instruction Level

One can easily imagine that the administrative tasks on a computer during processing are manifold. It takes time to find the correct line within a program, for instance, after a branch command. Another example is the different cache hierachies that cheaper modern desktop computers already provide: The loading of data from the hard disk takes "endless" clock cycles (from the perspective of a CPU) during which the CPU itself could accomplish so many more useful tasks instead of waiting until the data has finally arrived at the processor.

To overcome this disadvantage and to optimize the internal processing in a computer, different techniques are implemented which all provide parallelism at instruction level.

Pipelining

In most modern programming languages, like C or Fortran, single commands are internally split up into various microcommands or machine commands. Transporting data within the computer, for example, has at first sight nothing to do with adding two integers within the ALU. Therefore, microcommands indicate where in the computer's memory the next required datum is stored. Transporting this datum from this memory location to the processor can be intertwined and thus executed simultaneously with the loading of the next command in one's program. While the specialized hardware for memory manangement is busy fetching the data, the processor now has time to take care of the next program step. By interlocking all these various necessary microcommands in a clever way, a so-called *command pipeline* or *instruction pipeline* can be set up. Figure 3.2 shows such a pipeline with 5 microcommands.

In the example given in Fig. 3.2, the following 5 operations are independent of each other:

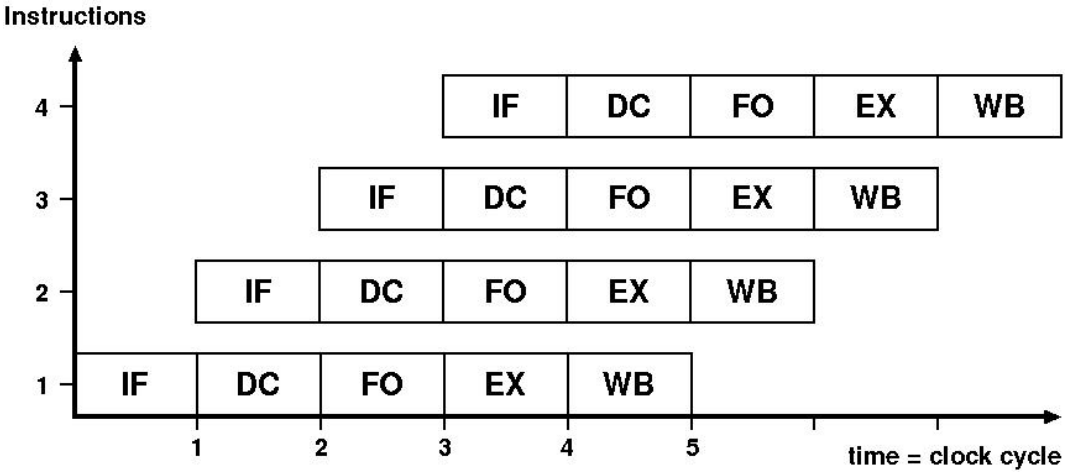


Figure 3.2: Command pipeline with 5 interlocking independent microinstructions

- **IF:** The instruction which the program counter is pointing to is fetched from memory or cache and loaded into the command buffer. The program counter is incremented to the next command (IF = *instruction fetch*)
- **DC:** Internal processor signals are produced from the microcommand (DC = *decode*)
- **FO:** The operands which are needed for the operation are fetched from cache or registers (FO = *fetch operands*)
- **EX:** Execute the necessary command on the operands (EX = *execute*)
- **WB:** The result of the operation is written back into a register or cache (WB = *write back*)

There is no reason why such steps should not be performed simultaneously. This way the computation can be accelerated by overlapping these phases. It takes 5 clock cycles in the example above until the pipeline is "filled." Ideally, beginning with the 6th clock tick, a result "drops" out of the instruction pipeline with each clock cycle. After a short while, the so-called *latency time*, a processor with such a pipeline is about 5 times faster than one without instruction pipelining [Pacheco]. The grade of parallelism of pipelining is determined by the number of different pipeline stages, and generally ranges between 2 and 14 [Rauber].

In [Quinn] a nice little example from daily life can be found to illustrate the problem of pipelining. Suppose automobiles get constructed on a four-stage assembly line where each of the stages requires one hour of assembly time. Starting out with an empty assembly line, it would take 4 hours for the first automobile to be assembled. At this point the assembly line would be entirely filled and the second car would then be completed one hour later. This way the assembly of the kth automobile would be finished at hour $3 + k$. Fig. 3.3 illustrates the principle of pipelining.

A practical example from daily programming could be the following short snippet of Fortran code:

```
float x[100], y[100], z[100];
for (i = 0; i < 100; i++)
    z[i] = x[i] + y[i];
```

These three lines of code do not look too exciting and it is not obvious what this all has to do with pipelining. This will become clearer when the operations which are executed invisibly to the

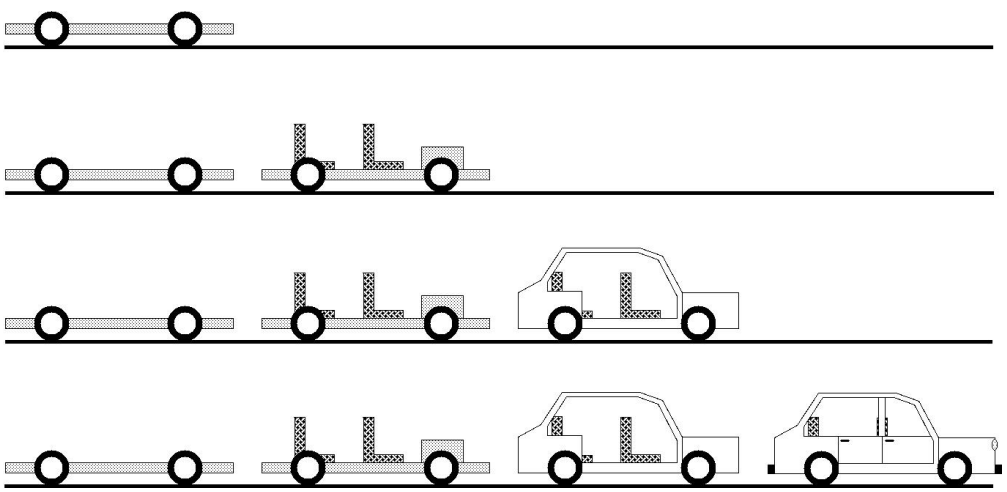


Figure 3.3: Assembly line with 4 assembly stages for automobiles

computer user are taken into account. Such code is transformed from a“readable form” for humans into a special machine code by tools called compilers. By such compilation these 3 exemplary lines of code are broken down into many smaller micro-code sequences. The better such a compiler works, the more optimally the computer hardware is used effectively later on. In the best case the machine instructions resulting from the small Fortran code example can finally be executed in the CPU similar to an assembly line, as in Fig. 3.3.

The problem is that such an instruction pipeline is only effective when the pipe is kept full; that is, only when there are operands available for operation on each segment clock cycle. Unfortunately, user computations rarely satisfy this requirement [Allen]. Examples, algorithms and analyses with regard to pipelining are givenm in [Wilkinson].

Superpipelining

A further improvement of pipelining techniques as introduced in subsection 3.2.2 is the so-called *superpipelining*. This requires the existence of at least 6 or more pipeline stages. The reason for this is found in the fact that modern processors operate internally with much shorter clock cycles than the remaining hardware outside the processor. To overcome this discrepancy between the chip-internal and chip-external processing speed, the pipeline is divided into even more independent stages. This necessarily requires a realization of fast cache hierarchies where the fastest cache, a so-called primary cache or L1 cache, is located directly within the processor. At decreasing speed, further caches are installed outside the processor. This way a very high frequency within the CPU is possible whereas the hardware outside the processor can operate at slower frequencies and thus, as a side effect, use much cheaper components [Ungerer01].

Unfortunately, it is not possible to constantly introduce more and more stages in such a command pipeline, although the grade of parallelism normally then increases. The restricting factor is given firstly by the fact that instructions can be broken down into independent arbitrary subcommands only to a certain degree; and secondly, data dependencies often impose limitations to the full exploitation of such an instruction pipeline [Rauber].

In common computer architecture families, both trends have been realized: Short pipelines with 4 to 6 stages can be found, for example, in the quite familiar PowerPC processors made by IBM

and Motorola. Long pipelines with 7 to 12 stages have been realized in Sun's SuperSPARC, UltraSPARC processors as well as in Intel's PentiumPro processors.

Superscalar Processors

A further improvement in comparison to pipelining or superpipelining is implemented in so-called superscalar processors. Agarwala and Cocke introduce the term *superscalar* in [Agarwala]:

> Superscalar machines are distinguished by their ability to dispatch multiple instructions each clock cycle from a conventional linear instruction stream.

In other words, a superscalar microprocessor now contains several independent functional units of the same type as (just to name a few):

- ALUs (3.1)
- floating point units (FPU), specialized hardware for very fast floating-point operations
- load and store units, specialized hardware for fast and efficient memory access,

All these different units are able to process independent instructions in parallel. Implementing such a superscalar technique results in a much shorter processing time, as on average, more than just one instruction can be dispatched to the different hardware units with each clock cycle. The only requirement is that something comparable to a "supervisor" be implemented within the microprocessor to take care of the handling and dispatching of instructions. This unit is called an *instruction dispatch unit*. Although these specialized units might be able to work in parallel, the program is still sequential. The scheduling is performed *dynamically* by the dispatching unit if there are no conflicts among the tasks to be scheduled (compare to statical scheduling in section 3.2.2).

There are a few drawbacks, called *pipeline hazards*, such as, for example, data dependencies. The problems resulting from such hazards in superscalar processors are very complex and a discussion of how to treat such conflicts is beyond the scope of this introduction. For further information, see [Ungerer02], [Johnson] and [Diefendorff].

VLIW Processors

VLIW denotes a typical computer architecture where a certain number of independent micro-instructions are tied together in one single machine instruction. Depending on how many functional units are present within a CPU or microcontroller, this *very long instruction word* will have a fixed length of 256, 512 or 1024 bits. With every clock cycle, instruction word after instruction word is then assigned to the different functional units like the ALU (see section 3.1), FPU (see section 3.2.2), etc., as depicted in Fig. 3.4.

At first this looks like a slightly complicated type of superscalar technique as described in subsection 3.2.2. However, the major advantages and disadvantages of VLIW can clearly be seen:

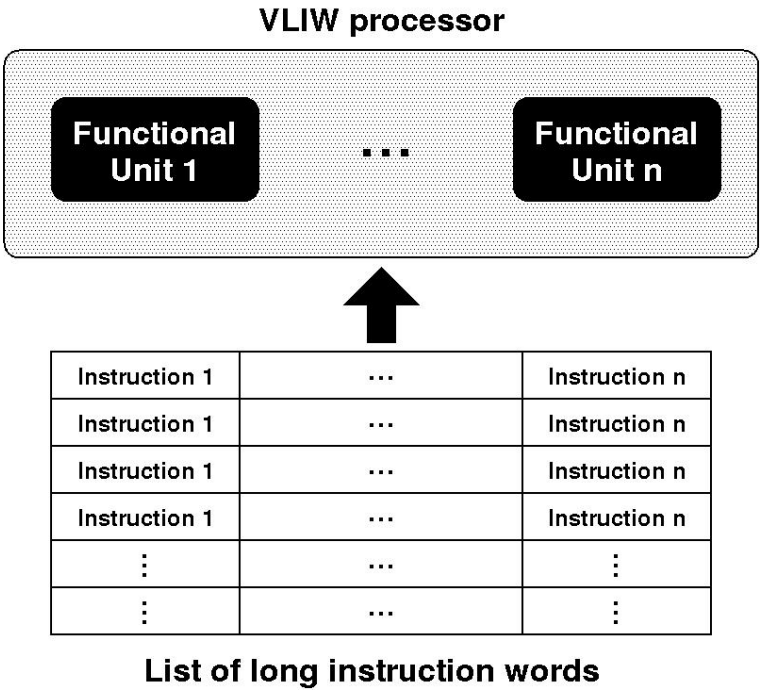


Figure 3.4: A stream of very long instruction words is assigned to the different functional units within the VLIW microprocessor

- No implementation of dispatching unit is necessary compared to the superscalar techniques (see subsection 3.2.2). Instead, a *statical scheduling* is automatically performed. Thus, the hardware complexity is less than that of superscalar architectures.
- As the assembly of the long machine instruction word is performed automatically by the compiler (a "translator program" which reads in a commandline from a programming language like Fortran or C and then translates this into a hardware-dependent machine instruction), there is practically no chance to react quickly and flexibly to unforeseen problems during assignment of the long instruction word or to hazards occuring during program execution.

Superscalar machines using the VLIW technique met with poor acceptance by the computer market; nevertheless, many of the basic concepts of the VLIW machines are found nowadays in our modern optimized compilers [Ungerer01].

Many more and new details on how to organize modern microprocessors can be found clearly structured in [Šilc]. Current information on the development of microprocessors is published either by [Wisconsin], [Berkeley] or [Ras].

3.2.3 Parallelism at Block Level

Parallelism at block level shows some similarity to pipelining and superscalar techniques (see subsection 3.2.2 for both). All these different concepts have one basic aspect in common: They try to minimize idle time of the processor. Only with a CPU or microprocessor that is kept busy most of the time can the best performace be achieved. While operating on the hardware-dependent microinstructional level with the concepts of pipelining or superscalar implementations and interlocking functional units *within* the microprocessor, one tries to minimize the idle time imposed

on the CPU from functional hardware units *outside* the microprocessor when using concepts of block-level parallelism.

A typical example is the problem of memory latency. When requesting data from the hard disk or an external device in general, the transport of the information to the microprocessor requires many clock cycles due to all the necessary administrative tasks that have to be performed. Furthermore, hard drives, tape drives, dvd drives, etc., are mechanically operating devices which are automatically slower than a microprocessor due to this fact.

Without any preventive measures, handling this deadtime from the point of view of the CPU, the performance of a computer is slowed down enormously as this task is repeated with every datum that is loaded. To circumvent such unnecessary delays, some improvements are implemented. The most common one is the usage of a *cache* and *register hierarchy*, where the fastest registers and so-called *caches* are installed close to the microprocessor, whereas the slowest memory and cache components are installed as far away from it as possible. Additionally, as soon as some delaying task within the execution of the program appears, the hardware will try to switch to something else which can be processed much faster in case there are no dependencies in between the single program steps. This requires some prediction or "good guessing" by the administrative hardware units regarding the different commandlines of the program that determine what should be processed when.

We can take, for instance, a loop in a C or Fortran program and some additional commands before and after this loop. As long as there are no dependencies of the data within the loop and outside the loop, the order in which the different lines are programmed does not matter at all. Thus, a good compiler (see also subsection 3.2.2) will reorganize the code in such a way that idle time is kept to a minimum. Although the program code is still sequential, a certain degree of parallelism is now achieved by reorganization.

The theory and implementation of these so-called *multithreaded processor techniques* or the *lightweighted process management* is quite complicated and is beyond the scope of this short introduction. For further information, see [Ungerer01], where one can find a good introduction and a long discussion on the different problems and implementations.

3.2.4 Parallelism at Task or Process Level

In many cases a program does not consist of just one long piece of code but a number of subprograms or procedures. When starting a program the necessary parts are then started and terminated automatically and transparently through the operating system in the background. All these procedures and program parts are called a *process*.

It could now be that one such process is "hanging" because some hardware device is not accesible or other problems have arisen. Any user would become quite angry if his or her program just "stopped" all of a sudden and seemingly did nothing. Therefore, the operating system tries to minimize such idle times by interlocking all the program-related processes in such a way that the hardware is used most efficiently, thus improving the program execution. The hard- and software units of the operating system which are involved still have to ensure the correct result upon terminatation of the program.

3.2.5 Parallelism at Job or Program Level

Most computer users nowadays are just "aware" of the one and only computer that they might have at home or at their office. Quite often, though, the machine that one sees is not really a self-contained and independently operating computer. In fact it is just a computer terminal which is connected to a main or central computer system. This constellation is often present in banks or travel agencies, to name a few.

Now, let us take such a computer system with various terminals as a starting point. All of its users expect it to react as fast as possible if they issue, for example, a command or start up a program. None of its users would accept an idle time of minutes if one user submitted a print job to the attached printer. And why should they? There is no need for the majority of users to wait for a task to be finished if the invisible operating system in the background manages to take care of all user requests in an optimal way.

To allow maximum throughput of jobs and thus a high performance of the system, the operating system tries to gear all currently running jobs, processes and programs of the users and of its own administrative programs in such a way that practically no time delay occurs.

This concept of the interlocking of processes appears at many different levels of parallelism, as shown above. It is the major principle behind speeding up existing and concurrently running tasks. Even if tasks and processes are spread on several independently operating computers, this simple aspect will show up again (see also subsection 3.4).

3.3 Different Types of Parallelism

A basic question on the way to parallelization is quite simple but important:

When can *what* be executed by *which*?

This leads us to the problem of the different types of parallelism.

3.3.1 Spatial Parallelism

To describe the aspect of *spatial parallelism*, a short example will help here to clarify what is meant by this term. When implementing the superscalar technique (see subsection 3.2.2), several operations are executed in parallel on different functional units. This means our tasks are spread "simultaneously" onto various independently operating hardware devices within the microprocessor. This way superscalar microprocessors implement the aspect of *spacial parallelism*.

3.3.2 Temporal Parallelism

To elucidate the aspect of *temporal parallelism*, the superpipelining technique as described in subsection 3.2.2 will help here: The idea of superpipelining is to interlock several operations on one functional unit. This was achieved by introducing pipeline stages and "chopping" commands into subcommands, similar to the example of the automobile assembly depicted in Fig. 3.3. Now more tasks can be performed simultaneously or in parallel although we haven't doubled our functional unit. Therefore superpipelining emphasizes the aspect of *temporal parallelism*.

3.3.3 Data Parallelism

Quite often in programming or even in daily life we experience situations where we have data parallelism. A typical example of data parallelism would be "increase the salary of all employees with 5 years of service" [Foster01]. Thus, in more abstract terms, *data parallelism* means applying the same operations to different elements of data sets.

Here is another short example using pseudo code:

```
For i = 0 to 99 do
    a[i] = b[i] + c[i]
End
```

The same operation, which is the addition of two elements, is performed on the first 100 elements of the arrays *b* and *c*. The results are then stored in the elements of array *a*. All 100 operations of this loop can be executed simultaneously, as no element set depends on another one.

3.3.4 Functional Parallelism

In comparison to the above-mentioned types of parallelism, *functional parallelism* is the easiest one to understand. It can simply be described as: "applying different operations from independent tasks to different data elements" [Quinn].

3.4 Parallel Computer Architectures

This section briefly discusses the different architectures of parallel computing systems. [Pacheco] expresses the problem of classifying parallel computer architectures as follows: "There are as many varieties of parallel computing hardware as there are stars in the sky ..." The good news is, there's not *that* many; unfortunately the hardware industry was sedulous enough to come up with so many differently implemented concepts that this section can only describe the major architectures found on today's market.

3.4.1 *Flynn's* Taxonomy

The first and still most extensive taxonomy of parallel computers is popularly known as *Flynn's Taxonomy*. The basic principles of his classification are based on the number of instruction streams and the number of data streams. This means that at a given time, a computer is processing either one instruction or more than one instruction and, equally, that at a given time, a computer is working either on one data value or more than one data value.

Combining these statements, the following table of 4 major classes of computer architectures results:
The classical *von Neumann* machine (see subsection 3.1) has a single instruction stream and a single data stream, and hence is identified as a SISD computing system according to Table 3.1.

SISD	**S**ingle **I**nstruction	–	**S**ingle **D**ata
SIMD	**S**ingle **I**nstruction	–	**M**ultiple **D**ata
MISD	**M**ultiple **I**nstruction	–	**S**ingle **D**ata
MIMD	**M**ultiple **I**nstruction	–	**M**ultiple **D**ata

Table 3.1: Taxonomy of Parallel Computing Architectures after *Flynn*

The opposite extreme would then be a collection of autonomous processors which operate on their own data streams, symbolizing a computing architecture of the MIMD class. This is the most general architecture. The four different architecture classes derived from *Flynn's Taxonomy* are discussed in more detail in the following sections. Fig. 3.5 nicely shows a summary of the different classes after *Flynn* and which computer architectures can be found in which category.

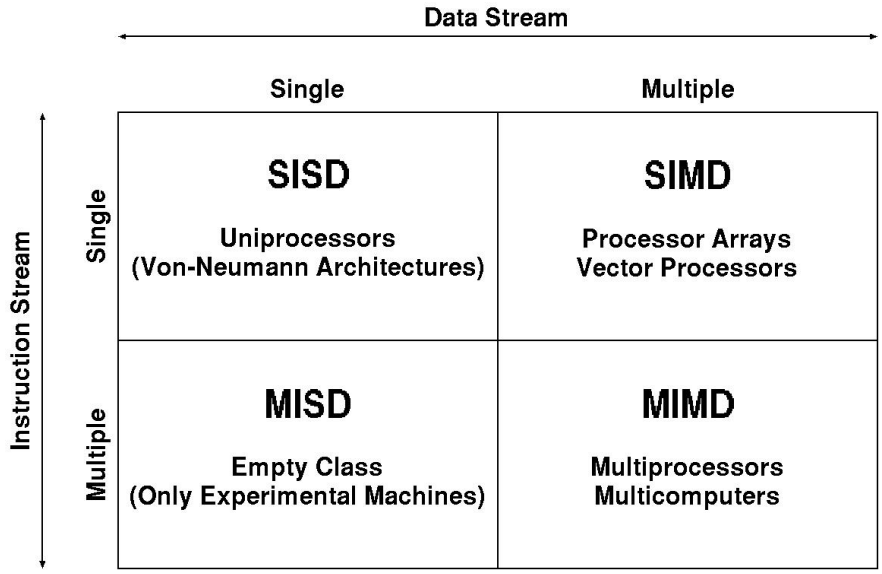


Figure 3.5: Flynn's taxonomy of computer architectures

3.5 Single-Processor Systems

According to *Flynn's Taxonomy*, the two classes which can be found in this category are the SISD class and the SIMD class. They have in common that every processor works with the *same* instruction, but the data they are processing might be different.

3.5.1 SISD Computer Architectures

The computer systems found in this class of *Flynn's taxonomy* can quickly be characterized by the term *uniprocessor architectures* or *scalar processor architecture*: SISD – single instruction stream with single data stream. This is generally the typical *von Neumann* architecture which was already introduced in subsection 3.1. This machine symbolizes the traditional sequential computer, consisting of one processor, a memory and one communication channel as is depicted in Fig. 3.6. Obviously this computer architecture will always be restricted by that part of the three elements

with the least capacity. With our present technology that provides almost unlimited memory and nowadays extremely fast central processing units, the typical bottleneck is generally the communication channel.

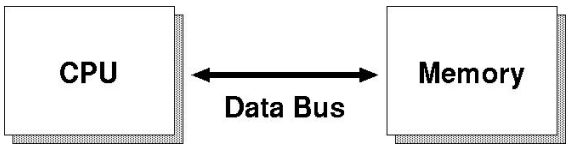


Figure 3.6: The structure of a traditional sequential computer architecture

According to [Quinn], a modern uniprocessor may still exhibit some concurrency of execution, even though it has only a single CPU executing a single instruction stream. In his opinion, superscalar architectures (see subsection 3.2.2) support the dynamic identification and selection of multiple independent operations that may be executed simultaneously. Instruction prefetching and pipelined execution of instructions (see subsection 3.2.2) are other examples of concurrency typically found in modern SISD computers, though according to *Flynn* these are examples of concurrency of processing, rather than concurrency of execution [Flynn2].

As a suitable example from daily life, one can consider a one-person office. The person working here has to take care of everything that might happen during a long working day. He or she is processing his or her necessary office work step by step.

3.5.2 SIMD Computer Architectures

In comparison to the SISD architectures, we now find multiple data streams but still a single instruction stream. The typical architectures of SIMD are the so-called *processor arrays* and the *vector processors*, which will be explained a little bit more in detail below.

These architectures have in common that they have either just one program memory (this is where the instructions about what to do with which data to be computed are stored) or they will be fed the commands to process by a single root processor. This means that every single processor of that machine uses the *same* instruction, but the data they are processing might be different. The data to work with can either be read in from a shared memory or each processor can have its own memory and thus, its own data stream. The latter is called distributed memory architecture. In Fig. 3.7 the shared memory architecture is shown, whereas Fig. 3.8 depicts the distributed memory realization of such a SIMD machine, but in both figures a central root processor distributes the commands that are to be processed.

The advantage of the architectures in this class is that they overcome the above-mentioned bottleneck with the SIMD systems by implementing multiple processors and memory module(s), but are still working on a single instruction set. Using just one instruction set simplifies the programming task as the same instruction set is performed on all processors at any given time but the number of data sets are program independent. However, host communication may prove to be troublesome for this architecture since each processor must share the same communication path through the pipeline, according to [Topping].

In SISD we took a one-person office as a typical example. Now with SIMD, where we still have the same, identical instruction that is executed by all processing units but different data streams, we

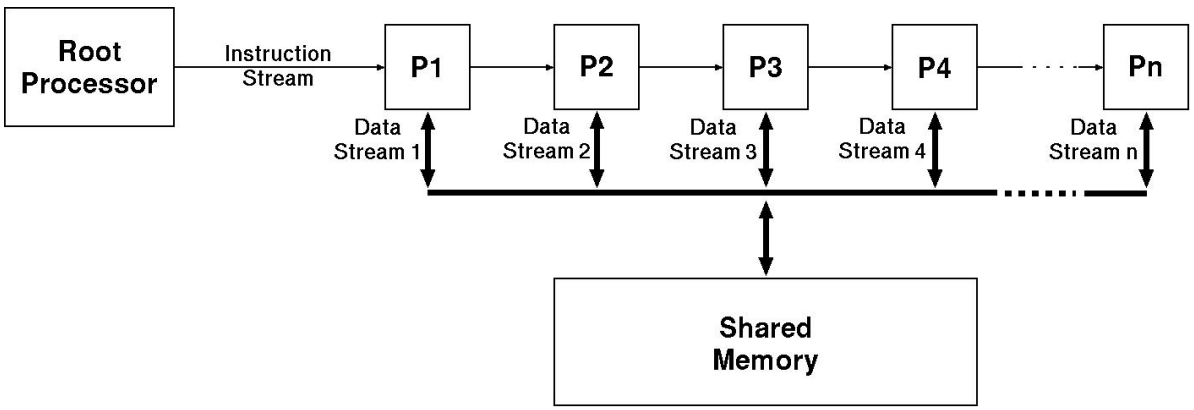


Figure 3.7: Single Instruction Multiple Data (SIMD) architecture with shared memory

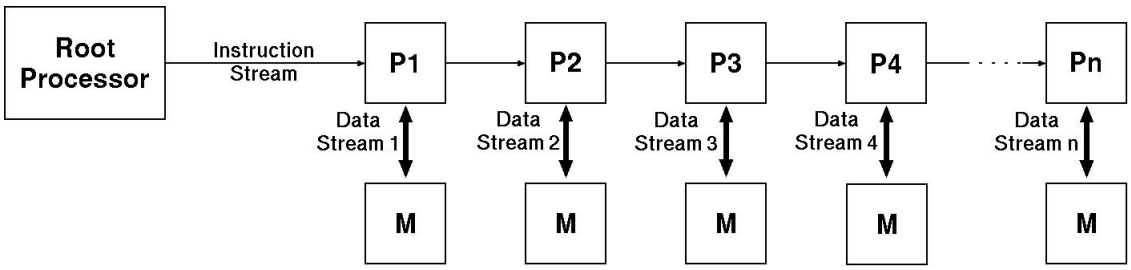


Figure 3.8: Single Instruction Multiple Data (SIMD) architecture with distributed memory

could imagine a less pleasant working environment, with many office clerks at their desks in a big open-plan office and a supervisor with a megaphone shouting instructions which everybody has to obey. The employees are not working at their own pace but are synchronized by this megaphone coordinator.

Vector Processors

Vector computers are specially designed computer architectures which are conceived for the computation of specific mathematical problems. Such machines are used in the field of high-performance computing (HPC) where one encounters a vast amount of congeneric data which has to be processed using, for example, huge matrix-vector multiplications or enormous differential-equation systems. Typical fields of application are meteorology and geology, where complex models which are to be simulated are often created.

Another major area of application of such vector processors is graphical simulation as found in 3D computer games or computer graphics. Here, the problem of numerous matrix operations on big input-data sets has to be confronted. Therefore, the majority of common graphic processors show marked similarities to the original typical vector machines.

The term *vector* as it is used with vector processors or vector computers does not directly derive from the mathematical term of a *vector* as found in physics or mathematics. The fact that the instructions work on an ordered number of congeneric operands which have to be treated in the same way finally led to the usage of the term *vector*. Thus, operating on single operands is called *scalar processing* in contrast to vector processing. Sometimes the term *array processor* instead of

vector processor is used in the literature, as one is most likely to operate on large arrays of input data as described a little later. (This should not be confused with the *processor arrays* which will be discussed in section 3.5.2).

One typical characteristic of vector processors is the implementation of (instruction) pipelining (see also subsection 3.2.2). The instructions pass through several sub-units in turn, where the first one, for example, reads the address and decodes it, the second one gets the value, the next does the math and so forth. In 3.2.2 above, this concept was compared to an assembly line as shown in Fig. 3.3.

Additionally, vector processors take this pipelining concept even one step further. Instead of simply pipelining just the instructions, they now also pipeline the data itself. For this purpose special vector registers of generally 128 to 256 floating-point numbers are addressed, on which one floating-point instruction is then applied pairwise and the resulting vector itself is stored in another vector register. This way every vector processor now has to read and decode one single instruction from the memory and "knows" that the next address will be one larger than the last one. This allows for significant savings in decoding time instead of constantly decoding and fetching data from the slower memory.

The problem with vector processors is that they have a number of severe drawbacks. If a program consists of many irregular structures or tends to branch a great deal, the principles of pipelining and vectorization no longer work. After [Pacheco], the key to performance with vector computer systems is filling the pipeline and keeping it full. If operands aren't laid out properly in memory, this is completely impossible. The more conditional branches can be found in a code, the less use of vector instructions can be made. One major drawback though is the lacking possibility of scaling. This means that it is not clear how to modify existing systems in such a way that they could handle larger problems. Even if several more pipelines are added and one manages to keep them full, the upper limit of speed for vector processors will still be some small multiple of the speed of the CPU.

It should be noted that depending on the literature in question, vector processors are sometimes also counted towards SISD as an improvement and extension of the classical *von Neumann* architecture, where vector processors implement pipelining and vector instructions to the basic instruction set. Some authors even regard vector machines as MISD machines, whereas others state that vector computers are not even really parallel machines at all.

Over the past few years vector processing systems have been increasingly replaced by massive parallel computer clusters. These are built from thousands of standard processors. Using common hardware components, costs can be reduced enormously, since, in addition, such standard CPUs have become quite powerful and effective through constant technological improvement and development. A quite well-known vector computer of the past was, for example, the Cray-2. Also the powerful supercomputers at the High Performance Computing Center in Stuttgart (HLRS), one of the 5 German federal high-performance computing centers, has installed such typical vector processing systems as NEC SX4, NEC SX6 and currently NEC SX8.

Processor Arrays

The pure SIMD system (as opposed to a vector processor) has one instruction processing unit, which is sometimes called a *controller*, as it is devoted exclusively to control, and a large collection of data processing units, each with its own (small amount of) memory.

During each instruction cycle the control unit is responsible for fetching and interpreting instructions. In case it encounters arithmetic or other data-processing instructions, the control processor broadcasts them to all subordinate processing units, which then all perform the same operation or remain idle.

In [Pacheco] a simple but nice example can be found: Suppose we have three arrays *x, y* and *z*. These are distributed in such a way that every processor contains one element of each array in its memory. The following sequence of serial instructions should now be executed:

```
for (i = 0; i < 1000; i++)
{
    if (y[i] != 0.0)
       z[i] = x[i]/y[i];
    else
       z[i] = x[i];
}
```

Each of the processing units would execute something like the following sequence of operations:

Time Step 1: Test `local_y != 0.0`
Time Step 2:
–> **a)** If `local_y` is not zero, `z[i] = x[i]/y[i]`
–> **b)** If `local_y` is zero, do nothing
Time Step 3:
–> **a)** If `local_y` is not zero, do nothing
–> **b)** If `local_y` is zero, `z[i] = x[i]`

This automatically implies completely synchronous execution of statements. At any given instant of time, every subordinate processor is either *active* and does exactly the same things as all other active processors are doing or it is inactive, which means it is *idle*.

Surely, one of the advantages of this style of parallel machine organization lies in enormous savings with regard to hardware logic. Generally, about 20 % to 50 % of the logic on a typical processor chip is devoted to control, like fetching, decoding and scheduling instructions. The remainder is used for registers and caches (so-called on-chip storage) as well as for the logic required to implement the data processing, i.e., the adders, multipliers, etc.

As we have seen, only the controller fetches and processes the instructions in an SIMD machine. This way more logic can be dedicated to arithmetic circuits and registers. As a fitting example, the MasPar MP-1 can be taken, where 32 processing units fit on just one single chip, and a processor system of 1024 processors is built from only 32 chips all together. They fit on a single board! (The control unit itself occupies a separate board.)

One of the major drawbacks of such processor arrays can be seen directly from the given example above: If our program consists of many conditional branches or long segments of code which depend on conditionals, it is more than likely that many of the processing units will remain idle for a long period of time. In contrast, if the underlying problem shows regular structures, SIMD machines tend to be relatively easy to program. Especially as communication is quite expensive, as, for instance, in distributed memory MIMD systems (see subsections 3.6.4 and 3.6.5), SIMD machines can be a good alternatives. Furthermore, they do scale well to increased problem sizes.

The CM-1 and CM-2 Connection Machines are well-known SIMD machines produced by Thinking Machines. The CM-2 had up to 65,536 1-bit processors and up to 8 billion bytes (about 8 GB) of memory. The MasPar MP-2 has up to 16,384 32-bit ALUs and up to 4 billion bytes (about 4 GB) of memory.

3.6 Multiprocessor Systems (MPS)

The key difference between MIMD systems and SIMD systems is that with MIMD systems, the processors are autonomous: Each processor is a full-fledged CPU with both a control unit and an ALU [Pacheco]. Unlike SIMD machines, MIMD systems are asynchronous, as each processor is capable of executing its own set of instructions at its own pace. Often, such systems do not have a global clock for synchronization purposes. Unless there is a special programming synchronizing the different processing units of such MIMD systems, it is more than likely that there is no correspondence between what is being done on different processors, even if they are all executing the same code.

MIMD systems are commonly subdivided into two major types of architectures: *shared memory multiprocessor systems* and *distributed memory multiprocessor systems*. In the literature different terms for the two systems are sometimes found, like *multicomputers, parallel processors, supercomputers, multiprocessors*, etc. Unfortunately none of the different names has gained wide and common acceptance. This might lead to some confusion when names are used interchangeably. Often different basic concepts of the multiprocessor systems are combined, as in *distributed shared memory multiprocessor systems*.

The following sections will try to bring some order into the wide field of multiprocessor systems. There are some architectures which implement not only one but several techniques. Here only the most common architectures will be briefly described and summarized. Additionally, the underlying type of communication network or bus system implemented in such multiprocessor systems is important as well. Dwelling on networking structures and its pros and cons would be beyond the scope of this introductory chapter; whole books have been published on the topic.

3.6.1 MISD Computer Architectures

The class of MISD computer architectures does not exist [Ungerer01]. It is just mentioned in the taxonomy for reasons of completeness. When considering a computer which is able to perform multiple instructions simultaneously on just one set of data, it immediately becomes evident why no existing computer can be found in this class. Variously in the literature a few experimental machines are mentioned in this category, but none of them has been commercially successful or has shown any impact on computational science.

One type of system that sometimes is classified as a MISD computer and which indeed was built, is a *systolic array*. The name comes from the word *systole*, which refers to a contraction of the heart [Quinn]. This is a network of small computing elements connected in a regular grid which "pump" data. All the processing units are controlled by a global clock. On each cycle, every element will read a piece of data from one of its neighbors, perform a simple operation (e.g., add the incoming element to a stored value), and prepare a value to be written to a neighbor on the next step [Drakos].

3.6.2 MIMD Computer Architectures

The category of MIMD machines is the most diverse of the four classifications in Flynn's taxonomy. It includes, for instance, machines with processors and memory units specifically designed to be components of a parallel architecture, large-scale parallel machines built from "off-the-shelf" microprocessors, small-scale multiprocessors made by connecting a couple of vector processors together and a wide variety of other designs. With the continued improvement in network communication and the development of software packages that allow programs running on one machine to communicate with programs on other machines, users are even starting to use local networks of workstations as MIMD systems [Drakos]. In Fig. 3.9 and Fig. 3.10 two exemplary archictures of a typical MIMD computer are shown: Fig. 3.9 depicts the structure of a shared memory machine, whereas Fig. 3.10 is a typical example of a distributed memory architecture in the MIMD class.

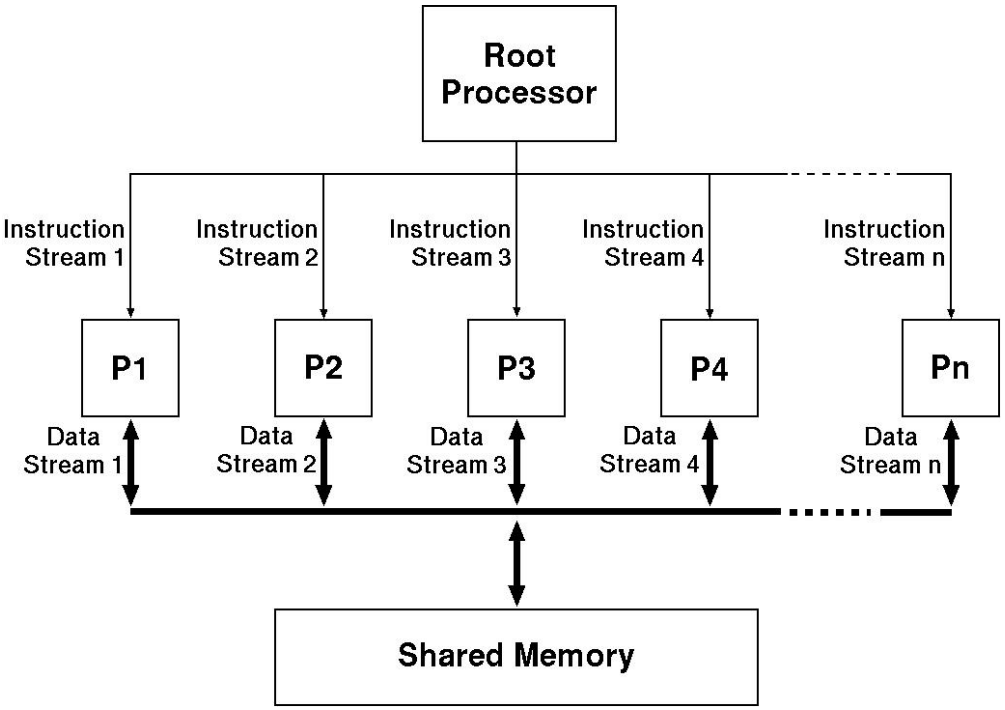


Figure 3.9: Multiple Instruction Multiple Data (MIMD) architecture with shared memory

Computer systems with two or more independent processors have already been available commercially since the 1970s. Multiprocessors of this era were intended to work on different jobs (*job-level parallelism*, see subsection 3.2.5), i.e., each would run a separate program.

Parallel processing, in the sense of using more than one processor in the execution of a *single program*, has also been an active area in corporate and academic research labs since the early 1970s [Drakos]. Commercial parallel processors started to become widely used in the mid 1980s and by the early 1990s, these systems began to approach top-of-the-line vector processors in computing power. The actual trend for high-performance computing (HPC) is clearly with parallel processing. The surely best-known example is the *Earth Simulator* of NEC at the Earth Simulator Center in Yokohama, Japan, which led the list of the 500 fastest computers in the world [Top500] for many months. In subsection 3.6.4, where distributed shared memory systems are described, the *Earth*

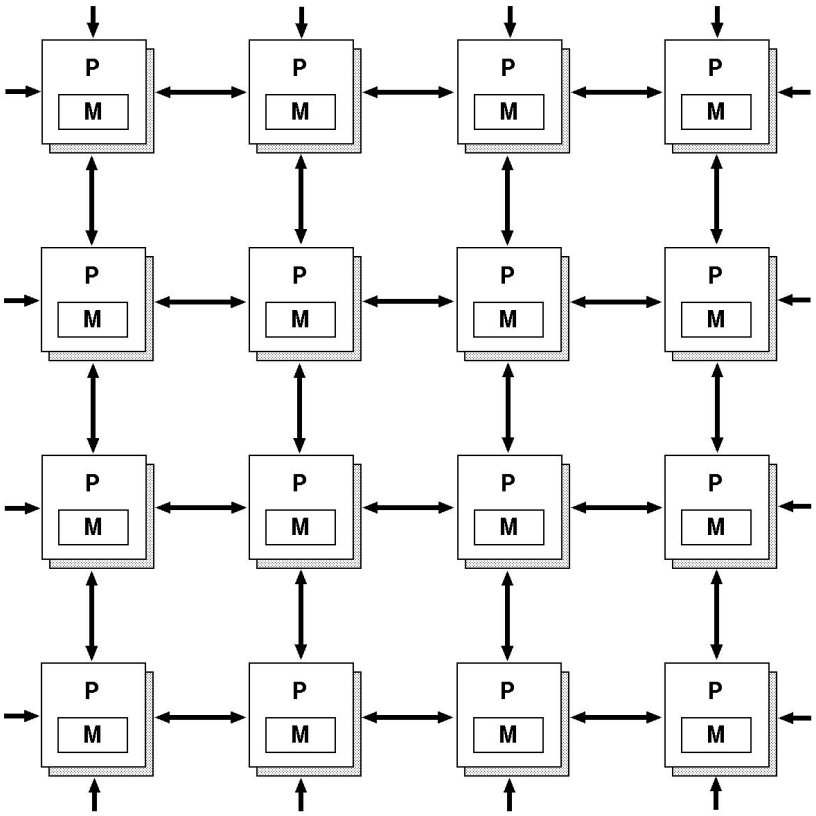


Figure 3.10: Multiple Instruction Multiple Data (MIMD) architecture with distributed memory

Simulator is introduced in a little bit more detail as one example of such powerful supercomputers.

Homogeneous and Inhomogeneous Systems

Normally all processors or processing nodes within a multiprocessor system (MPS) are of the same type and construction with regard to hardware, as if they were twins. Such systems are called *homogeneous MPS*. Such systems are the most likely ones to be used. Nevertheless there are also systems where the processing units or nodes differ in hardware within the same multiprocessor environment. That is not the standard, but they do exist. Such MPS are called *inhomogeneous MPS*.

Symmetrical and Asymmetrical Systems

The typical characteristic of a *symmetrical multiprocessor* (SMP) system is that all processors are essentially identical and perform the same functions and tasks. Often in such a symmetrical multiprocessing architecture the CPUs or processing nodes reside in one cabinet or on one mainboard. Boards can be purchased on which 8 identical processors are installed; such would be a typical symmetrical SMP system. Also vector computers (see subsection 3.5.2) could be designed as a SMP system, but they do not necessarily need to be a SMP system.

Often SMP systems share the same memory and provide good scalability in hardware. When business increases, additional CPUs can then be added without problems to absorb, for example, the increased volume in transactions. Fig. 3.11 shows what such a symmetrical multiprocessor environment could look like.

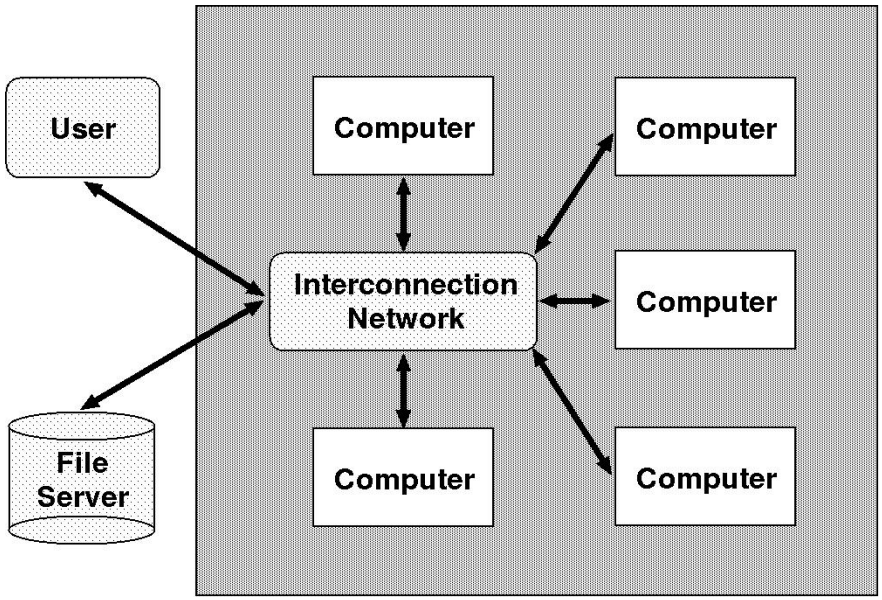


Figure 3.11: An example of a simple symmetrical multiprocessor architecture

There is no limit to SMP systems. They can range from just two processors up to as many as 32 or even more computing nodes. Unfortunately, some systems have the disadvantage that they are down completely if just one of the operating nodes fails. If the operating system is set up properly, SMP systems can easily move tasks between the computing nodes to balance the workload efficiently.

A big disadvantage is the fact that the CPU is much faster than the memory in general. Already with a single-processor machine (see also subsection 3.2.2) the CPU spends a very considerable amount of time waiting for data to arrive from memory. In SMP systems with a shared memory, only one processor out of many can access memory at a time. This can result in the *starvation* of other processors.

In contrast to the SMP system, in an *asymmetrical multiprocessor* (AMP) system, the processors are assigned different roles. For instance, one processor may handle I/O, another executes a user program, and so forth. This approach also involves some typical advantages and disadvantages. Sometimes I/O operations are extremely slow. Instead of slowing down the whole system, one or a couple of dedicated processors would take care of such tasks while others could just continue with normal and much faster computations. The problem of memory incoherence can be avoided or at least reduced through the application of sophisticated cache and memory models. One big disadvantage is that a single point of failure is easily created on AMP systems when, for instance, a specific task is assigned to just one single processor. If this one fails, the whole system is down. In an SMP system, when more than one node is working on the same task, the remaining ones would then automatically take over. Fig. 3.12 shows an example of a very simple asymmetrical multiprocessor.

For example, the SMP system could be compared with a group of allrounders. Each of the group members can substitute for another and take over whatever work has to be done. This requires

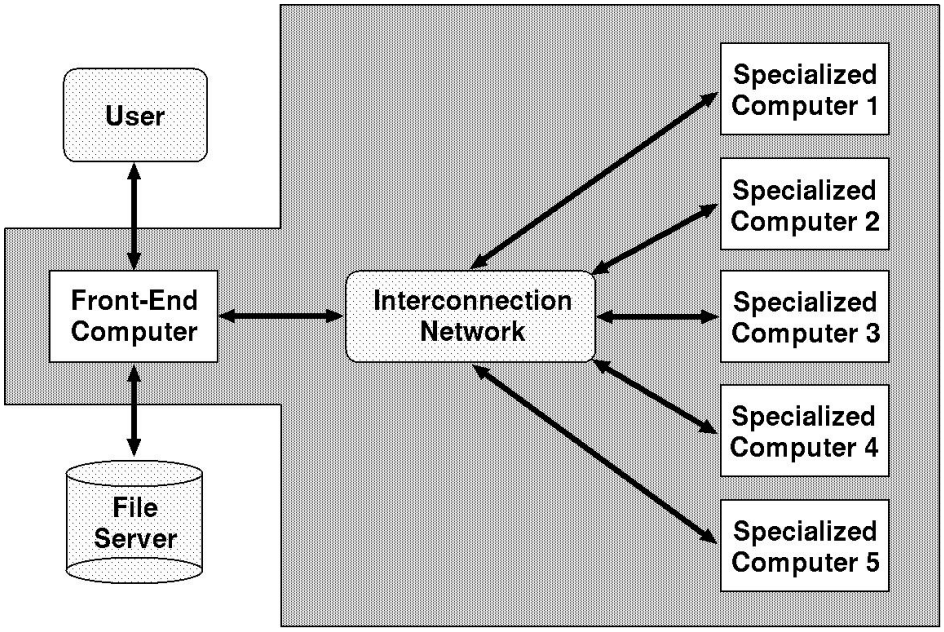


Figure 3.12: An example of a simple asymmetrical multiprocessor architecture

cooperation and coordination of course, because the allrounders have to talk to each other, slowing down the process as a whole. But with good organization, they will become a successful team.

An AMP system, in contrast, would be a group of specialists, where everyone would have his or her specific task, i.e., one would just take care of accounting, another only of customer service and customer relationships, etc. They all know what has to be done and whom to contact during their work. But imagine if the accounting specialist or the person in the central reception handling customer requests were sick and stayed home for a few days! The advantage of this system is that the time loss due to organizational matters can be reduced to a minimum. Unfortunately if there is no work for one of the specialists, he or she can't just take over some other tasks, as the allrounders can. The poor specialist might be bored to death and manpower is just wasted.

3.6.3 Shared Memory (SM) Multiprocessor Systems (MPS)

When talking about the term *shared memory*, one has to be more than careful! This expression is used not only with regard to hardware matters but also to software related to interprocess communication. Here, in this context, we are talking about the technique of *shared memory* with respect to hardware architectures.

Shared memory refers to a generally very large block of a so-called random access memory (RAM) that can be accessed by several different processing nodes in a multiprocessor environment. (The RAM could be compared to the short-term memory of a computer and should not be confused with a harddisk as a long-term memory!) Fig. 3.13 shows a generic shared memory multiprocessing system. Depending on how the network is realized in hardware, so-called *bus-based* or *switch-based shared memory systems* have to be distinguished.

There are two major problems commonly encountered with any shared memory system. The first one depends on how the internal structure of the multiprocessing system is implemented in hardware. The transport of any physical signal on a wire or connection from a source to a sink takes time. Depending on how the different nodes of the MPS are connected to each other, effects

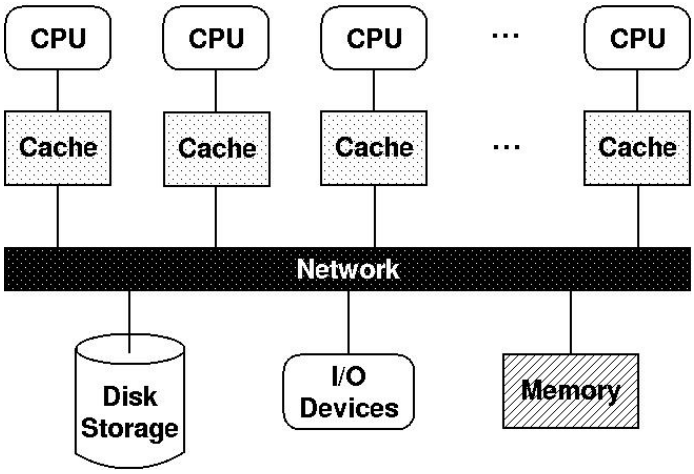


Figure 3.13: Architecture of a generic shared memory multiprocessor system

of signal propagation delay must be considered. For short distances this time is not important but the bigger and longer the network connection is in the end, the more such delays have be taken into account.

As a short, slightly exaggerated example, one could imagine a network with 3 processing nodes which are all attached to a networking bus system by a couple of hundreds of meters of cable. Node A is closest to the shared memory and computing unit C is the furthest one away. All 3 nodes are jointly processing a program where they constantly need to access their mutual data. If node C now requests some data from the shared memory, the signals representing this information travel quite some time until C can start working with the received data. In the meantime the closer nodes A and B are also computing their results and store them in the shared memory. If C now wants to store its results which A and B need for their further computations, the data again travels this long distance back until it can be stored in the shared memory. In the worst case, A's and B's information stored in the shared memory could have been changed before C could even read or store its data from there!

This little example shows quite clearly that the underlying hardware implementation of such a shared memory system is important and that especially physical effects like these propagation delays do play a role that has to be considered. (Those who would like to have more information on this topic can search in the literature for the terms *NUMA* and *UMA systems*.)

The second problem which is also commonly encountered with all shared memory implementations is that of *cache coherence* or *cache consistency*. A typical issue of fast and powerful multiprocessing systems is the fact that caching is very important with regard to processing speed. (In subsections 3.2.2 and 3.2.3 the principle of caches and cache hierarchies has already been introduced and described.) Therefore the different nodes of the *shared memory MPS* most likely will cache the information they have received from the memory. But if a processor node now accesses a shared variable in its cache, how will it know whether the stored value is still the correct, current one and that it hasn't been updated in the meantime in the shared memory?

The same problem arises when a processing node, let us call it A, stores a newly computed result of a shared variable in its cache before this variable is written and stored in the shared memory. How will another node B know about this change in value at node A when node B is going to access exactly this variable in shared memory? It doesn't know about the altered variable in the cache of node A.

There are a number of cache consistency protocols and they vary considerably in complexity depending on the size of the multiprocessor system and its hardware implementation. One of the best known is the *snoopy protocol* where the cache controllers "snoop" on the bus if they detect a change of shared variables. By monitoring the bus system they can mark outdated information as invalid. Another quite well-known cache coherence protocol is the *MESI protocol*. A clear description of the very complex protocol procedures can be found in [Giloi] and [Maertin].

Coherence protocols can, in case they work well, provide extremely high-performance access to shared information between multiple processing units of a MPS. On the other hand, they can sometimes become overloaded and all of a sudden bottleneck performance.

Depending on the underlying hardware, it may be necessary to provide some means of synchronization, typically some kind of *barrier synchronization*. This guarantees that no process will proceed beyond a designated point in the program until every process has reached the *barrier*. Also important when programming such platforms is to take care of *mutual exclusions*, using special locking mechanisms or some types of *semaphores*. This ensures that only exactly one process can be engaged in a certain activity at any time [Wilson].

Generally, shared memory architectures tend to scale well up to a certain number of processors until their internal communication structure (normally the main bus) becomes the main bottleneck.

In the literature the term *multiprocessor* is sometimes found as a synonym for a *shared memory environment*, since many processing nodes access just one main common memory. Generally, it is better to use the correct expression, *shared memory multiprocessing environment*, to avoid any confusion. The alternatives to such systems are either *distributed memory* or *distributed shared memory systems*. They work with a slightly different set of issues.

3.6.4 Distributed Memory (DM) Architectures

In a *distributed memory (DM) system*, as opposed to a SM multiprocessing system (see subsection 3.6.3), the memory is associated with individual processors. Here, a processor is only able to address its own memory, reflecting the fact that the building blocks in the multiprocessor environment are themselves small complete computer systems with processor and memory. A typical *distributed memory system* is a computer cluster (see also subsection 3.6.7).

This organization results in several benefits: Firstly, as there is no bus or switch connection, each processor can utilize the full bandwidth to its own local memory without interference from other processors. Local memory accesses are much faster than nonlocal memory accesses. Secondly, the size of the overall system is only constrained by the network connecting the different computing nodes in the multiprocessing environment because there is no inherent limit to the number of nodes through the lack of a common bus. The biggest advantage of this architecture, though, is the fact that there are no cache coherence problems. Each processing node is in charge of its very own data, and does not have to worry about putting copies of it in its own local cache and having another processing unit reference the original.

There is a major drawback to this *distributed memory design*, though: The interprocessor communication is now more difficult in comparison to SM systems. If a node of the DM system requires information from another processing node's memory, it must now exchange messages with the other computing unit. The result is a potential source for overhead as it takes time to construct and

send a message from one processing node to another, and the receiving processor must be interrupted in order to deal with the incoming messages from other operating nodes. Fig. 3.14 shows a generic distributed multiprocessing system.

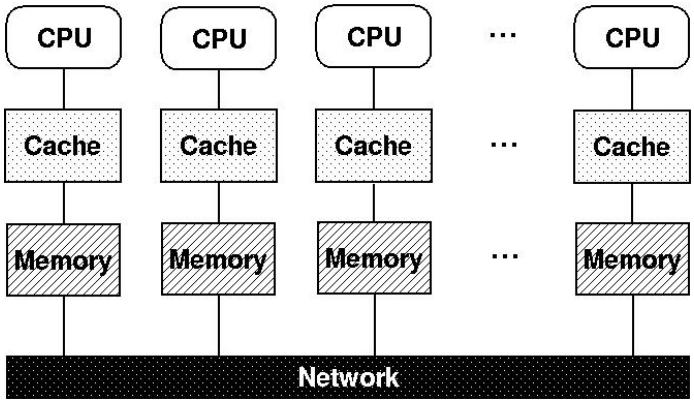


Figure 3.14: Architecture of a generic distributed memory multiprocessor system

In contrast to SM systems, the concepts of semaphores, monitors, and other concurrent programming techniques are not directly applicable to *distributed memory machines*. In such DM environments, programming is a matter of how to organize a program as a set of independent tasks that communicate with each other via messages. Programmers must be aware of where the information is stored. This introduces the *concept of locality* in parallel algorithm design. Generally, an algorithm that allows data to be partitioned into discrete units and then runs with minimal communication between the processing units involved will be more efficient than an algorithm which requires random access to global data structures. In DM environments the data normally has to be reassembled.

When realizing a distributed memory architecture, how the different processing nodes of the system are connected to each other is quite important. In the ideal interconnection network every node can directly communicate with every other computing unit. Such a fully connected network communication would involve no delay and can take place simultaneously among different processing nodes. Unfortunately, the cost of such a network makes it impractical to construct a machine like this with more than a few nodes [Pacheco]. There are endless possiblities of how to realize such interprocess communication depending on cost, speed and delay, etc. (see also subsection 3.7). The typical *dynamic interprocess network* today is built by using a crossbar switch. A common example of a *static interconnection network* would be the total contrary to a crossbar when using a linear array of processing nodes or a ring topology.

DM multiprocessing systems are also called *multicomputers* in some of the literature, but the correct complete term, *distributed memory architecture*, should be used to avoid confusion. In [Pacheco] a very good introduction to such systems can be found in addition to a full discussion of interconnection networks as well as of routing algorithms and programming of such different environments. [Quinn] is also worth a look in case one is interested in network topologies, interprocess communication and multiprocessor architectures.

3.6.5 Distributed Shared Memory (DSM) Architectures

A *distributed shared memory (DSM)* architecture is a combination of the two concepts which were shortly introduced in subsections 3.6.3 and 3.6.4. In such a *DSM environment* a virtual

shared memory system is based as an abstraction upon a physically distributed memory. The realization of the DSM system can either be done in software as a shared memory programming model and/or in hardware using cache consistency protocols to support shared memory across physically distributed main memories.

There are many different DSM systems available right now which are all based on the aforementioned technique. A good starting point for readers with an interest in this topic can be found on the Internet at [Huseynov].

3.6.6 Hybrid Architectures

When analyzing *hybrid systems* the concepts of shared memory systems (see subsection 3.6.3) and distributed memory systems (see subsection 3.6.4) is present again. In such architectures generally a hardware hierarchy is to be found. A small number of computing nodes, like, for example, 8, build a processing node. Within this processing node these 8 subsystems realize a shared memory system.

The hybrid multiprocessor system is finally assembled through a certain number of processing nodes. These different nodes all together form a distributed memory system. A nice example of such a hybrid multiprocessing architecture, here a NEC SX-8, can be seen in Fig. 3.15.

In Fig. 3.15 the blue square depicts the smallest unit in the whole system, which is a single CPU. Eight of these CPUs form the next bigger unit in the hiearchy, i.e., the processing node. The CPUs realize a shared memory system within the node they belong to. 1 or up to 512 of these processing nodes can be combined at the SX-8, which gives 4096 CPUs in total. The nodes of this multiprocessing environment of the SX-8 work as a distributed memory system, which means that each of the nodes behaves as if it were one single, independent computer. It is not externally obvious that a shared memory system is hidden inside. Fig. 3.16 shows an example of a medium-sized SX-8 model.

The SX-8 was chosen by way of example as such a supercomputer is installed at the High-Performance Computing Center (HLRS) in Stuttgart, which is one of the 5 federal German high-performance computing centers. There the parallel version of GeoSys/RockFlow, which is introduced very briefly in chapter 4, was implemented and tested. It is exactly for this parallel project that the preprocessing of the data has to be accomplished.

3.6.7 Computer Cluster

A typical *cluster* (or sometimes also called a *commodity cluster*) is a collection of mass-produced computers and (network) switches which operate independently but are connected through a special cluster software. They appear virtually as one "big" and powerful computer. The coupling between the individual cluster computers, so-called *cluster nodes*, is via a simple standard networking technique like, for example, *Ethernet* with 10Mbit/sec or 100 Mbit/sec. All cluster machines run the same version of operating system software. A data server is also present, which is shared by all nodes.

Usually such clusters can only be accessed via the network they are installed in. The cluster components generally do not have a LCD, monitor or a keyboard attached. Sometimes average users might not even be able to log in into the individual nodes. When submitting jobs or programs

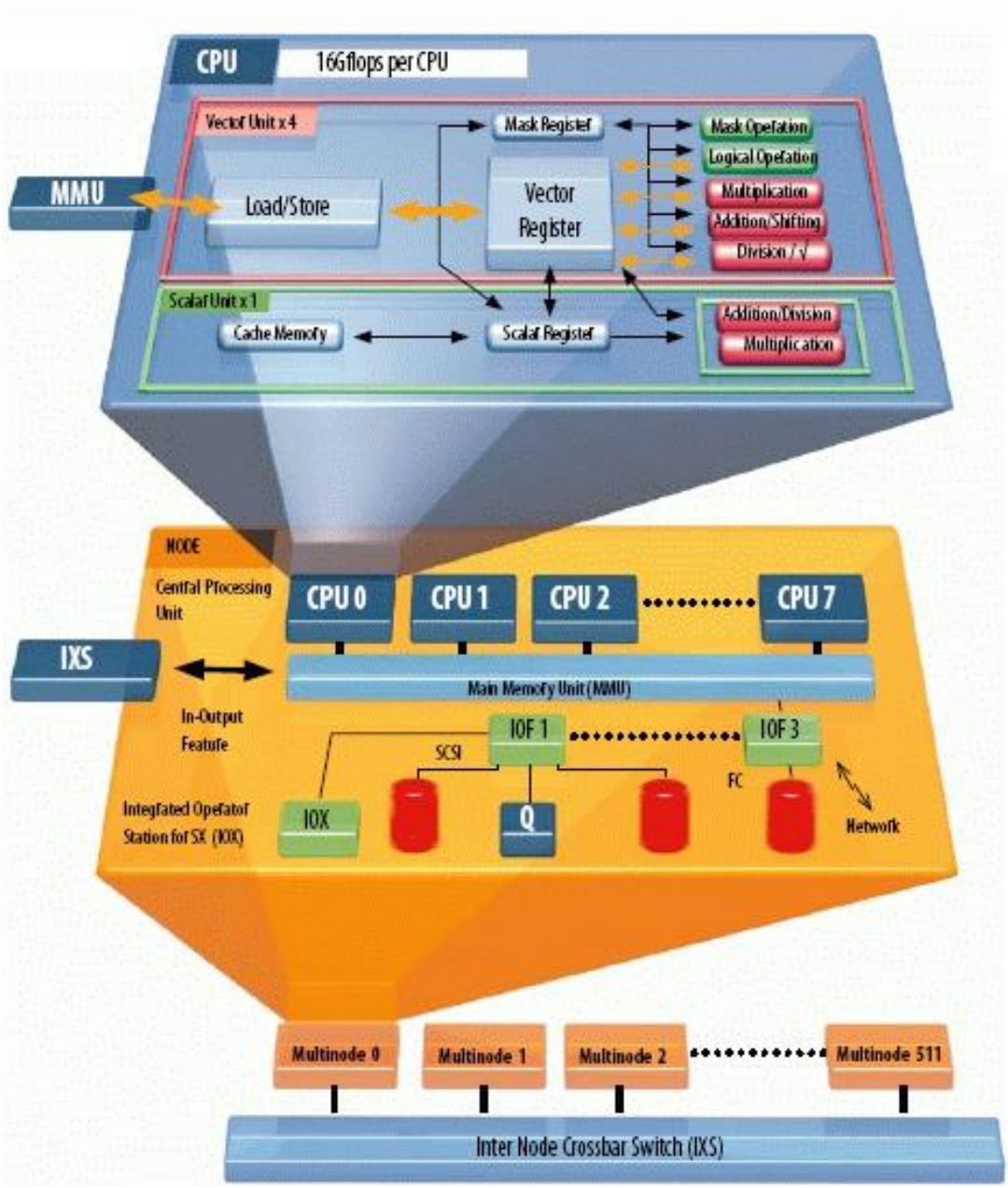


Figure 3.15: Hierarchical architecture of the NEC SX-8 vector supercomputer

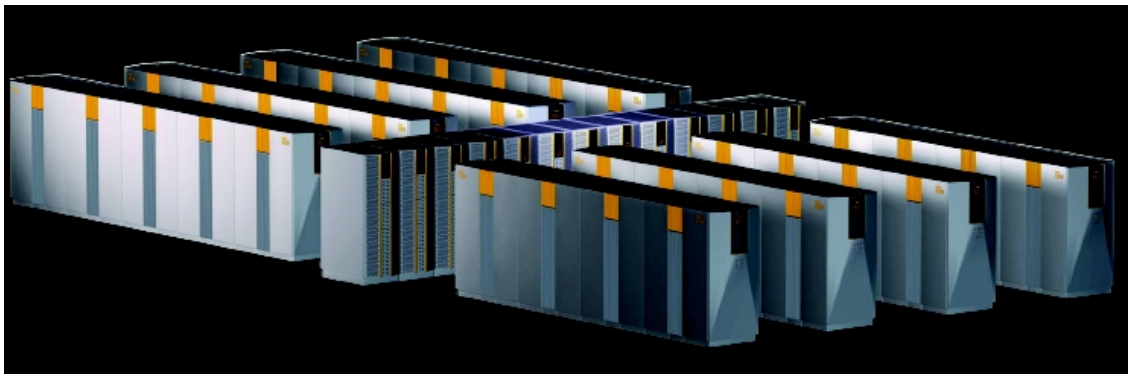


Figure 3.16: Medium-sized NEC SX-8 model

to the clusters they will be accepted by a dedicated node, a *front-end*, which will then distribute the task among the other nodes.

In case the installed hard- and software is identical on all nodes of the cluster, it is a *homogeneous cluster*; otherwise it is an *inhomogeneous cluster*. A so-called *Beowulf cluster* is one that runs *Linux* as the main operating system, whereas one which runs *Windows* is called a *Wolfpack cluster*.

Fig. 3.17 shows a small cluster which is set up from off-the-shelf desktop computers. There is just one (input) terminal with a keyboard and monitor from which the cluster nodes will be configured and administrated, if necessary.

Figure 3.17: Small cluster which is built from off-the-shelf components

When looking at Fig. 3.17, one big advantage of such a cluster is directly evident: It is easily scalable in comparison to an expensive and sophisticated supercomputer. In case the cluster size should be increased, additional machines need only be bought and "plugged in." One major drawback can be derived directly from the size of the cluster: With increasing size (number of cluster nodes) the operating expenses for administration and communication load also rise.

Installing and using clusters is reasonable if the problems to be worked on can either be easily distributed among several nodes or broken down into several independent sub-problems which could then run in parallel on the different cluster components. Distributing the problem focuses towards load balancing, whereas subdivision of the task aims towards an increase of performance.

Within the last few years especially Beowulf clusters have become very powerful. Nowadays there is a larger number of cluster architectures on the Top500 list of the fastest computers of the world.

3.6.8 Grid Computing and Distributed Computing

The key concept of *grid computing* is the usage of many networked computers on which the computing problem is distributed and which model a virtual computer. At first sight this sounds like the basic definition of a cluster. Unlike a cluster, the single nodes of a grid are often widespread and can range from big supercomputers or multiprocessor systems to small desktop machines or workstations or even to clusters themselves. All participating computers of the grid are "plugged" together over the network infrastructure. Generally they are connected via the Internet. A cluster operates just locally instead and "offers" only computing power; no sophisticated storage modes or user-access systems are required, however.

The key term behind a grid architecture is the *grid resource*. Such resources can be, for instance, CPU cycles and/or disk storage of large numbers of disparate computers, treated as a virtual cluster embedded in a distributed telecommunications infrastructure. Grid computing's focus on the ability to support computation across administrative domains sets it apart from traditional computer clusters or traditional distributed computing.

The name *grid computing* came up in the early 1990s as a metaphor for making computer power as easy to access as an electric power grid. The original idea behind the exploitation of grid computing was the goal of solving problems too big for any single supercomputer, whilst retaining the flexibility to work on multiple smaller problems. Thus grid computing provides a multi-user environment. Additionally, as a side effect of the original idea, grid computing makes very good use of available computing power and in parallel caters to the intermittent demands of large computational challenges.

Ian Foster, Carl Kesselman and Steve Tuecke are the so-called "fathers of the Grid" (see [Foster02] and [Foster03]). They also led the efforts to create the Globus Toolkit [Globus] incorporating not just CPU management (e.g., cluster management and cycle scavenging) but also storage management, security provisioning, data movement and monitoring as well as a toolkit for developing additional services based on the same infrastructure, including agreement negotiation, notification mechanisms, trigger services and information aggregation. Although the Globus Toolkit remains the de facto standard for building grid solutions, a number of other tools have been built in the meantime which answer some subset of services needed to create an enterprise grid, for instance.

The aforementioned terms clearly show the huge difficulties that the fantastic idea of grid computing still faces today: First of all, grid computing involves sharing heterogeneous resources (based on different platforms, hardware/software architectures, and computer languages), located in different places belonging to different administrative domains over a network using open standards. In short, it involves virtualizing computing resources. Furthermore, as remote users have to log in and use the grid resources, state-of-the-art secure authorization techniques for the control of the computing resources are obligatory.

Depending on its functionality, such a grid can be classified as one of the following grid types:

- *Computational grids:* they focus primarily on computationally intensive problems
- *Data grids:* they allow a controlled sharing and management of large amounts of distributed data

- *Equipment grids:* such grids have a primary piece of equipment, like, for example, a telescope, and the surrounding grid is used to control the equipment remotely and to analyse the data produced

- *Service grids:* resources are no longer "visible" to the users. The service provider defines the service he or she is offering on an abstract level. The user no longer knows how this service is then provided

- *Information grids:* their main purpose is just to provide all kinds of information. (The Internet with its services like WWW and FTP can be taken as an omnipotent information grid.)

Data grids, computational grids and equipment grids are often summarized under the term *resource grid*, as they provide some kind of resource. Depending on its size, a grid can be categorized as either a *departmental grid*, an *enterprise grid* or – as the biggest possible size – a *global grid*.

The *National Science Foundation* as well as the *NASA* are successfully applying their *National Technology Grid* or *NASA's Information Power Grid*, just to name two of today's grid users.

The term *distributed computing* is often incorrectly used as a synonym for grid computing. Although very similar to grid computing, distributed computing employs a central server to delegate and assign "chunks" of computational intensive problems to so-called *clients*. Such clients then work on their own and upon completion, the result is sent back to the central machine which then assembles the overall result at the end.

Projects using distributed computing normally have their own protocols which are generally not standardized. They are just focused on the problem or computing task in question. The most widely distributed computing project is *SETI@home* of the University of Berkeley, CA [Seti], which was launched in May 1999. Instead of using special-purpose supercomputers located at the telescope for the bulk data analysis, a large number of Internet-connected computers now take over this part. Users can subscribe and offer their free CPU time and resources for the computation of the signals.

3.6.9 Example of a MIMD Architecture

Up to this point quite a few different architectures found in the world of parallel programming have been introduced. Many of the really "big" machines, though, are quite something special. Therefore, the *Earth Simulator* (ES) already mentioned in the subsection of MIMD systems will be briefly described.

The ES was intentionally chosen here as an example of a supercomputer because its main purpose of construction and installation was the simulation of environmental and geological problems. Especially the investigation of climate and the serious problem of pollution threatening our environment, influencing the weather and leading to environmental changes have become more and more important in the past decade. The ES was the first supercomputer to be built for HPC aiming at such problems and headed the TOP 500 international list of all supercomputers in the world for two years straight (06/2002 to 06/2004). Although it is a rather "old" machine in the fast developing world of HPC (it started its work officially in March 2002), it still ranks 30th in the most recent list at the time of this writing!

The construction of the ES nicely shows how the different fields in the world of science finally merge. The ES is a "masterpiece" of computer hardware originating in the world of computer science and aimed at the world of mathematics with their simulations and complex modelling to produce a good forecast of geoscience matters.

The ES is a highly parallel vector supercomputer system which implements distributed memory in hardware, and consists of 640 processor nodes. These are connected by 640 $\times$ 640 single-stage crossbar switches. Each processor node itself is again a system with shared memory, consisting of 8 vector-type arithmetic processors, a 16-GB main memory system, a remote access control unit, and an I/O processor [Earth]. Fig. 3.18 [Earth] shows a system configuration of the ES and the components one processor node consists of.

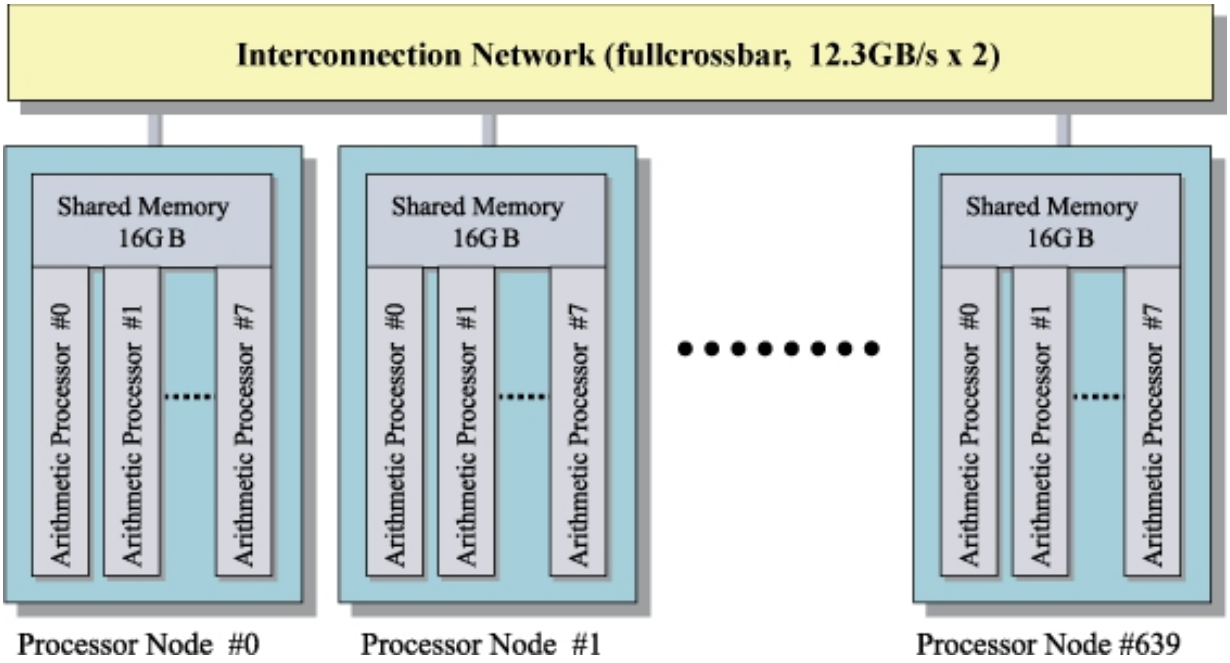


Figure 3.18: System configuration of each of the processing nodes of the *Earth Simulator*

The peak performance of each of the arithmetic processors is 8 gigaflops. The ES as a whole thus consists of 5120 processing nodes with 10 terrabytes of main memory and a theoretical performance of 40 terraflops. The room in which this gigantic supercomputer is installed has a dimension of 50 m $\times$ 65 m. The photo in Fig. 3.19 is taken from inside the giant room where the different operating nodes of the ES are installed. It is not possible to fit everything on just one single photo without special photographic equipment, as the size is prohibitive.

The building itself is a two-story steel construction with a seismic isolation system (the whole building is situated on thick "rubber pads"), a special lightning protection system, extra double floors for cabling, a very powerful air conditioning system and a special power supply facility for the extremely high power consumption of the ES. As a comparison: Annual power consumption of the ES is 112 GWh/year; the annual power consumption of the Austrian city of Linz plus another 82 municipalities was 476 GWh in 2001 [Zinterhof]! Fig. 3.20 [Earth] gives a simple but significant overview of the computing center where the ES is installed.

Fig. 3.21 [Earth] shows an even more impressive picture: This is a view into the extra double floor just for the cabling of the ES. The cables installed at the *Earth Simulator Center* have a total length of 2,890 km. (As a comparison: Each of the vertical cable bundles hanging down from the ceiling has the size of a fully grown, strong and muscular man!)

Figure 3.19: Partial view of the operating room of the *Earth Simulator*

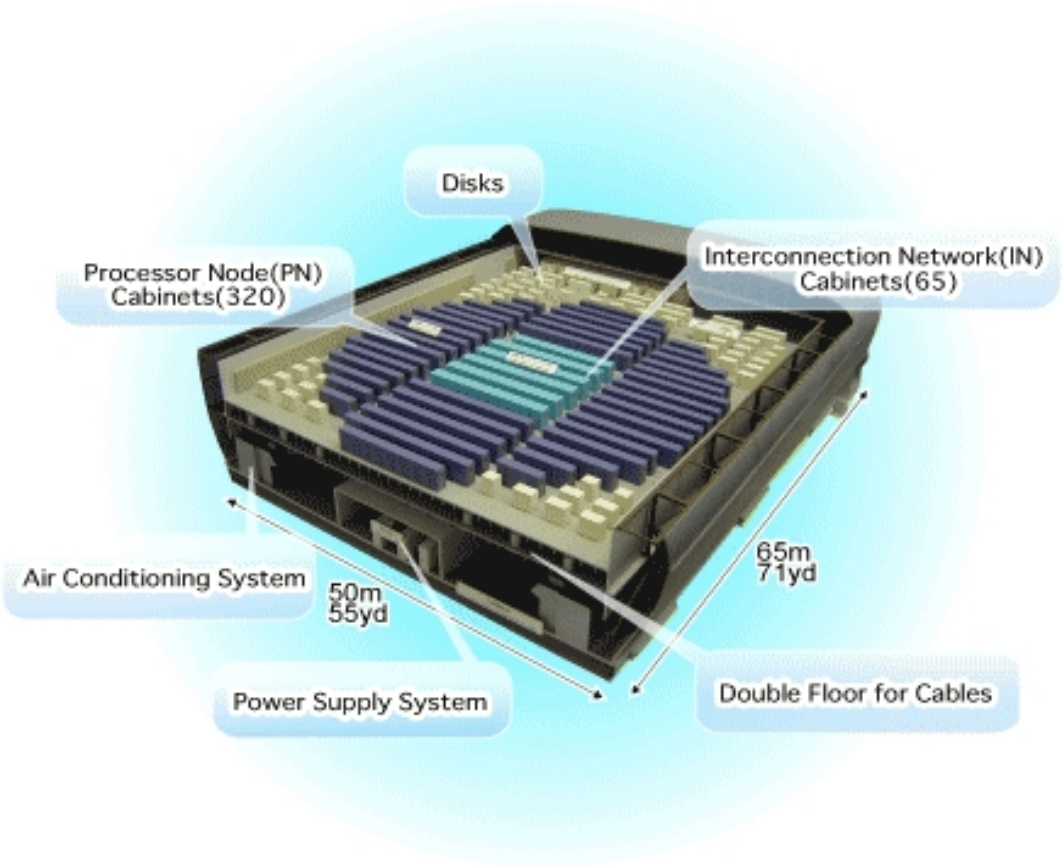


Figure 3.20: Overview of the *Earth Simulator Center* in Yokohama, Japan

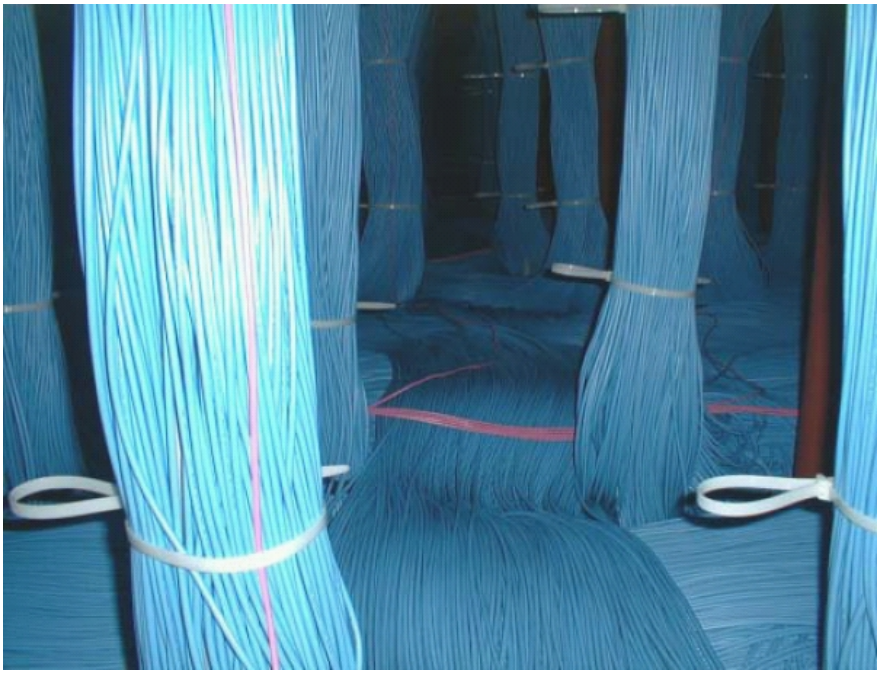


Figure 3.21: 2,890 km of cables connecting the 5,120 processing nodes of the *Earth Simulator*

3.6.10 Summary of Processor Architectures

Many different multiprocessing architectures have been introduced in the subsections above. When starting a parallel project or when switching from a sequential solution to a parallel version, one has to be very careful to analyze precisely what requirements exist. Not all architectures are suitable for every type of problem; the choice also depends on the goal to be achieved. Should the solution be computed much faster, for example, but with the same starting parameters? Or is time not an important factor, but rather the size of input data and starting parameters? All this has to be considered thoroughly before one finally decides on a multiprocessing system.

3.7 Interconnection Networks

A very important aspect with regard to parallel computing is the question of how the single operating nodes of a parallel computer are interconnected. Whole books have been written on this specific topic and the same is true for the best-suited routing algorithm, of how the messages are transferred between the processing nodes.

We shall not dwell upon this topic too long, as the user normally has no influence on this matter at all. Quite often the underlaying interconnection network is already well established and given through the existing hardware and architecture. The same also holds true for the routing algorithm.

With regard to the interconnection of the processing nodes of the parallel machine, terms like bandwith, latency time and scalability, just to name a few, are of quite some interest. For those who would like to learn and read more about this surely fascinating field of computer science, many interesting facts and whole chapters can be found in [Rauber], [Foster01], [Ungerer01] and [Quinn], which provide a good starting point.

3.8 Load Balancing and Domain Decomposition

The aspect of *load balancing* and *domain decomposition* is significant in any discussion of parallelization. As all of chapter 5 is devoted to this topic, it is only touched upon briefly here.

Load balancing is a certain technique to spread work between two or more computers, also network links, CPUs or any other type of resources, in order to achieve an optimal recource utilization, to maximize the throughput, minimize response time, etc. Also using multiple components instead of one, single component, may increase reliability through redundancy.

Domain decomposition is a method of splitting one major problem domain into smaller subdomains in which the problems to be computed are independent (if possible). This way the single subproblems can be computed simultaneously which makes the method suitable for parallel computing. Such domain decomposition methods are normally applied in numerical analysis or numerical partial differential equations, for example.

When working either on load balancing or on domain decomposition, we are confronting the "chicken and egg" problem: Which came first? This also applies to the field of parallel computing. When dealing with domain decomposition, this directly affects load balancing and vice versa. They are directly related.

When parallelizing a given problem, one of the main goals is to obtain a substantial speedup (see also subsection 3.10.3). This can only be achieved if the model or problem is equally distributed onto all participating processing nodes. In this way all processors can work at the same time and none of them would have to wait for the others. The work load is balanced in such a case over all the processing units involved. At first sight this sounds quite simple and seems to be a minor problem. But on close look this task suddenly turns out to be of major importance and shows that it is not an easy-to-solve problem at all.

The fact that we are dealing with a supercomputer or parallel computing environment automatically implies that the models or simulations exceed an average problem size which cannot be solved using a normal desktop machine or PC anymore. Behind these models with such enormous amounts of data, complex mathematical computations are often hidden, as in the example of the weather forecast in the motivation chapter (see chapter). It is also clear that the equations necessary depend on each other in most cases and that it is impossible to just simply "chop" the model into handy pieces without providing for such dependencies.

In whatever way the original model is broken down into smaller chunks, this always affects the load balancing of the parallel computer. The optimal case, where the subdomains resulting from the domain decomposition could be evenly distributed, hardly ever occurs due to the complexity of the problems which are to be computed. Usually, the so-called balancing factor for the domain decomposition will be slightly bigger than 1, which would be the optimal case. Thus, the resulting parts from the domain decomposition vary a little bit in size and this of course then leads to more or less idle time for the affected processing node.

The problem of load balancing and of domain decomposition is a research area in itself within the field of computer science. Such a mapping problem of how to distribute the single chunks of the original model onto the processing nodes is called NP-complete (stands for *non-deterministic polynomial time*, see section 5.2 in chapter 5, starting on page 99). Unfortunately, up until today, there are almost no efficient sequential or parallel algorithms known that solve this problem sufficiently.

Some commonly known partitioning tools on the market like *Jostle* or *Metis* have implemented sequential graph-partitioning algorithms based on effective heuristcs; most of the time they achieve good balancing factors close to 1.

Chapter 5 will dive into more detail in regard to the basics and the theory behind load balancing and domain decomposition. It also gives a detailed overview of *Jostle* and *Metis*, as these have both been integrated into the wrapper program for the preprocessing of the GeoSys/RockFlow (see chapter 4) data sets.

3.9 General Programming Techniques for Parallel Systems

In [McGraw] four distinct paths for the development of software for parallel computers are identified:

- Extension of an existing compiler to translate sequential programs into parallel programs
- Extension of an existing language with new operations that allows users to express parallelism
- Addition of a new parallel language layer on top of an existing sequential language
- Definition of a complete new language and compiler system

Generally, each of these 4 different suggestions has its advantages and disadvantages. But in most cases only the third item of the list is really applicable.

Although dozens of parallel programming languages have been introduced in the past, not one has gained widespread acceptance in the parallel programming community [Quinn]. Instead, most parallel programming continues to be either in Fortran or in C, augmented by functions that perform message passing between the processing nodes. The MPI (message passing interface) standard is now the most popular message passing specification. Almost every commercial parallel system has support for MPI programming, and even free libraries conforming to the MPI standard are available. The next section gives a brief overview of MPI.

3.9.1 MPI

MPI provides many different and powerful commands for parallel programming and is usually implemented as an additional library to standard C or Fortran. The official MPI standard (see [MPI-1] and [MPI-2]) meanwhile comprises two complete books about the basics of MPI and important extensions. When programming with MPI useful commands like

- `MPI_Init` to initialize MPI
- `MPI_Comm_rank` to determine a process's ID number
- `MPI_Comm_size` to find the number of processes
- `MPI_Reduce` to perform a reduction operation
- `MPI_Finalize` to shut down MPI
- ...

(just to name a few) are available.

Originally MPI was developed to run in a distributed memory environment (see subsection 3.6.5) at a time when a modest number of CPUs cost hundreds of thousands of dollars. Nowadays, already small multiprocessor systems are available at just above the price of a well-equipped home computer. But such systems rather comply with shared memory systems.

Although it is possible to write parallel programs for multiprocessors using MPI, better performance can often be achieved by using a programming language which is tailored to a shared memory environment (see subsection 3.6.3).

3.9.2 OpenMP

OpenMP is an application programming interface (API) for parallel programming on multiprocessors. Recently, it emerged as a shared memory standard and consists of a set of compiler directives, a library of support functions and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. OpenMP is maintained and further improved by interested parties from the hardware and software industry, government and academia. Originally it was mainly conceived for shared memory environments, but in the meantime it has also achieved acceptance in the programming world of distributed shared memory systems (see subsection 3.6.5).

Below is a small excerpt from the powerful set of compiler directives which OpenMP provides:

- `parallel` precedes a block of code to be executed in parallel
- `for` precedes a *for* loop with independent iterations that may be divided among threads executing in parallel
- `parallel for` combines the power of the `parallel` and `for` directives
- `sections` precedes a series of blocks that may be executed in parallel
- `parallel sections` is a very useful combination of the `parallel` and `sections` directives
- `critical` precedes and marks a critical section
- ...

Both MPI and OpenMP have their advantages and disadvantages. The existing sequential code or the multiprocessor environment must be quite thoroughly analyzed to benefit from the improvements provided by these parallel programming extensions. The choice of an implementation model is largely determined by the type of computer architecture targeted for the application, the nature of the application and a "healthy dose of personal preference" [Chandra].

Readers who are interested in either MPI or OpenMP and would like to read and learn more about them, can find very good documentation either in the form of books or on the Internet. For MPI the *MPI standard* [MPI-1] and citempi2 is definitely a must; or one can check out their website [MPI-3]. A very good book is also [Quinn], which shows the use of MPI while programming C.

For those interested in OpenMP, the best starting point is the official website [OpenMP]. Readers who rather prefer using a good book can have a look at [Chandra], which is also recommended on the OpenMP homepage.

3.9.3 Hybrid Programming – Combining MPI and OpenMP

It is quite common in the meantime to combine MPI and OpenMP, which then is called *hybrid parallelization*. They are easy to combine to have a dual paradigm program. Generally the combination of the two results in two levels of parallelism: The MPI decomposition shows typical block-level parallelism (see also subsection 3.2.3), where each processing node is given a section or block of the complete problem with which to work. Adding the OpenMP directives gives a loop-level parallelism (see subsection 3.2.2). Thus, MPI is used for a "coarse-grain" type of parallelization whereas OpenMP implements a "fine-grain" type of parallelization. The parts of the program which are affected by OpenMP are completely independent of the modifications for MPI.

The advantage of OpenMP over MPI is that the parallelization of already existing sequential programs is achieved much faster and more simply. Also, parallel programming from scratch is a great deal easier with OpenMP than doing the same job with MPI. On the other hand, MPI is much more flexible to use, as it can be deployed either in SM MPS (see subsection 3.6.3), DM MPS (subsection 3.6.4) as well as in DSM MPS (subsection 3.6.5). If the program structures or grid structures are very complicated, OpenMP is no longer applicable, in contrast to MPI. In conclusion, it can be stated that only a program with suitable coarse- and fine-grain structures can be effectively parallelized by hybrid programming techniques.

More on the topic of hybrid programming can be found in the book by [Quinn].

3.10 General Parameters for Performance Analysis

If the reader has made it up to here in this introductory chapter, it is hopefully clear to him or her how many different but important aspects must be taken into account when setting up a parallel system or when parallalizing a serial program. Independent of the starting situation, what counts is only how much of an improvement the final result is over the initial conditions. Especially program performance analysis is an ongoing, integral part of program development.

When talking about speeding up computations or decreasing execution time on the computer, many terms are used which are often not correct for a given situation. The following section gives an overview of the main important parameters which are generally applied when it comes to measuring time-dependent aspects in the comparison of sequential and parallel programs.

3.10.1 Runtime Approximation for Sequential Programs

When evaluating or discussing the runtime of a program, the goal would be a statement like the following: "The running time of a program is $T(n)$ units if the input has size n." [Pacheco]. The actual time a program takes to solve a problem – the time from the beginning of execution to the completion of execution – will depend on factors other than just the simple input size. Typical questions are generally:

- Which hardware is being used for the execution of the program?
- What programming language is being used?
- What compiler is going to translate our source code into machine code?

- What is the type of input? Simple arrays or long, structured data sets from a data base?
- What is the estimated rate of communication during program execution?
- ...

Some of the questions above apply not only to sequential programs only but also to parallel code. It is also clear in regard to the multiple questions that we can't entirely avoid all of the problems that arise when estimating the runtime of program. Therefore, we will always introduce some "imprecision" into our measurements.

There are certain constructs and statements in a code which determine the actual runtime of a program, like long loops or complicated computations. When checking one's serial program code, which is the starting point for all evaluations to follow, an approximation something like this one results:

$$T(n) \approx kn,$$

with a given input size of *n*. Now, if we increase n by a factor of r, $T(n)$ should increase by the same factor. If $T(n) = t$ and our input size increases to rn, we would predict logically a runtime of

$$T(rn) = rt.$$

As one can see clearly, this is a simple linear equation. Although every scientist should understand such a simple mathematical expression, the following example will be given for reasons of completeness.

Let us recall Peter, our Roman contractor, and his laborer Paul from the motivation chapter. Peter knows that Paul can dig 1000 cubic meters per day. If Peter now accepts a contract for the excavation of 10,000 cubic meters, he quickly realizes that Paul can't finish any sooner than 10 days from the very second he takes his shovel and starts digging under perfect and optimal conditions (assuming that Paul will not get sick, and the ground is composed of a material that permits Paul to indeed dig his 1000 cubic meters/day and thus meet Peter's expectations.)

3.10.2 Runtime Approximation for Parallel Programs

When porting a sequential program to a parallel computer we assume for reasons of simplicity that there are no big changes in our code. This way, the approximation of the sequential execution time gives a good basis for a comparison with the outcome of the parallel program execution.

Nevertheless, we need to take a closer look at what happens during the execution of a parallel program on a parallel computer. As we have n different processors on each of which this program is being executed, it takes a certain *startup time* T_{stime} until all necessary information is distributed, administrative tasks of the operating system are performed in the background, etc.

At some point the parallel computer is able to start the program execution and now all processors which are involved in the execution operate independently of each other. This time we can describe as *computing time* $T_{compute}$.

Most likely, during the execution of a parallel program the independently operating processors need to share information or need to synchronize each other to finally succeed in computing the correct result of the program together. Therefore we will experience some time needed by the

computing nodes of the parallel computer to communicate with each other. This time is called *communication time* T_{comm}. As not all processors might be able to respond right away to a request from other processors because they might be right in the middle of a complex calculation of an equation, for example, we also have to take into account some waiting or *idle time* T_{idle} until communication between the operating nodes can be started.

The total execution time for an arbitrary processor i of the p independently operating nodes executing the parallel program on the parallel computer will result in the following approximation:

$$T_i = T_{i_{stime}} + T_{i_{compute}} + T_{i_{comm}} + T_{i_{idle}}$$

Hence, the total execution time T of the parallel program can be defined as the sum of startup, computation, communication and idle time over all processors divided by the number of processors p [Foster01]:

$$T = \frac{1}{p}\left(T_{stime} + T_{compute} + T_{comm} + T_{idle}\right) =$$

$$\frac{1}{p}\left(\sum_{i=0}^{p-1} T_{i_{stime}} + \sum_{i=0}^{p-1} T_{i_{compute}} + \sum_{i=0}^{p-1} T_{i_{comm}} + \sum_{i=0}^{p-1} T_{i_{idle}}\right)$$

Fig. 3.22 gives an example of how activities could be distributed during the execution of a parallel program on 8 different processor nodes of a parallel computer. As can be seen, each of the 8 processors spends a certain time computing, communicating and even doing nothing, maybe as it is requesting synchronization with other computing nodes or just waiting for a chance to communicate for the exchange of important data.

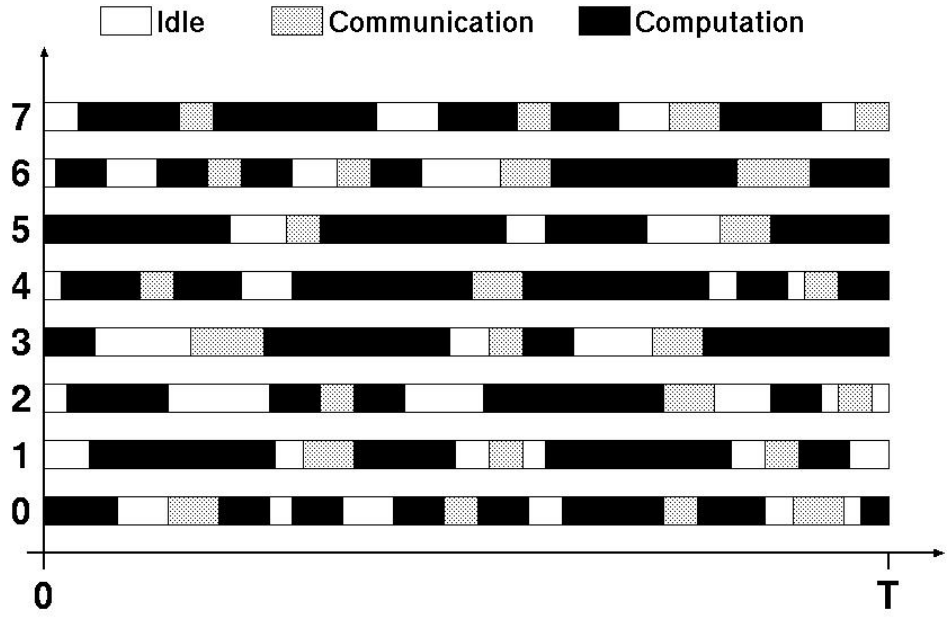


Figure 3.22: Activity plot of a parallel program during execution on 8 processors

What this all means can again be nicely transferred into a daily situation by Peter, the Roman contractor, and his company. When Paul and his colleagues arrive in the morning at the excavation

site, it takes a little bit of time until they are able to start their work. The reason is that they have to get their shovels and equipment ready, find and take their location from where they are going to continue work and adjust bucket and equipment in a convenient way to dig as productively as possible. This is nothing but the startup time which every processor also needs before it can start with its computations.

During their work, the laborers and Paul have to see that they don't get into each other's way. This means they have to communicate and agree on how to continue with their work. Discussing and making plans does take some time in which they can't continue to work. Peter shows up on the excavation site once in a while, coordinating the work and making sure that nobody digs around in an area where Peter would not like them to work. In such a way he is "synchronizing" his laborers and seeing to it that they succeed all together with the excavation as well and quickly as possible.

In the end nobody cares anymore who worked on which spot the most or who dug the fastest. Only the excavation itself is what counts in the end, independent of how much each of the single laborers participated in the result.

3.10.3 Speedup

The main reason for developing parallel programs is the hope that they will run faster than their sequential counterparts [Quinn]. The clear difference between these two program executions is the fact that the runtime of a parallel program should depend mainly on two variables: the input size n and the number of processors p which can be used during the execution of the parallel program. Thus, *speedup* is defined simply as the ratio between the formerly sequential execution time of our program and the resulting parallel execution time:

$$Speedup \ = \ \frac{Sequential\ execution\ time}{Parallel\ execution\ time}$$

or, using our denotation given in subsection 3.10.1, we can also write:

$$S(n,p) \ = \ \frac{T_s(n)}{T_p(n,p)}$$

[Lee] states that for a fixed value of p, it will be usually the case that

$$1 \ \leq \ S(n,p) \ \leq \ p$$

This is also known as *Lee's principle*.

If $S(n,p) \ = \ p$, a program is said to have *linear speedup*. This is of course a rare occurence, since most parallel solutions will add some overhead because of the communication among the processes running on the different processors. Generally, a far more common occurence is the so-called *slowdown*, as the parallel program running on more than one processor is actually slower than its sequential counterpart. This effect results from an excessive amount of overhead mainly due to communication and administration. This fact can be nicely seen in Fig. 3.22, where the times of computation are often interrupted.

What this means in real life is again demonstrated by Peter, our Roman contractor. Let us assume that he accepts a job of excavating 400,000 cubic meters in not more than 30 days. With every day that he finishes earlier he'll earn a nice extra bonus in addition. At first sight, the reader would immediately suggest that Peter simply hire another 19 good laborers like Paul, and terminate this lucrative digging job easily in 20 days, all of his new employees excavating 1000 cubic meters as Paul does. But Peter is a wise and experienced contractor and he purposely closed this contract with 30 days instead of 20 days, most likely still making good extra money. Why?

Peter definitely hired another 19 men as the reader would have suggested, meaning a maximal excavation process of 20,000 cubic meters per day. But, when all of a sudden more than one man needs to be supervised, and people work side by side, the work needs to be directed and time is always lost. We need only consider the fact that it is no longer possible to just "walk straight"; now the laborers need to circle each other, discuss who is digging where and how to get rid of the soil, etc. Even with a perfectly organized boss and well-motivated laborers, they could not be as effective as a single employee Paul. Thus, on the basis of his experience, Peter had calculated all this from the very beginning, knowing that his hard-working men would give their best and that they most likely would finish the excavation in 23 to 24 days (still giving him a nice extra bonus of 6 or 7 days!)

3.10.4 Efficiency

As we have seen during the discussion of *speedup* in the above section, many different factors influence our resulting execution time. Without going into further details, linear speedup can be an insufficient measure for the quality of a parallel algorithm. Therefore quite often an alternative performance measure is used, the parallel *efficiency*.

This more convenient metric to evaluate our effectiveness is defined as the ratio between our original sequential execution time and the parallel execution time in regard to the utilization of the processors involved in the parallel version:

$$Efficiency \;=\; \frac{Sequential\ execution\ time}{Processors\ used \;\times\; Parallel\ execution\ time}$$

which can be written as

$$E(n,p) \;=\; \frac{T_s(n)}{p \;\times\; T_p(n,p)} \;=\; \frac{S(n,p)}{p}$$

using the notation introduced in subsection 3.10.3, where n symbolizes the input size of our problem and p the number of processors working in parallel which are used to solve the given problem. Applying *Lee's principle* from 3.10.3 above, we find that efficiency is characterized by

$$\frac{1}{p} \;\leq\; E(n,p) \;\leq\; 1.$$

Plainly put, this inequality simply means that if the efficiency stays close to $1/p$, we would have been better off with our sequential program. The more the efficiency approaches 1, the better our parallel version of the program does. In other words: *Efficiency* characterizes the effectiveness

with which an algorithm uses the computational resources of a parallel computer in a way that is independent of the given problem size [Foster01].

What does this mean practically? We have seen in subsection 3.10.3 that even if Peter had the most motivated and best-digging laborers in the entire Roman Empire, they couldn't be as effective as one single excavator working day after day on his own, just needing much more time for the job in total. The reason for hiring more than one worker is still time-related matters. And here we can see a big difference between speedup and efficiency.

Let us return to our example in subsection 3.10.3, where Peter will earn a nice extra bonus for every day he and his men finish the 400,000 cubic meters excavation job earlier. We assume the same conditions and that his men are all motivated and hard working and loyal to Peter. In scenario one, each excavator is working on his own and carrying away his filled buckets, which means that he has to drop his equipment quite often, cease working and walk all the way to the location where the soil and rocks are collected and piled up. Upon his return to his workspace, he has to sit down again, pick up his equipment and restart his work. This way, Peter's laborer might be able to finish the excavation in, let us say, 24 to 25 days.

In scenario two, in comparison, Peter assigns two of his workers the job of bucket carrying. Of course, he is now short two motivated diggers, but now the other 18 men can work continuously without interruption, while two take care of the bucket business only and thus help the others be more productive. In the end Peter is more effective and therefore more efficient and might even be able to finish his excavation job in 23 to 24 days!

3.10.5 Scalability

Up to this point our performance analysis pertained to quality and execution time. Unfortunately such parameters fail when questions like the following need to be answered [Foster01]:

- Does the algorithm meet design requirements (execution time, memory requirements, etc.) on the target parallel computer?
- How adaptable is the algorithm? That is, how well does it adapt to increases in problem size or processor counts?
- How sensitive is the algorithm to machine parameters like, for example, transfer time or startup time?
- How does the algorithm compare with other algorithms for the same problem?
- What difference in execution time would be expected if a different algorithm had been used?
- ...

An important aspect of performance analysis is therefore the study of how algorithm performance varies with regard to execution time and efficiency with changing parameters such as problem size and processor count; in particular, how effectively parallel algorithms scale with an increasing number of processors.

If we evaluate scalability, either the problem size can be kept or the processor count increased, which generally results in a monotonically decreasing efficiency curve. At some point, depending

on the algorithm and the particular problem which needs to be solved, it may not be productive to use more than a certain maximum number of processing nodes.

This is easily demonstrated with Peter's excavating company. With regard to the size of the excavation job that Peter accepted, it makes no sense at all to hire more people than can work efficiently on the excavation site! When the laborers start to "step on each other's feet," Peter has definitely too many employees.

In case our problem size varies, the parallel program is called *scalable* if, as the number of processing units p is increased, we can find a rate of increase for the problem size n, so that the efficiency remains constant [Pacheco].

Here again our excavation example will demonstrate what is meant. In case Peter accepts a bigger and more work-intensive excavation job, he definitely will also hire more laborers – not too many, so that they won't get into each other's ways, but also not too few, so that those who finally do the digging do not have to work like horses.

3.10.6 Utilization

The last parameter to be discussed here is the *utilization*. It is the standardized parallel index and gives the amount of operations every processing node has performed on the average per time unit:

$$U(n,p) \;=\; \frac{P(n,p)}{p \;\times\; T_p(n,p)}$$

with n as our given problem size, p the number of participating processor nodes, $T_p(n,p)$ the complete computing time needed by all nodes, and $P(n,p)$ the total of operations over all processors. In the optimal case, he have $U(n,p) = 100\ \%$, but there is always additional work and expense when porting a problem onto a multiprocessor system, so that the 100 % will never be reached.

Translating these findings to the excavation example, the *utilization factor* gives us each laborer's grade of digging efficiency over the whole time. Perfect would be of course 100 %, but this is totally impossible. Therefore we might end up with a result of, for example, 75 % digging time, 10 % recreation time with meals and another 15 % for coordination and communication with the other excavators, supervisors or Peter.

The utilization gives us an upper boundary for the efficiency as follows:

$$\frac{1}{p} \;\leq\; E(n,p) \;\leq\; U(n,p) \;\leq\; 1$$

3.10.7 Example of Performance Analysis

A sequential program needs for its execution 1,000 operations which are performed in 1,000 time steps. A multiprocessor system with 4 processing units requires 1,200 operations, but these are accomplished in just 400 time steps. This results in:

$$n = 1000,\ p = 4$$

$$P(1000,1) = P(1) = 1000,\ P(1000,4) = P(4) = 1200$$
$$T_s(1000) = T(1) = 1000,\ T_p(1000,4) = T(4) = 400$$

With these values, speedup and efficiency can be calculated as

$$S(1000,4) = S(4) = \frac{T(1)}{T(4)} = 2.5$$

$$E(1000,4) = E(4) = \frac{S(4)}{4} = 0.625$$

.

Applying *Lee's Principle* (see subsections 3.10.3 and 3.10.4), this results in:

$$1 \leq S(4) = 2.5 \leq 4$$

and

$$\frac{1}{p} = \frac{1}{4} \leq E(4) = 0.625 \leq 1$$

For the parallel index and the utilization, the following values can be computed:

$$I(1000,4) = I(4) = \frac{P(4)}{T(4)} = \frac{1200}{400} = 3$$

$$U(1000,4) = U(4) = \frac{I(4)}{4} = \frac{3}{4} = 0.75 = \frac{1200}{1600} = \frac{1200}{4 \times 400} = \frac{P(4)}{4 \times T(4)}$$

The results of the computation state quite clearly that on average about 3 of our 4 processing nodes are working on our problem and each of the computing units is active about 75 % of the total computation time. The additional expenses $A(n,p)$ of the multiprocessing system with regard to our single-processing environment can also be calculated as follows:

$$A(n,p) = A(1000,4) = A(4) = \frac{P(4)}{P(1)} = \frac{1200}{1000} = 1.2$$

This means that in case we are porting our sequential program onto a parallel computer, we need about 20 % more operations in comparison to the execution of our program on a single-processor system.

3.11 Summary and Conclusion

Many different concepts, techniques and architectures in regard to parallelism and parallelization have been touched on in this introductory chapter. The focus was not on giving a complete, lengthy and detailed comparison to all related topics in this field. The goal is rather to provide a basic understanding of the manifold field of parallelization, be it from a programmer's point of view or that of a hardware designer or anyone else working in this area.

The terms, concepts and architectures introduced are just the "tip of the iceberg." Many important aspects had to be left out, like, for example, the complete field of networking architectures and cache models (they alone already fill complete books!), just to name two. With the topics described here in this chapter the reader is now able to advance in the text with a (hopefully) good understanding of the problems and difficulties parallelization has to cope with.

This introductory part also makes plain that it is impossible to look at only one single aspect when talking about parallization. Everything is connected, and taking care of just one specific part or task does not necessarily result directly in a highly and effective parallel code or solution. Only if the combination of all the means and techniques is as good as possible, can parallelization reveal its real power.

When looking at the results of what follows, it also becomes clear that parallelizing a given problem to its full extent is way beyond the scope of just one Ph.D. thesis. Instead, the combination of many little improvements and aspects will succeed in an effective parallel code. The work and resulting preprocessing program which is presented here in this thesis is just one of many necessary steps towards a measurable improvement of the existing originally sequential program GeoSys/RockFlow (see for more detail chapter 4) in its parallelization process.

Chapter 4

From Sequential to Parallel Modeling - With Emphasis on GeoSys/RockFlow

Up to this point the reader has already learned about the intricacy of preparing real-world problems and data in such a way that they cannot only be nicely displayed on a computer screen but also processed by the special modeling application presented in chapter 2. In case this specialized software is running on a parallel computer platform, chapter 3 described the various aspects that have to be kept in mind when working in a parallel computing environment.

The next important step necessary for the framework of this thesis is to have a closer look at the modeling software itself and how such modeling is accomplished in general. That is the goal of this chapter. Although it is rather short, it is nevertheless crucial for an understanding of the problems involved in the entire parallelization process.

The chapter begins with a short section about scientific modeling. This is followed by a brief description of a software package called *GeoSys/RockFlow*, which can be considered a placeholder application for any modeling computer suite in the field of geoscience. Then an explanation is given of how such modeling applications work in general and what main program parts they normally have in common. Towards the end of the chapter the problems which arise when switching now from a purely sequential modeling application to a modeling software version that does the same steps, only in parallel, are briefly discussed.

4.1 Scientific Modeling - A Short Introduction

To model complex and dynamic processes as they exist not only in geosystems but in all other scientific areas, exacting demands must be met by the scientists themselves, as well as by the computing environment and the software. The better the investigations and examinations of the given situation or problem, and the more thorough the model development, the more exact and accurate the outcome of the model solution later on.

The above sounds very convincing and reasonable, but it doesn't answer the question of *why* modeling is so important and essential nowadays. What exactly are we doing when we model certain circumstances or affairs?

The necessity of modeling simply breaks down to two main reasons:

- A new hypothesis, in the process of development, or an already existing one has to be checked and tested for correctness
- Certain parameters of a system or a hypothesis are difficult to measure in real-life experiments, are too expensive or the necessary testing apparatus does not exist

The second issue is especially interesting. Let us consider, for example, the pressure that is exerted on rocks deep down in the crust or the upper mantle. Due to this force the rocks are deformed in a certain way. But this process is very slow and sometimes takes millions of years. We just see the final result when we examine such rocks after they have been uplifted slowly through geologic eras. But still today with our modern and sophisticated machinery it is simply impossible to simulate the exact conditions that prevail deep down under our feet. Not only the pressure is extremely high, but also the temperatures, and to some extent time is an important factor. Only through appropriate models that simulate such conditions adequately can we examine the different processes that might be taking place there.

Most likely nobody will ever be able to directly observe these processes with the naked eye. Assuming one could, one still would not be able to see anything "spectacular," as the processes are so slow and progress made in such tiny steps that it would take some time to finally realize that changes had occured in the rocks at all.

Another fitting example is the contamination of the ground by oil or other chemicals, as already briefly described in section 2.1 of chapter 2 and depicted in Fig. 2.1. Depending on the contaminating material, it takes a certain time until the chemicals reach the aquiferous layers. And again, in dependence of the type of bedrock, the amount of groundwater, the hydraulic head and the chemical itself, it takes still more time until the pollutant spreads out as a big plume. Some contaminants might just "sit" right where they descended, others could swim "on top" of the water table and be taken away, and again, still others could be either mixed up with the water or even be "sucked" up by the bedrock and then settle in its micropores.

No reasonable human being would ever pour such contaminants out in the countryside just to measure their effects on the surrounding area. And even if he or she did so, the results obtained would always be very specific for only this particular area with its specific material properties and prevailing parameters. But if the ground or the contaminant or the parameters given were just to change a little bit, for example, 50 km away from the area where this person poured out the chemicals, the results or the outcome of this unfortunate experiment would most likely no longer apply.

Often interesting ideas for a hypothesis develop from an experiment or some imagined framework. Then modeling is an appropriate first step to seeing if there is at least some "truth" to the basic idea hidden in such a hypothesis, by simply playing around with the main parameters and rules.

Whatever the reason for modeling ultimately is, the process of scientific modeling is mainly the same and can be broken down into the following steps:

- **Problem Description:** What is the problem? - In this first step the general problem itself is identified.
- **Input and Output Definition:** Here the input and output of the model are identified. The parameters for the input are determined and also the precision and accuracy of the output are chosen.

- **Processes and Mechanisms:** What steps, equations, laws or or even experiment outcomes can be used? How should the solution of the given problem be found?
- **Model Construction:** What tools are needed? Which conditions must be met when solving the problem? Are any assumptions made to find a useful solution? Is a simplification necessary to come to solution at all? Are there already existing or similar models?
- **Model Implementation:** Depending on the field in which one works, the realization of such a model would differ slightly. This is the main step in producing a useful result of the model.
- **Model Validation and Tuning:** Is the model correct? Is the result precise and accurate enough? - This is the most crucial step in the whole chain of actions towards a useful model. Here the model is checked and, if necessary, even tuned towards an improved model.
- **Model Application:** If the previous steps have been successful, the model is now ready to be applied.

A final but important remark that most people normally never regard: One should always consider the field in which one works when talking about a "model" in general! For an artist a model can be something totally different than it is for a scientist, for example. And even in the world of science the models can vary quite a bit. One need only contemplate the slight but still existing differences of the following models: a numerical model, a statistical model, a physical model, a geological model, a computer model, a molecular model, a data model, etc. A short but very nice introduction to modeling in general can be found in [Bons].

4.2 The Program Package *GeoSys/RockFlow*

4.2.1 Why Choose GeoSys/RockFlow?

GeoSys/RockFlow (GS/RF) is a finite element geo-process modeling software tool [Kröhn]. It is mainly applied in fields of modeling and simulation of water resource management, geotechnics, design of geo-engineered barriers, exploitation of geothermal energy, soil and groundwater contaminant transport and remediation strategies. Its original design is based upon a sequential approach.

GS/RF is a complex modeling tool with a lengthy developmental history. Its beginnings date to the middle of the eighties of the last century. It should be noted though, that GS/RF can be seen just as a placeholder for any other modeling software. The reason for specifically choosing GS/RF here is simple and quickly explained: At the time this thesis was written, GS/RF was one of the best-known modeling packages at the School of Geosciences at the University of Tuebingen, and several groups were still actively developing the code in Tuebingen, Hannover and also Switzerland. It already had quite a reputation in the field of applied geosciences. Nevertheless, it is just an exemplary program. Any other software package could have been chosen instead.

The second reason for explicitly using this modeling software as an example here is that in the course of the growing problems that arose during its developmental process, the question of parallelization was posed. It was precisely the long existence and continuous development of GS/RF that finally made it a valid candidate for considering such a parallel version in the near future. And again, the problems that were encountered when coding GS/RF could also have arisen with other modeling packages.

4.2.2 A Closer Look at GS/RF

GS/RF is a modeling program that is based on finite element (FE) computations [Kröhn]. It dates back to 1985 and started out as a little project at the University of Hannover. Over the years, more and more tools and features were added. Originally written in Fortran, it was then re-written in 1996 in the programming language C to make use of dynamic data structures. In 2003/04 it was re-designed again to its current version in C++ to make use of the concept of object-oriented programming.

The key concept of C++ is object-oriented design, which is a completely different programming approach from the functional or procedural method used by C. In the most recent version of GS/RF, the data objects can be treated independently from each other. The objects have their own data constructs (= properties) and functions that operate on them (= methods). To communicate with each other, the objects can pass messages. This can also be utilized by superimposing a graphical user interface (GUI) which can then be accessed by the user of GS/RF [Kolditz].

GS/RF was designed and developed to solve special problems that arise in the field of applied geology or hydrogeology, where so-called THMC processes often must be solved [Zielke] [Kohlmeier]. The term THMC stands for "thermal-hydrological-mechanical-chemical." (Sometimes the "H" in THMC is also referred to as "hydraulic.") These THMC processes which can be worked on using GS/RF can be described further as follows [Kemmler]:

- **Thermal Processes:**
  - Heat transport with density changes
  - Non-isothermal multiphase flow with phase changes
- **Hydrological Processes:**
  - Groundwater flow in confined and unconfined aquifers
  - Multi-phase flow
  - Fracture flow, dual porosity
  - Density-dependent flow (thermal, tracer)
  - River flow (based on averaged 1-D Saint Venant equations)
- **Chemical Processes:**
  - Multi-component transport with density changes
  - Sorption models
  - Reactive Transport (i.e., Freundlich Isotherm)
  - Chemical reactions via coupling to PHREEQC2 (a computer program for speciation, batch-reaction, 1D-transport, and inverse geochemical calculations) [PHREEQC]
- **Mechanical Processes:**
  - Poroelasticity
  - Elastoplasticity (hardening)

A typical and quite recent application of modeling such problems can be found in the field of nuclear waste disposal, for example, when disposing this leftover in former mines. The remaining radioactive material is sealed in special containers which are then cemented in the pit by a clay mineral called bentonite. When dehydrated bentonite is humidified again, it starts swelling and

increases enormously in size. Using this effect by placing the radioactive waste containers inside such a coating of bentonite, they are easily insulated inside the bentonite and have no direct contact to the surrounding host rock, as clay is typically a very impermeable material [Ziefle].

Unfortunately, some reactions which are triggered by the heat of the nuclear waste inside the containers are possible. For the simulation of such cases, the corresponding parameters can be fed into a modeling program, like GS/RF, to examine the possible effects on the environment of the bentonite, of the surrounding host rock as well as of the containers themselves (see [Xie01] or [Xie02]).

Again, modeling is crucial in such cases, as no living human being - without being suicidal - would voluntarily examine the processes in such an environment. Even if someone did volunteer, the events and procedures taking place are too slow to be recognized by the naked eye. Only time reveals the variations in this system.

When modeling and computing such problems with a program like, e.g., GS/RF, three main operational steps can be distinguished: the preprocessing or input phase, the main model calculation, and the final postprocessing or output phase. Each of the three main phases can be divided further into more subphases, but this always depends on the program and its organization. A more general examination of such a workflow in a special, but sequential program for modeling can be found in section 4.3, which follows below.

The most work-intensive phase is surely the second one, when mainly quite complex equations must be solved. Here, various techniques and methods can be applied, which again depend on the type of the given problem. Especially in applied geology, initial-boundary-value problems arise from flow and transport processes in the subsurface systems. In general, these methods can be classified as analytical or numerical. Analytical solutions can be obtained for a number of problems involving linear or quasi-linear equations and calculation domains of simple geometry. For non-linear equations or problems with complex geometry or boundary conditions, exact solutions usually do not exist, and approximate solutions must be obtained. For such problems the use of numerical methods is advantageous.

In GS/RF, various methods are implemented: the *finite element (FE) method* and the *finite volume (FV) method* to calculate the spatial discretization of the regions that are worked on, and the *finite difference (FD) method* to approximate time derivatives. An introduction to FE, FV and FD is way beyond the scope of this work. The interested reader can find more on these topics in [Topping], [Meissner] or [Huckle], for example.

The models and data to be processed are read in from so-called *.rfi* and *.rfd* files (from "Rockflow Input" and "Rockflow Data"). The former holds the information regarding the grid and the coordinates of the nodes of the model; this is the geometrical information. (Appendix D, starting on page 257, shows some exemplary, small input test files.) The latter file stores all further information about the model itself, depending on the problem to be computed, which is the parameters, for instance, hydraulic head, fluid properties like viscosity or density, etc. The second file is controlled via certain keywords and takes care of the necessary adjustments for the computations, like the time-stepping scheme, material properties and physical as well as numerical parameters.

In the main computing phase this input data is then processed by solving big equation systems consisting, i.e., of partial differential equations (PDEs). What types of equations and so-called solvers are being used depends strongly on the imposed problem to be computed. As GS/RF is in its original version a sequential program, it iterates in internal loops over its solutions, refining the

results step by step. Also certain built-in tools, like preconditioners, can be used to accelerate the convergence of the iterative solver.

The final computations are stored upon program completion in two output files called *.rfo* and *.rfe* (from "Rockflow output" and "Rockflow Edits"). The `.rfo` file has the same structure as the `.rfd` file. The `.rfe` file is a protocol file that is written during the execution of the program.

Further information about the GS/RF program package can be obtained either at its website under `http://www.rockflow.de` or directly from one of the developing groups or partners, like the Institute of Fluid Mechanics and Computer Applications in Civil Engineering, the Federal Institute for Geosciences and Natural Resources, or the Leibniz Institute for Applied Geosciences, all in Hannover.

The `.rfi` files in particular were of major interest in preparing this thesis. They had to be read in and preprocessed in a certain way so that they could be used later on for a domain decomposition treatment, which is essential in the parallelization process. Only with a reasonable subdivision of the existing input data sets can the original model data then be spread among the different processing nodes of the parallel computer. Chapter 5 deals with the aspect of domain decomposition and its various constraints, and in chapter 6 the issues of implementing the correct data structures and the resulting side effects are discussed in more detail.

4.3 The General Process of Sequential (Computer) Modeling

Whereas the previous sections either dealt with scientific modeling in general or specifically with the modeling software package GS/RF, a modeling program in particular, this section now gives a more general view of the main steps in the purely sequential application of a scientific modeling process itself. By identifying the most important parts in such a serial application the necessary changes during a parallelization can be pointed out easily. Part of the process was already described previously in the GS/RF section above, but in case the reader has skipped that part, the main steps will be repeated here again without the GS/RF focus.

Let us start out by briefly dwelling upon the term *sequential* (also often referred to as "serial"). "Sequential" can be either derived from the old French word *sequence*, which was used for a "sequence of cards" or also the late Latin word *sequentia*, meaning "the following." Thus, when talking about a sequential code or sequential computer modeling, one is referring to the fact that everything is done step by step. A typical sequential program has a list of commands that are worked on one after the other by the processing unit. As a simple example, we can imagine a long line of dominos, set up one after another. When we push the first one in the row, they will all (hopefully) fall over. A domino itself is able to tilt only when its forerunner has fallen over. With this simple but effective rule the whole long line of dominos finally falls over.

This straightforward domino example also can be used to simply explain the difference to a parallel program. We still have our long line of dominos placed one after another. But now, there is a spot where we have placed two dominoes close together, side by side, with more dominos lined up behind these two adjacent dominos. When the first domino in the beginning single row is now pushed, the dominos of course fall over until they reach the point where the two closely placed dominos are waiting. The very second their forerunner tumbles, both dominos will fall simultaneously and now two lines of dominos will fall at the same time (a nice effect that is often used by the domino pros in various contests). As a logical consequence the double number

of dominos is "processed" in this way (if the placing rules and distances between the dominos were the same in both lines). Other examples for parallelism can also be found throughout the introductory chapter on parallel machines and aspects (see chapter 3).

Sometimes computer scientists use another, third expression when explaining the differences between sequential and parallel. It is the term *recursive*. A recursive problem (no matter whether it is a "generative" or "structural" recursive problem) is broken down in smaller instances of the same problem until something like an "initial state" or "base case" is reached from which the problem can be "redeveloped." The recursive programming steps are normally "encapsulated" in a function, for example, and when using such a program later on, the function will then call itself as long as the base case hasn't been reached. A typical and well-known example of such a recursive application is the computation of the Fibonacci numbers with $F_n = F_{n-1} + F_{n-2}$ for n > 1 and $F_0 = 0$ and $F_1 = 1$.

When formally examining such recursive programs or functions one quickly realizes that they are nothing but simple sequential programs. They also process their programming steps one after another; just the "internal program organization" is different from a purely sequential, iterative program. As many people unfortunately mix up these terms and also confuse "recursive" and "parallel," these expressions had been repeated and explained here again. (It should be noted that this thesis does *not* deal with recursive but truly parallel matters and the problems arising with the latter.)

By briefly summarizing the main issues of the previous chapters and sections so far, we have discovered that the natural processes and problem domains are so complex that it is impossible to just simply translate them 1:1 to a computer model, which would resemble a perfect solution to a real-world problem. When trying to copy our real-world scenario into an artificial "world" we encounter big problems. Such an attempt is already impossible in most cases due to continuous transitions in values in nature which cannot be "mapped" onto a computer system. Here we have just bits, either 0 or 1, and with these, discrete numbers but not continuous values can be described. Due to this translation the resulting computer model will not be as accurate and precise as the real-world problem it represents. But when applying utmost care and diligence, the discrete model can closely resemble the original real problem.

The natural and complex processes themselves which take place in the real world and are to be modeled can be expressed in many cases by mathematical equations, like big homogeneous and inhomogeneous differential equations and equation systems in general. When now looking for solutions to such processes which are to be simulated or examined, it would be perfect to find an analytical solution. This is unfortunately rarely possible or simply not feasible. As a consequence, numerical solution techniques are needed to provide a correct or useful solution for modeling such processes. And as the values of the processes or equations normally also produce continuous and not discrete values, the final outcome of such numerical models calculated on a computer is not as precise and accurate as it would be in real life. But here the same is valid as with the discretization: The more careful the modeling using such numerical techniques is performed, the better the result is, and the closer the obtained solution calculated by the computer will be to the real-life solution.

Now with this knowledge in mind we can begin to consider the structure of a sequential modeling program. As we have to process the mesh or grid of our problem domain, there must be some part in the program at the beginning, where the mesh data is read in. Once all the necessary data is "in the computer" somehow, one has to think about the information that is "provided" by the elements, vertices and edges in the model. When all of this has been processed, at some point the mathematical equation systems must be set up to find a numerical solution for the processes that

are to be simulated in the model. Often these equation systems are extremely big and complex, so that it might be a good idea to simplify them somewhat before really computing them in the main and most important part of the modeling program. In the end one has to ensure that the calculated results "come out" of the computer system again.

No matter how sophisticated the real code of such modeling programs finally appears, it is still a "straight-through" processing when we have just one single processing unit, like a desktop computer, for example. With this in mind, Fig. 4.1 can be understood easily, where such a sequential program flow is shown.

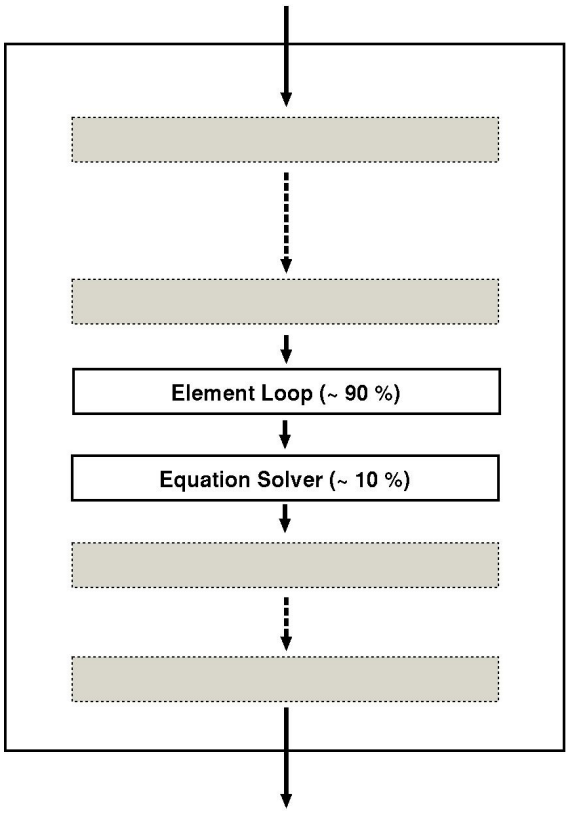


Figure 4.1: Simple sequential program flow of a software package using the finite element method

Fig. 4.1 shows a very simple, sequential program that uses the finite element method to solve its imposed problems. Here, typically elements resulting from a discretization of a real-world problem, as described in chapter 2, are processed (element loop) and, in a following stage, big equation systems, in form of a matrix-vector-product, must be solved (equation solver). The figure actually results from an analysis of the program flow of the GS/RF software package which was introduced previously in section 4.2 on page 71. Although specific and separate input and output operations had to be performed, the total computing time was mainly consumed in the part of the program where the elements had to be processed (with 90 % of the total run time) and the part where the complicated mathematical systems had to be solved (with 10 % of the total run time). Still, the program flow proceeded simply straight through: input processing, then main element processing and equation solving and, at the end, output processing.

Fig. 4.2 depicts the important steps which are normally performed during the main processing phase in a sequential modeling software. In the example shown in Fig. 4.2, the FE method is used for the solution of the model, which means that a transition from the FE presentation into a

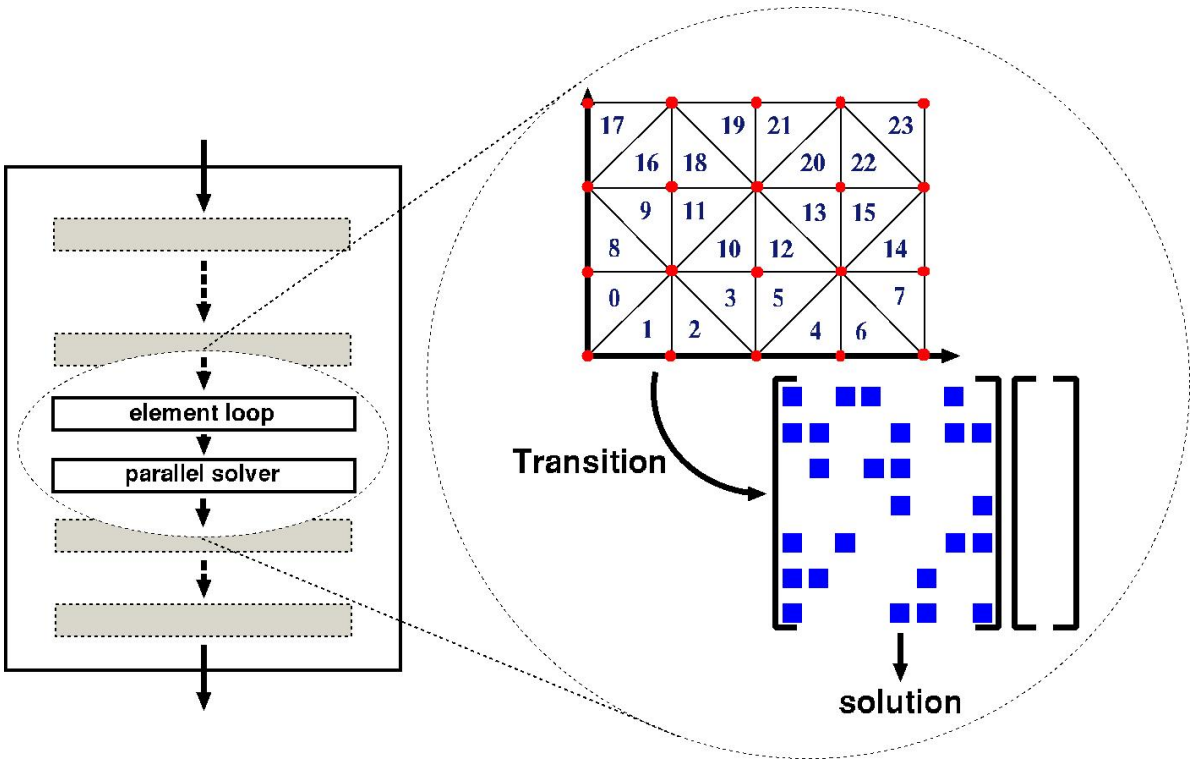


Figure 4.2: Transition from a FE model presentation into a matrix-vector-product in the main processing stages of a sequential modeling application

mathematical representation in form of an equation system has to be carried out (see also chapter 2, starting on page 9).

When investigating typical modeling programs and examining the details more closely, the following main processing stages can be distinguished:

- **Input-** and/or **preprocessing:**
  In this first main program phase generally the discretized models are read for further processing. The common problem at this stage is normally the typical bottleneck caused by the network or peripheral devices. Network connections as well as direct connections of computer components have certain bandwidths. Even if the processing node works with very short clock cycles and is very fast, there is always a "natural" limit imposed by the external devices or the physical network (see also chapter 3 for this). Additionally, the more sophisticated or precise a model is, the more nodes and elements must be regarded. Therefore more information has to be not only stored but also read and processed later on. Often the information is spread among various different files which again impose certain requirements on the underlying operating system and system organization. All open files, memory usage and storage as well as the utilization of the main computing components and devices must be kept track of by the underlying operating system, which costs additional precious computing time as well.

- **Model Computation** or **Equation System Solution:**
  This is the main phase and at the same time the most complex one. Its real structure strongly depends on the methods used to compute the models. They will already slightly vary, for instance, if the finite element, finite difference, finite volume or any other model-solving algorithm is applied. Here, normally the information attached to the elements or the main

model objects (that was read in in the input phase) is processed, the matrix-vector product or the equation system in general is set up and typical modeling algorithms which have been implemented in the software are utilized. Sometimes the amount of data is so vast that certain pre-routines, like a preconditioner, are deployed to accelerate the model computation. Once the resulting equation system is set up by the processing of the elements, for example, only certain algorithmic steps must repeatedly be processed until the desired outcome of the modeling is reached. Unfortunately, this requires a great deal of processing time, depending on how accurate and precise the solution should be. The more exact and better the outcome is, the longer the program run is likely to take.

- **Output-** and/or **Postprocessing:**
  Here, normally a huge amount of data has to be "shoveled" back to the memory or data storage device. And, again, the same restrictions or problems that have already been discussed in the input-processing stage will also arise. The results per element, for instance, must be stored in an output file. Often a special log file is kept and written additionally for analysis purposes, etc.

These three main stages are more or less common to the most modeling programs. Just the internal structure of the modeling code itself will vary immensely depending on the numerical techniques applied and the problem to be solved by this software. There is no "classical" type of modeling program or modeling algorithm, as already the differential equation systems will differ substantially. Can homogeneous differential equations be used? Or rather not? Or maybe a mixture of inhomogeneous and homogeneous partial differential equations? Another example is the fact that although FE, FD or finite volume approaches show many similarities, they are nevertheless three different methods that can be applied for finding a solution to a given problem.

When setting up the equation systems in the software, normally solved as a big matrix-vector product, how are the initial values for the starting vector generated? Are they obtained purely through random values or do they resemble the natural limitations in our system by using the existing boundary values?

There are some algorithms present in which one can obtain a solution to such big equation systems by only "one calculation step." These methods are called *direct methods* like, for example, the classical *Gauss elimination*, the *Cholesky decomposition* or one of the *QR decomposition* methods. In contrast to these aforementioned algorithms, the so-called *iterative methods* repeat certain basic mathematical operations for a mostly larger number of steps until the obtained solution is close enough to a desired result. Here, typical well-known methods are, for example, the *Jacobi method*, the *Gauss-Seidel method* and one of the *relaxation methods*.

And as if matters were not complicated enough, many of these procedures also depend on the structure of the matrices of the big equation systems that have to be solved. Are they regular or irregular matrices? In dependence of the matrix that has to be used, the methods can vary greatly again, as for regular matrices other mathematical operations are possible and therefore different procedures can be applied than for the irregular ones. Typical algorithms for regular matrices are the GMRES method, the BiCG method and the BiCGSTAB method, just to name a few.

In case the matrix is too complex or very unstructured, often so-called *preconditioners* are implemented and applied. By certain conversions and transformational steps one attempts to convert the original matrix into a different but more convenient structure, like a triangular matrix or a diagonal matrix, etc. This transition from one matrix to another of course takes time, but in the end such a matrix with a more simplified structure can be computed much faster and/or with fewer

repetitions than the original non-converted matrix. There are various types of preconditioners, like polynomial preconditioners or spitting associated preconditioners, etc. Some of the known preconditioners are the *uncomplete Cholesky decomposition*, the *incomplete LU decomposition* or the *incomplete Frobenius inverse method*.

The examples above show how complex and unique each single modeling problem can be. As the modeling itself and its related aspects are not the subject of this thesis, we will not dwell upon this topic any longer. But there are many excellent books available on the various issues involved in solving such demanding and complex mathematical problems. A very nice beginner's book for understanding differential equations in regard to boundary value problems, with many examples is, for instance, [Edwards]. For those readers who are rather interested in solving complex equation systems, the book [Meister] is definitely worth a good and thorough look.

Fortunately, some parameters, basic equations or even algorithms of problems that have to be solved in models often do not vary appreciably. Therefore, it is possible to reuse certain program parts in the form of libraries. This way the coding effort can be kept small or at least reduced when possible. The advantage of such libraries or precoded pieces of software is clearly that not only can the programming work itself be reduced but also that such existing functions have often already been tuned and optimized over a considerable amount of time. Sometimes groups of scientists work on improvements of such libraries. A great deal of knowledge is then hidden in these ready-to-use "code fragments" that a single user would normally not be able to accumulate in a short time.

The drawback is, however, that if the problem or the model data or its representation does not comply with the given form of a library or such functions, much effort must be invested either in modifying the existing data, processes or even the model itself until the library can be applied, or in changing the existing code pieces until they fit into the program scheme needed. All of this generally involves considerable programming work, which is then another source of not only logical but also simple syntactic mistakes. And one should not forget that the underlying model and processes should not be changed in their behavior just by adapting the code to an existing library or an already readily implemented algorithm.

4.4 Transition to a Parallel Modeling Program

Up to this point we have always assumed that the underlying computer architecture which is used for the computation of the models was just a simple one with only one processing unit, similar to a normal desktop machine with one CPU. As already indicated throughout the previous chapters and sections, the core of such sophisticated modern models of real-world problems is a big equation system that finally has to be solved by using the computer.

When remembering the suitable example of the dominos in the section above it easily becomes clear why multiprocessor environments can save much time, especially when computing such highly complex models. In the domino scenario, double the amount of dominos can be pushed over in a certain time when two lines instead of one are set up. Unfortunately computing real-world models is not so easy as just pushing over dominos, but when the equation system is finally set up at one point in the modeling process, everything else simply boils down to a question of pure computing power in the end. The more processing units available in a specially adapted hardware environment, the larger the number of equations that can be computed at the same time. (A "reminder" in regard to the problems and benefits of parallelization can also be found in chapter 3

when recalling the examples presented there of our friends Peter and his employee Paul with his other digging collegues).

The biggest problem one encounters when switching from a sequential modeling program version to a parallel one is *how* to "tear apart" the model itself. Fig. 2.17 on page 22 in chapter 2 quickly makes clear why. The whole complex model is represented by a big equation system, mostly a matrix-vector product. But how can one now "chop" the original model into suitable handy pieces so that each processing unit can work on without affecting the underlying equation system? Furthermore, it seems impossible to just cut this mathematical system into little subsystems without consideration of the original model. It appears that the solution for the problem depicted in Fig. 4.3 is one of the typical chicken-and-egg problems.

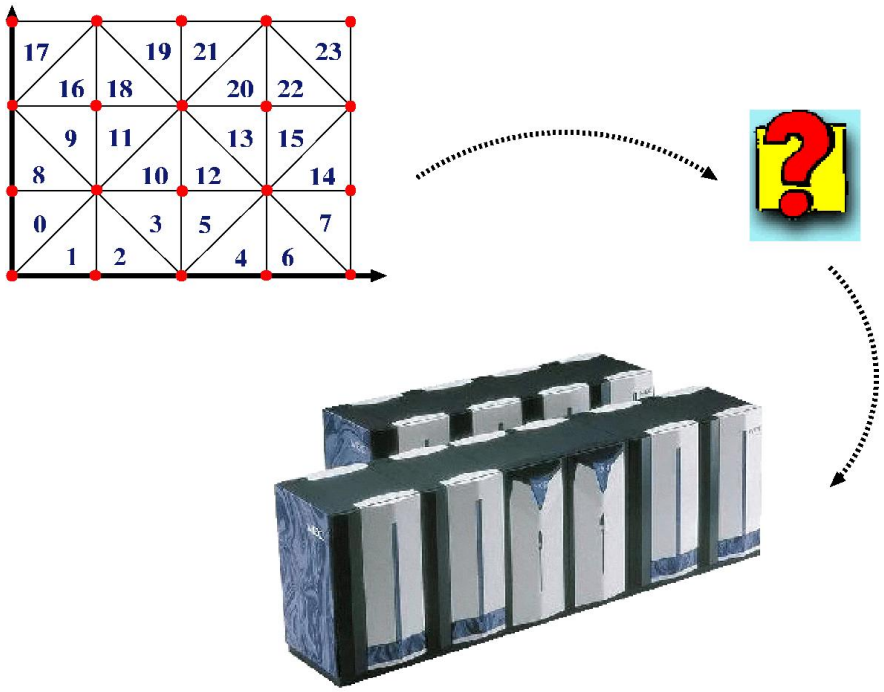


Figure 4.3: What steps are necessary to compute a finite element model on a supercomputer?

Fortunately, numerous researchers and scientists have worked on this challenging question and in the meantime very good heuristic algorithms have been developed that already produce suitable partitions to such problems. As this topic needs to be elaborated upon, it will be dealt with in a separate chapter about *domain composition* and *load balancing* following this one. For the sake of simplification we assume now until the end of this section that we have the problem of subdividing such models already conveniently solved. Furthermore, another assumption is that we are also lucky to have a powerful parallel computer handy for our future modeling work. This means we are now able to break down our original real-world model into smaller subdomains which then can be easily spread over different computing nodes of our parallel computer. This process is shown in Fig. 4.4.

When now "zooming" in on our main processing stage of the overall modeling computation process, and with the above assumptions, we in fact encounter the situation depicted in Fig. 4.5.

In a purely sequential program flow the equation system is generally just one big matrix-vector product which is then computed step by step by the single processing unit. But here, in a mul-

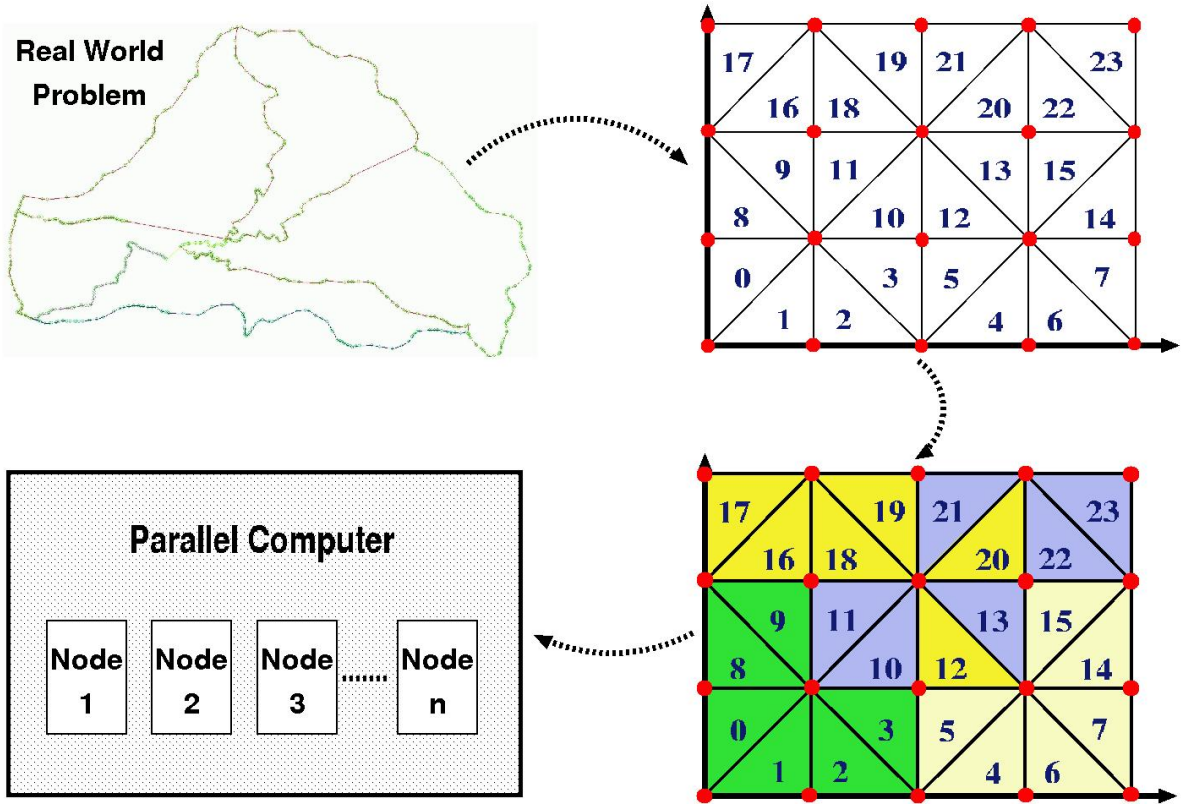


Figure 4.4: The simplified steps from a real-world model to its computation on a parallel computer

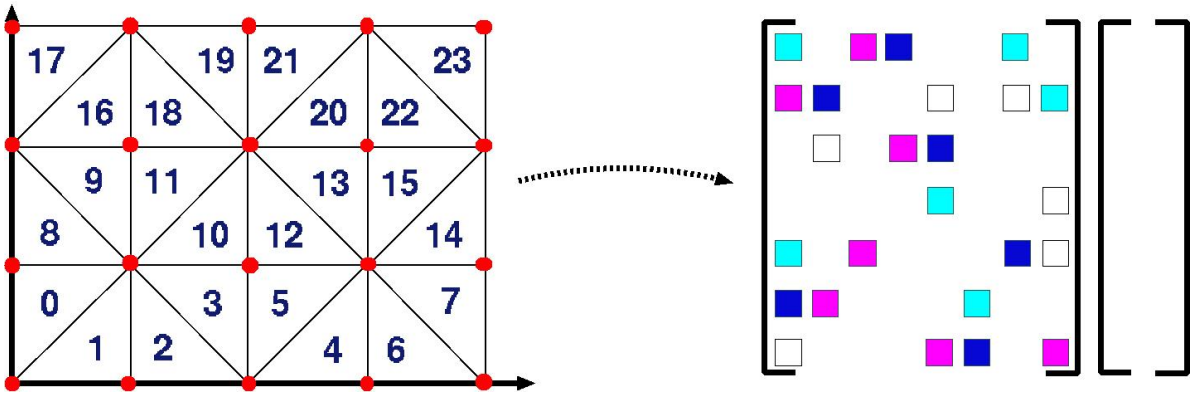


Figure 4.5: From a mathematical representation of the original real-world problem to a matrix-vector product with elements of the subdomains of the overall problem

tiprocessor environment, the elements of the subdomains of the original model are distributed throughout the whole equation system, as shown in Fig. 4.5. By obeying certain basic mathematical rules the overall matrix-vector product itself can now be broken down into smaller subsystems. Finally, each of them can then be processed and solved on a single processing node. This whole process is shown in simplified form in Fig. 4.6. (What important steps or processes are hidden behind the question mark will be discussed in the upcoming chapter).

At this point the reader most likely will lean back and relax, thinking to him- or herself that the parallelization process doesn't seem too complicated at all. Unfortunately this is not the case. Not only the handling and treatment of the model itself and its underlying data have to be done quite

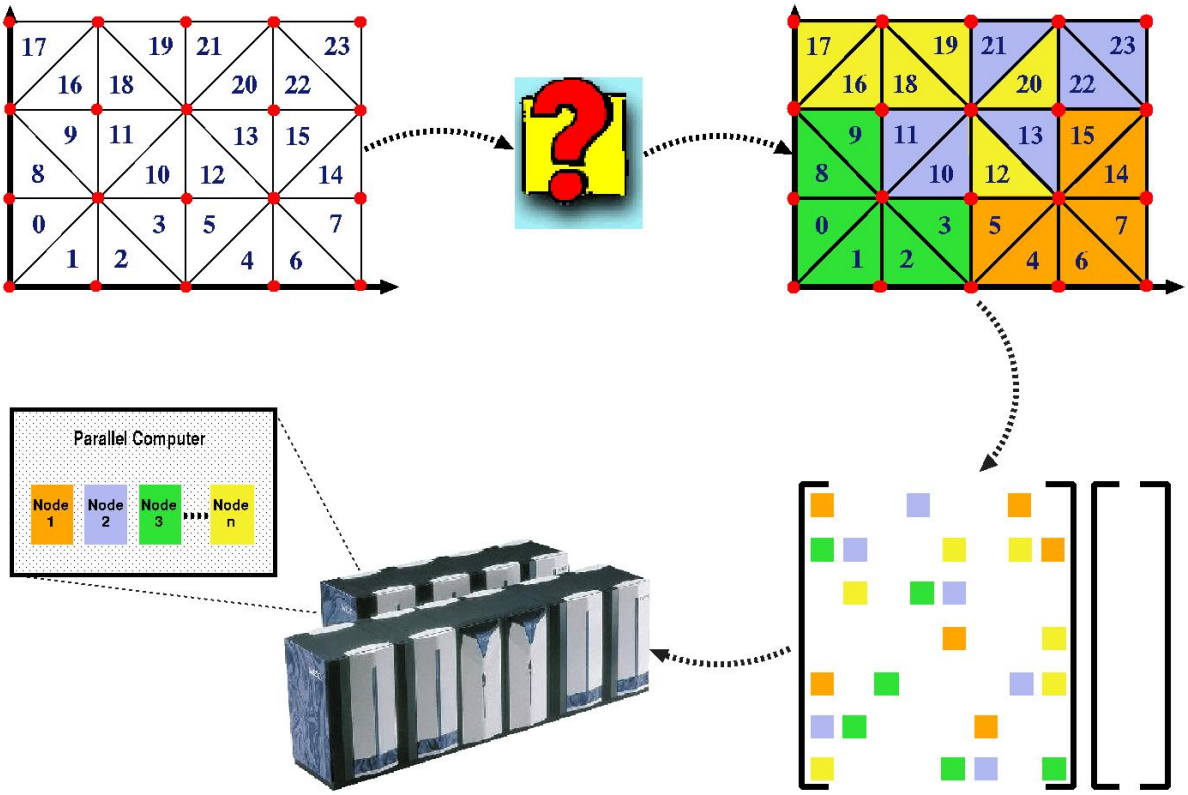


Figure 4.6: The main steps in the process translating a mathematical representation of a model into a form that can be processed on a parallel computer

thoroughly and carefully, but the remaining parts of the modeling software need equal attention.

Normally, when using a very expensive multiprocessor environment, special peripheral hardware is available. The typical network bottlenecks or problems with the peripheral devices are not as obvious, as they typically occur with an out-of-the-box desktop computer, for example. But still, even in such a multiprocessor environment, certain limitations are still present. Therefore, one also has to consider here, of course, the network traffic, the communication between the processing nodes and the peripheral devices, just to name a few important issues.

Typically lean data structures should be used in general. This is often a step backward in regard to programming issues, as suddenly very elegant and "readable" data structures used on single processor machines are no longer applicable in a multiprocessor environment without losing processing time or imposing additional communication between the processing nodes. Furthermore, already existing parallel code libraries can also be deployed, but then the code has to be adapted for their usage. The extra programming work usually pays off in shorter runtime. Many standard algorithms do exist in a purely sequential version but also in a specialized parallel version which should then be used if possible and applicable.

In some cases it can be beneficial if the programmer or engineer considers the hardware issues while coding. A typical example is, for instance, the so-called *loop unrolling* which a standard programmer would hardly ever apply on a simple desktop PC but which is a typical technique adopted in a multiprocessor environment. This is but one of a number of examples, and once again, many of the details depend strongly on the modeling application and its processes, the underlying algorithms and computer architecture, etc. In addition, such factors as time, money and speed in producing usable outcomes and solutions to the models should not be forgotten. Even if

very experienced programmers work on the parallelization of a software, they might not be very productive and successful if they have just a small budget and hardly any coding time.

Therefore, the improvements and changes in a parallel code in comparison to a sequential code for the same problem can extend to all magnitudes. The more trained and skilled the people involved are who work on such a demanding parallelization project under good working conditions in regard to money and time, etc., the better the resulting code and its adaptability to the computing environment will be and thus, the quicker and hopefully more precise and accurate the results of the modeling process.

It should also be clear at this point that a thorough and effective analysis of the problem or model proper and its parallel implementation is only one of the important aspects of the parallization process that affect the final outcome. There is never a typical "standard parallel version" of a problem or a "typical procedure." Furthermore, also recalling the different types of parallelization in chapter 3, one should always keep in mind that it is possible to parallelize the whole program right from the very beginning of the software code, for instance, starting with the input phase or only the most important parts in the main processing stage of the program, like the solver or the element loop. The various aspects of the different parallelization approaches again depend on the model itself and the different issues in the whole parallelization process. For reasons of simplicity, in the following we are going to assume that just the main processing part of the software is to be parallelized, as in the majority of cases, this is where the most runtime is consumed.

When returning to the differences between a purely sequential modeling software and a parallel software version, Fig. 4.2 depicted the important issues in the main part of a sequential modeling program. When now recalling the previously discussed fact that in a parallel version the matrix-vector product that results from a big equation system has to be subdivided for the distribution on the various processing nodes, it is immediately clear that important changes must be considered in this main program part.

This is demonstrated in Fig. 4.7. The program itself still can run sequentially, but the very moment we enter the main processing stage, the original, big matrix-vector-system is now split up into several smaller ones which can now be computed in parallel on the different processing nodes. In the end, one has to make sure that each of the results is gathered again, and a final, resulting big solution vector must be computed. A practical example of such a solution in a real modeling program can be seen in Fig. 4.8.

MPI was already briefly introduced in section 3.9.1 of chapter 3, starting on page 57. In the example given there, it is used to implement a first but very simple parallelization within the main processing stage of the GS/RF modeling program. Using the MPI commands which are placed carefully and with consideration within the original sequential code, the smaller matrix-vector products are computed all in parallel on the different processing nodes of the multiprocessor machine and ultimately, the sub-results are merged into one big final result vector. What is not shown in the figure but definitely has to come first is the subdivision of the original big matrix-vector product with which the whole computation starts out. (We had assumed for simplicity reasons that the problem of subdividing it had already been successfully solved.)

In Fig. 4.9 we have now "zoomed" into the oval on the right of Fig. 4.8. For the simple parallelization approach using MPI we start out with the FE presentation of the model, as GS/RF uses the FE method to compute the results of a given model (see also section 4.2). The original FE model presentation has been subdivided into smaller submodels, which is demonstrated by the usage of the different colors. Each of the smaller chunks of the FE presentation is deployed to set up an equation system that is only valid for this corresponding submodel.

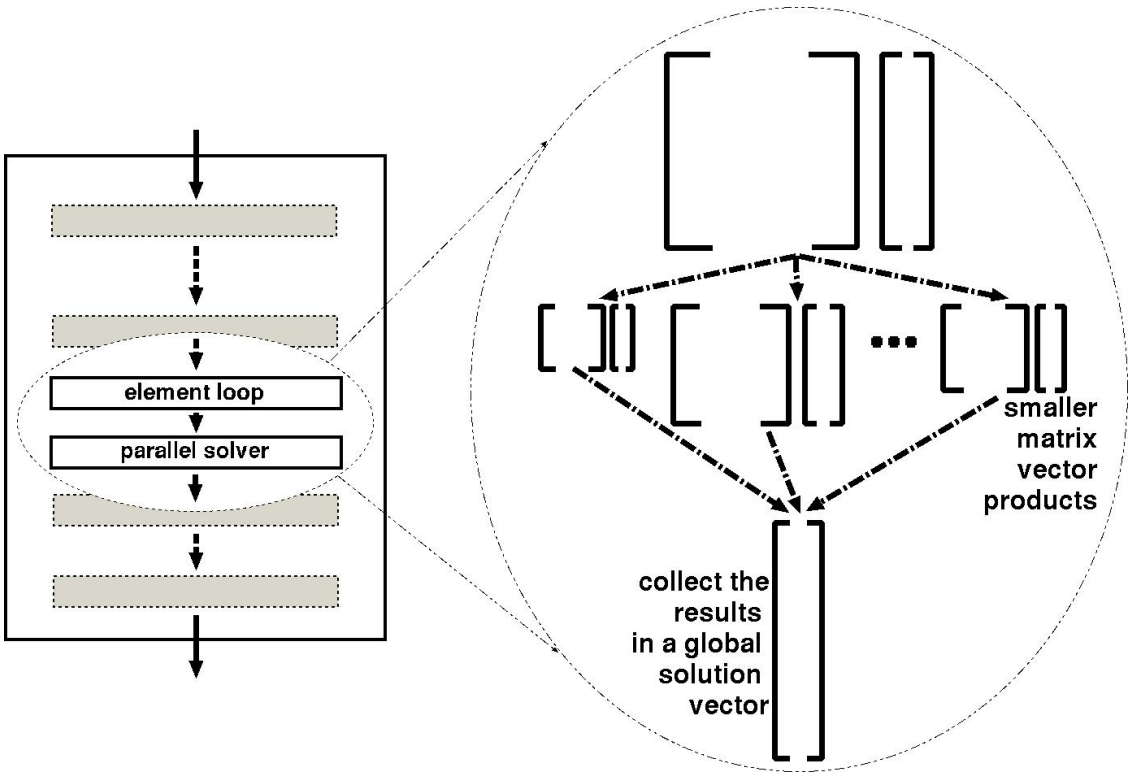


Figure 4.7: Zooming into an originally sequential modeling software which main processing part is be parallelized

As the computational steps for the solution of such a matrix-vector system are always the same, the calculation of all of the smaller equation systems can now be distributed over various MPI processes to the available computing nodes of the multiprocessor. Each of the equation systems is processed independently from the others this way. In the end the final outcome of each of the matrix-vector products is brought together and merged into one big resulting vector with the MPI command `MPI_All_Reduce`. This is also shown in Fig. 4.10.

When regarding Fig. 4.10 the reader should not be deceived by the regularity of the "slices" of the MPI processes which are depicted inside the upper global matrix-vector product. It is just shown this way in the figure for simplicity reasons. In reality the values within the matrix belonging to one single MPI process and used for setting up the smaller equation systems can be spread out all over the big original matrix.

Fig. 4.11 summarizes the most important steps when parallelizing the main part of a modeling program, including a small improvement. The original matrix, with which one starts out, holds all values of the complete model. For the parallelization this global matrix-vector product must be "torn apart" so that it can be subdivided and distributed on the various processing nodes. The decomposition of the overall matrix results in independent processes; in the example shown in Fig. 4.11, MPI processes have been chosen. Each of these processes can then be executed and computed independently on one of the available nodes of the multiprocessor. In a simple first implementation the smaller matrix-vector products work on the global solution vector. But it is also possible in a more advanced version for the local, smaller matrix of the subequation system to work with a local, and therefore also reduced solution vector. In the latter case one has to make sure that the results, determined this way, are merged into a final but global solution vector in a mathematically correct way.

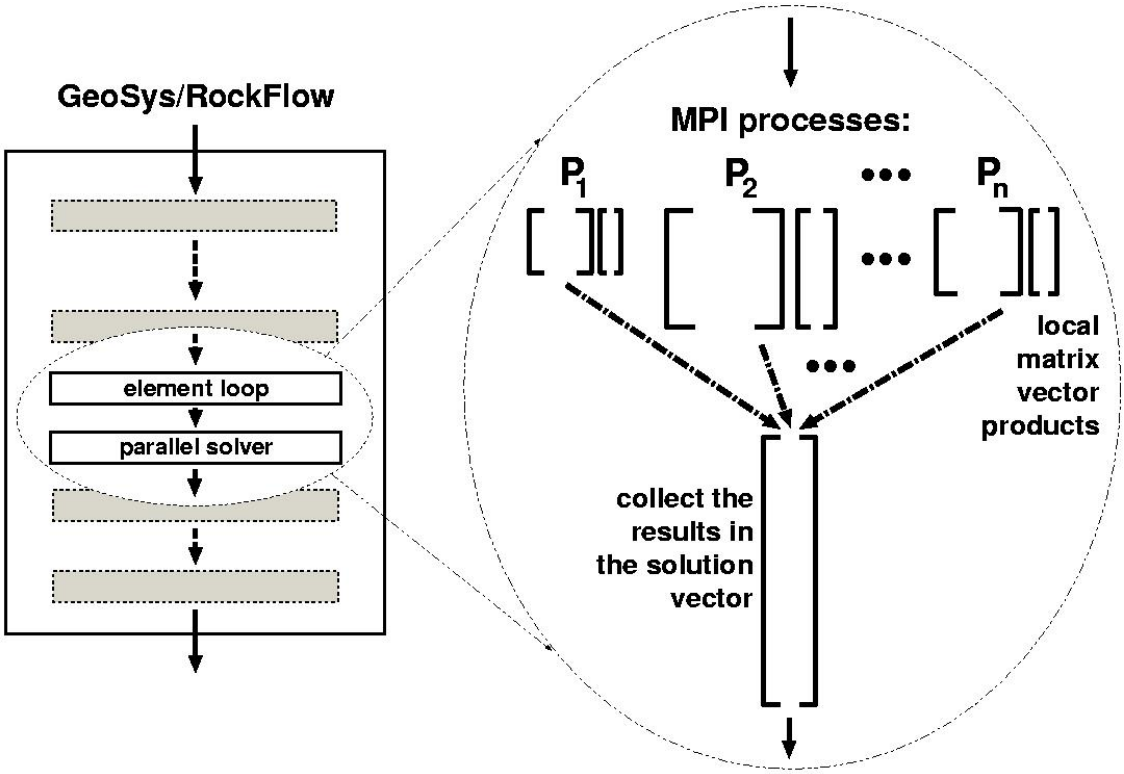


Figure 4.8: Simple parallelization using MPI in the main stage of the GS/RF modeling software

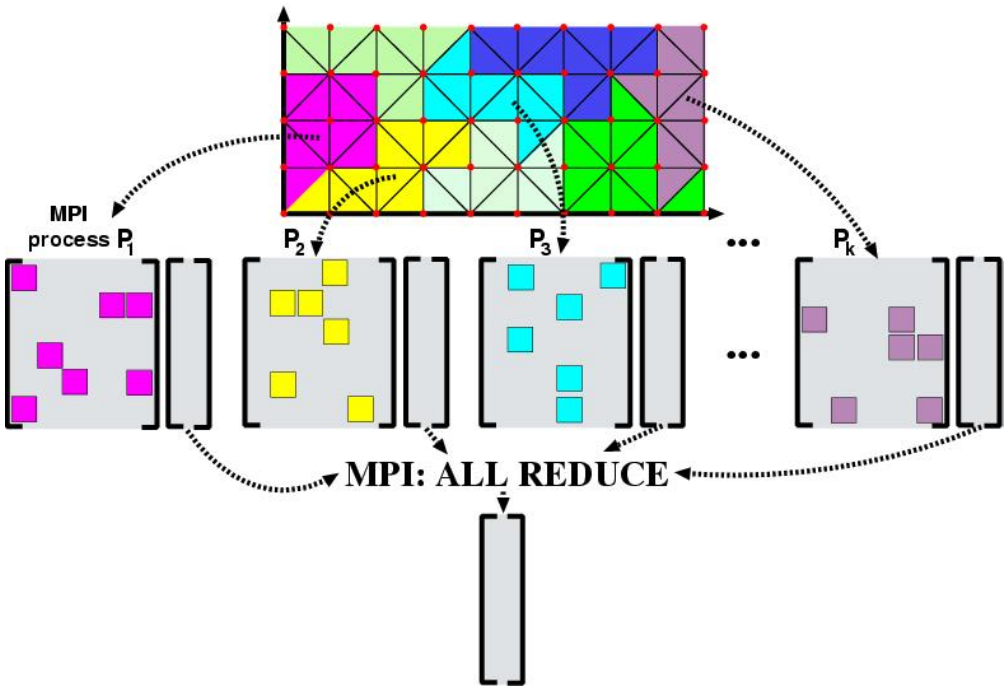


Figure 4.9: Matrix-vector products resulting from a subdivision of a FE presentation of a model are computed as independent MPI processes and their result vectors are merged into one final one using the MPI command `MPI_All_Reduce`

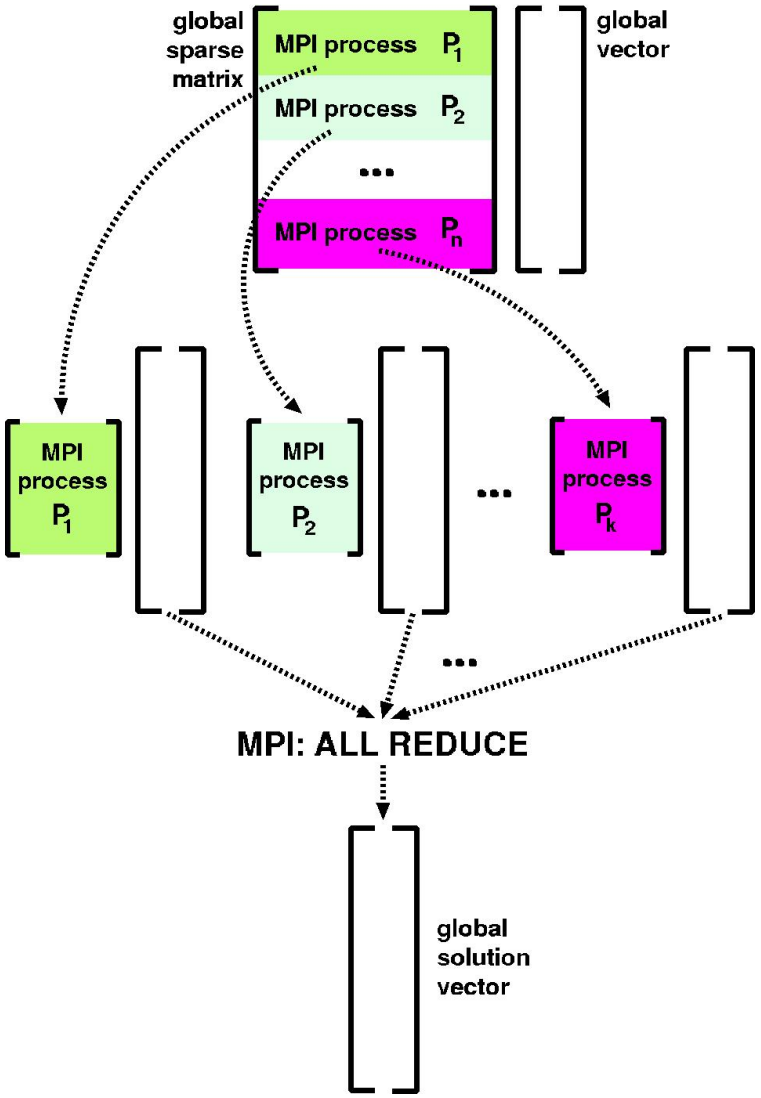


Figure 4.10: A matrix-vector product is subdivided into different smaller subsystems, each constituting one single MPI process. The final result is merged from the different solution vectors of the smaller matrix-vector products.

4.5 Summary

This chapter has dealt mainly with various aspects of the modeling itself. The reasons why modeling generally is necessary and what typical steps or rules should be followed when setting up a scientific model have been introduced. The structure of a scientific modeling software which might result from such a procedure has been discussed and the general composition observed in an exemplary modeling software called GS/RF. When transforming a purely sequentially organized modeling program into a version that works in parallel, certain important aspects have to be considered. The biggest changes will be encountered in the main program part where the complex equation systems are set up and calculated later on.

When now looking back from this point in the text to the very beginning of the thesis, the reader will have noticed a slow but constant approach from a very global point of view to a more specialized one. We started out with the general problem of the discretization of a given real-world problem as well as a quite broad overview of different computer-science-related topics in regard

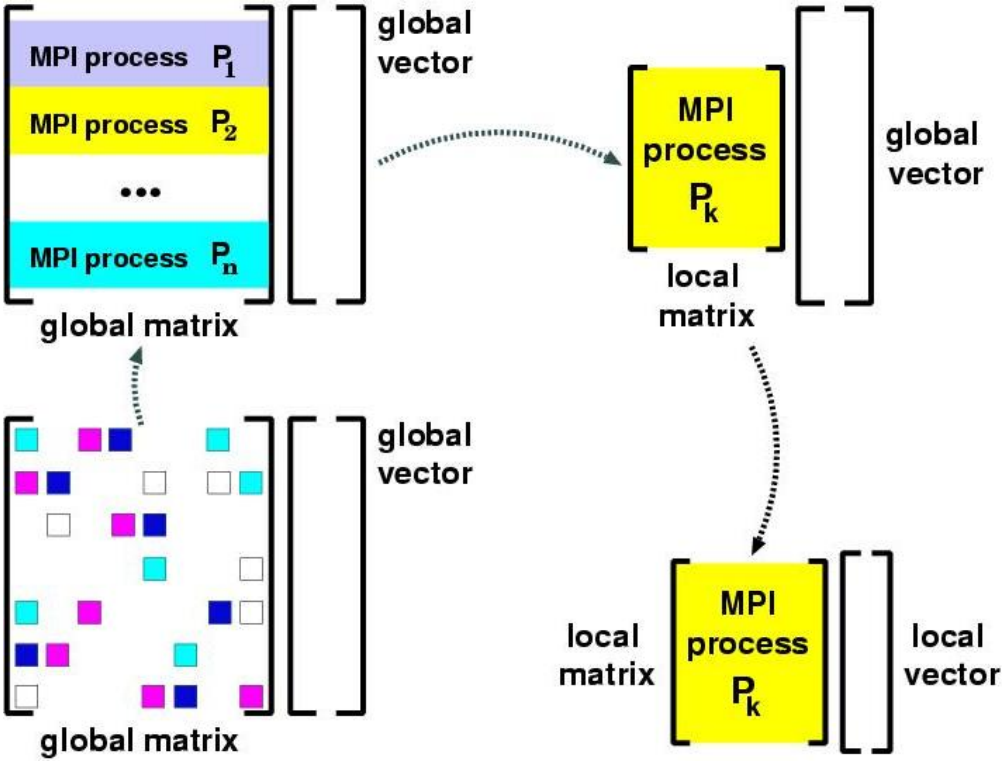


Figure 4.11: Simplified overview of how to break down a global matrix-vector-product into different independent subsystems for a simple MPI parallelization approach

to parallelization. This chapter has now focused on more advanced aspects of computer modeling and encountered problems when converting an originally sequential program into one that operates in parallel.

The following chapter will continue in this way, and now very specialized issues in regard to load balancing and domain composition will be discussed in more detail. Previously in the current chapter we had assumed for reasons of simplicity that the subdivision of the model data had been already accomplished. The upcoming chapter will now “dive” into more details of this interesting and complex problem.

Chapter 5

Domain Decomposition and Load Balancing

This chapter starts off with an introduction as to why data preprocessing is so important in regard to load balancing and domain decomposition. It is followed by a very brief glimpse into the theory behind these two aspects. The third section explains the algorithms and processes which are used in this program of the thesis software. As this C program itself also falls back at certain stages on other C routines and libraries, but which is hidden from the user, it is called a *wrapper*. For the user just one piece of software exists and is visible. In reality it is a collection of different functions, routines, etc., which are all "wrapped up" in this overall program which is executed by the user. For this reason, the code that was implemented during the project of the thesis is called wrapper (software).

At the end of this chapter, two of the most important packages in this field, Metis and Jostle, are introduced and their deployment in the wrapper software is explained. The chapter is concluded by a short comparison of the two packages and indications of future improvements and optimizations of the wrapper.

5.1 The Importance of Load Balancing and Domain Decomposition

As could already be seen from the introductory chapters of this thesis, parallelization of an existing program or software package means tearing it virtually apart into single pieces like data input, computation steps, data processing, just to name a few, and to check everywhere for optimization potential, communication aspects, storage issues, etc. The best parallel algorithms implemented in a piece of software are useless if the majority of the computing nodes is idle while a few of them work themselves to death, or if the communication between the single processor nodes increases to a level where it takes more time to wait, send and process (status) messages between themselves than for them to actually work and compute.

Certain issues which involve, for example, hardware aspects, like the famous bottleneck of input and output processing, cannot be optimized at a certain point by the programmer or software engineer. This is a given fact imposed by the architecture or structure of the supercomputer or parallel machine. But the efficiency or the utilization of the various processing nodes, their single workload and the communication rate are all affected by the way the program is structured and developed.

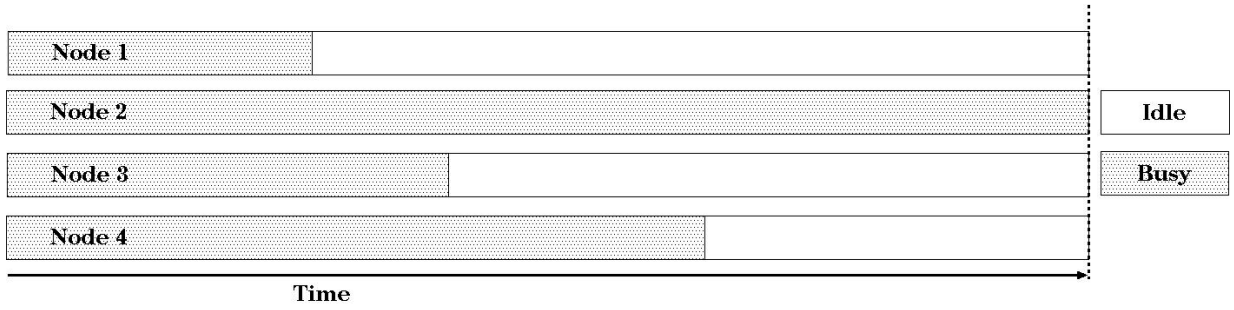


Figure 5.1: Highly differing computation times of 4 processing nodes

Let us take, for example, Fig. 5.1, where the computing times of 4 processing nodes is shown. Node no. 4 needs most of the time while the other 3 are waiting for it to finish its job. If it is possible to distribute the workload evenly (which is not always feasible) or as evenly as possible, the other remaining three nodes would have to work longer than they do now. But as all four of them compute simultaneously and in parallel, the whole job itself would be completed sooner than it is now.

In the example shown in Fig. 5.1 we have just four nodes and it seems that the time saving is minimal. But as shown in chapter 3, modern supercomputers and number crunchers have hundreds or even thousands of processing nodes. If they all work efficiently together, the amount of time saved is much greater in comparison to a sequential program, when the software is perfectly suited to the architecture of the machine in question.

Taking the above problem and an appropriate example with Peter and Paul, let us assume that the ground the laborers have to dig in is not 100 percent equal in its consistency and composition. There might be areas where the ground is rather rocky and hard and others where it is a bit more sandy and contains more clay, thus being much softer to dig in.

Peter now has two options. Firstly, he could just assign every digger the same-sized area to dig through. This means that some of the workers will be done rather quickly, while others might still have a hard time to get through the rocks and the hard ground. We assume further that there is the rule that the laborers always have to leave the digging ground together at the end of their workday. The diggers of the softer areas, having finished earlier, have to wait in the worst case maybe for hours, until the last one, most likely digging in the hard ground, finally finishes his job.

This situation would correspond 100 % to Fig. 5.1. Here the laborer, working on solid and rocky ground, would be computing node no. 4, whereas the other diggers, working on much softer ground, are represented by nodes no. 1 to 3. In this case laborer/node no. 1 needs just half the time of the slowest one, no. 4, and also no. 3 is done rather quickly in comparison to no. 2 and no. 4. In the example with Peter and his diggers, this situation, as shown in Fig. 5.1, would be an extreme waste of work power due to mismanagement or misorganization, which every employer would try to avoid. This is the same with programming a multiprocessor computer. Every idle processing node here is also a waste of resources, time and money.

The other option that Peter has could be that he takes a closer look at the ground and its material before the start of the digging job and then assigns unevenly divided, but differently sized areas to each of his employees. The more rocky areas with the harder material will be smaller than those with softer ground.

As Peter is an experienced employer and knows the average workload that a single digger can handle, he will roughly calculate how much time one person would need on the average for his

single piece of ground. Of course, not everybody works in the same way; one digger might be stronger than another or one might be having a bad day, feeling a little sick and therefore being somewhat slower than a healthy one, etc., but Peter's approach helps to evenly divide the work among his employees. The ones who have the hard ground and the rocks might need more breaks as it is more exhausting to dig through such material than just shoveling sand.

In the second case, the laborers who have a sandy but therefore bigger area to dig in might work longer in comparison to the first scenario, but in the end all laborers will finish earlier all together in the second scenario than they would in the first one. And this is exactly what one also would like to accomplish using a supercomputer. On the average it should complete the entire computation much faster than a non-specialized machine.

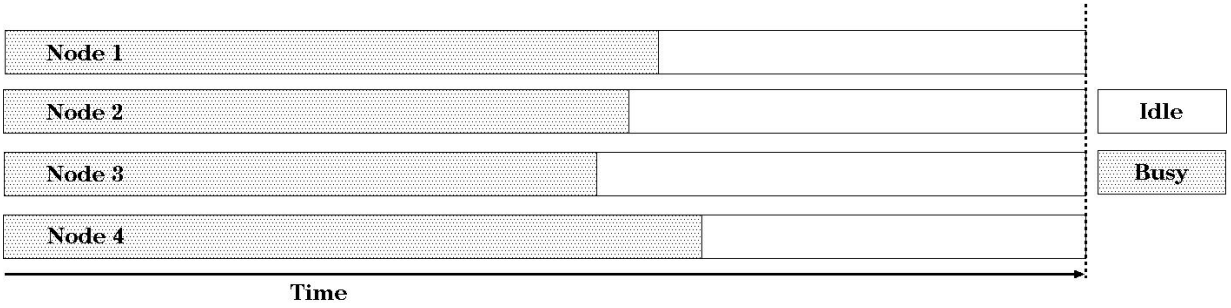


Figure 5.2: Quite similar computation times of 4 processing nodes

This is shown in Fig. 5.2, where we again have 4 different processing nodes. Here the programmer has provided for an almost even distribution of the workloads to the different computing nodes. Referring to scenario 2 with Peter and Paul, above, nodes no. 2 and no. 4 might resemble the work area of the diggers who have to dig in the rather rocky areas; or it could also be that the sandy areas are somewhat larger than Peter had thought, so that the sand diggers need more time in the end than their rock-digging colleagues, although they have the easier terrain. Or, third option, the laborers of areas no. 2 and no. 4 are not in their normal physical shape, and are therefore somewhat slower than their employer, Peter, had expected them to be.

Whatever the reason is for the slowdown of nodes no. 2 and no. 4, it is not important in the end. What really counts for the final result is how much computing or digging time could be saved by distributing the work optimally for the specific situation. There is hardly any chance for a programmer or even Peter to assign the workload in a perfect way. But the closer the distribution is to an almost perfect subdivision of the overall work that has to be done, the best result in regard to time saving can be achieved.

Previously, communication between the computing nodes was also mentioned as a potential problem for slowing down the processing. We can again take our Peter and Paul example here to exemplify this. Let us assume the first of the two scenarios above, where every digger is assigned the same size of terrain to work on, regardless of the material. We still have the rule that the laborers are only allowed to leave the area together when the work is done.

As we have already seen, the diggers working on softer ground are able to dig much faster and more easily than those working on rocky and hard ground. Let us now imagine the sand diggers being already done, whereas the rock diggers are still working in their area. The latter might see that the sand diggers are done and would start shouting at them something like, "Hey, come on over and help us. Together we're faster and we can all leave earlier tonight." Those on the sandy ground might not understand exactly, as it is too noisy, and probably would call back to see what

their colleagues want of them. Now the laborers on the rocky ground have to cease their digging again and maybe even walk over to the sandy area to be better understood. Most likely in the end the sand diggers will catch on to the idea of the rock diggers and help them, so that they all can leave earlier. But just by arguing, discussing and communicating they have already lost a lot of time which, when well organized, could have already been used for digging and working. They still might be able to finish their jobs much earlier than they would have done by just sticking to scenario one above.

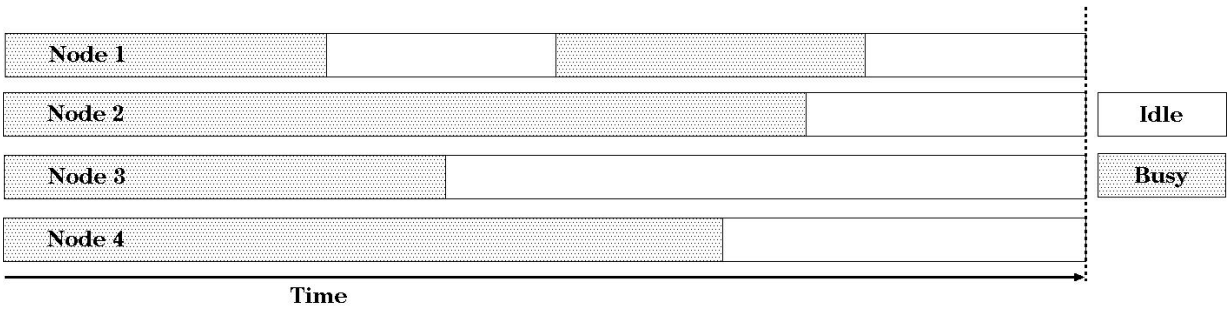


Figure 5.3: Quite similar computation times of 4 processing nodes, but with an interruption of the computation/work process at node no. 1

This can easily be seen just by comparing Fig. 5.2 with Fig. 5.3. In neither picture has the workload and computation time of node no. 2 and no. 3 changed at all. Let us assume that node no. 4 in Fig. 5.3 represents the rock digger of scenario 1, and node no. 1 symbolizes the sand digger. The latter already loses precious time just due to communication. In Fig. 5.3 the laborer of the rocky area doesn't cease his work and continues, simultaneously communicating with his colleague on the sandy ground. No matter how fast they agree on how to split the remaining work of the rock laborer, they have already lost time due to their communication and thus, they will definitely finish a good deal later than in the second scenario above, depicted in Fig. 5.2.

All of this is also directly comparable to our parallel machine. Even if the workload is divided in a good manner among the different processing nodes, and even if each of them can work undisturbed on its piece of code, the very second they have to stop to start communicating they are directly losing time. At least one of the nodes has to start the communication; maybe it then has to wait until the recipient finishes its job or reaches a good point to interrupt its work. Then they are forced by protocol to maybe acknowledge the message or even share information or intermediate results. This depends also on the programming paradigm which is used and on the programming language and its parallel extensions. Not all of them have good communication methods implemented. But with an already well-organized program and structure before the programming phase itself, the extent of communication can be reduced to the necessary minimum.

If just two of the processors were involved in a communication, this deceleration would hardly be noticeable with regard to computation time. But if all of the nodes needed to communicate with each other or had to send results or data once in a while, this definitely would slow down computation in a notable way.

Let us have a look at Fig. 5.4, which we already are familiar with from section 3.10.2 in chapter 3. It shows an activity plot of 8 different processing nodes which are executing a parallel program. The plot shows the times when the nodes are actually really computing/working on something, when they are occupied by communication and when they are idle.

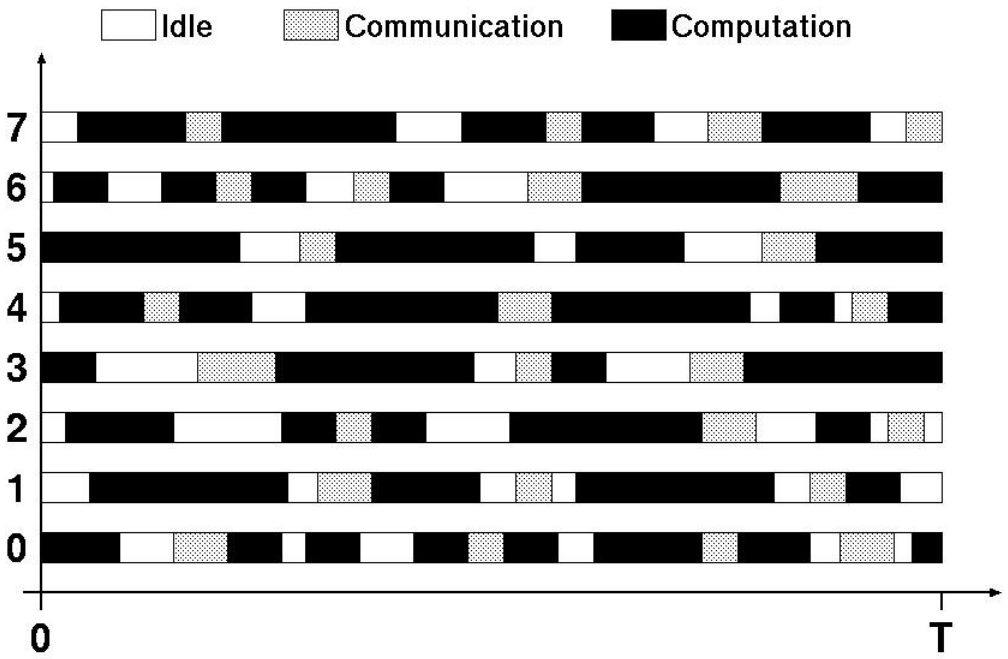


Figure 5.4: Activity plot of a parallel program during execution on 8 processors

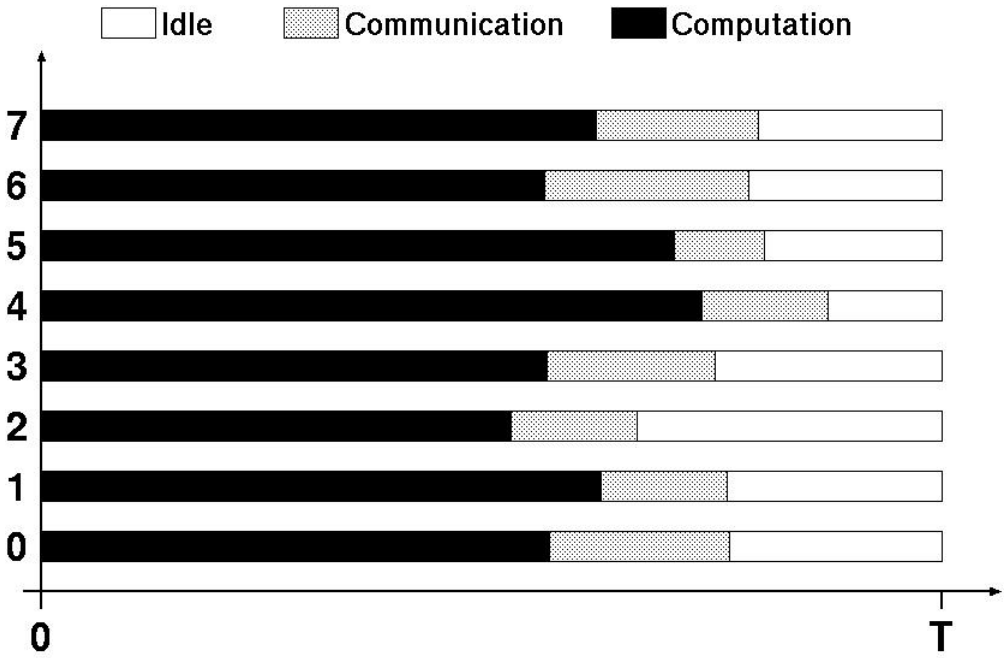


Figure 5.5: Accumulation of the computation, execution and idle times during an execution of a parallel program running on 8 processor nodes

Now we consider Fig. 5.5. Here we have the same 8 nodes again, still working on the same program in parallel. The only difference to Fig. 5.4 is that the small time slots for computation, communication and for the idle times are now accumulated. Already in Fig. 5.4 one can easily see that node no. 2, for example, doesn't have as much black in its activity column as the other ones and that node no. 4 seems to be rather busy in comparison. But, when now checking the accumulated times for the nodes, the result is quite deflating.

Although node no. 4 is really the busiest one among all 8 nodes, it still wastes about a fourth of

the overall available time on communication or simply doing nothing instead of spending that time computing. Taking node no. 2, matters are worse: It just works about half of its precious time and the second half is simply wasted on communication or idle time. The overall picture of all 8 nodes shows that every processor spends about a fifth of the total time completely inactive or dormant.

With this pseudo-realistic example of the activity plot in Fig. 5.4 and its revealing results due to the accumulation shown in Fig. 5.5, every good programmer and software engineer would sit down and try to minimize this great amount of idle time and also reduce the communication times as much as possible. Sometimes, for example, a better algorithm, improved data handling or a good restructuring of the program code could help. But there are also cases where an activity plot like the one shown in Fig. 5.4 could be the best achievable one, using all existing tools, algorithms, improvements and knowledge available. It all depends on the situation and the project which is to be parallelized.

Generally, problems that are going to be parallelized are normally quite complicated to compute or need challenging mathematical equations and algorithms for their solution, for which average desktop machines or out-of-the-box computers do not offer the suitable hardware. Often big equation systems or matrices have to be solved. The mathematical expressions used normally depend on each other or share variables or intermediate solutions. As already discussed in the previous chapters, one can't just sit down and simply "chop" these equation systems or matrices into single suitable pieces like a cream cake. This would directly lead to unusable results or completely false equation solutions and iterations of the problem.

The former chapters already showed that when processing numerical problems on a simple desktop system or a normal "out-of-the-box" computer, the responsible programmer or software engineer has to know the correct "translation" between the necessary steps, beginning with the real-world problem, until he or she finally ends up with the corresponding matrix-vector product of the given model or problem. This equation system then can be broken down into code and solved when running the ready software, as shown previously in Fig. 2.18.

Now, when using, for instance, a supercomputer with more than one CPU or even just a small cluster with a handful of processing nodes, it was discussed earlier (see, e.g., section 4.4) that just translating every single step in the conversion process from the real-world problem to a matrix-vector product to be run on a specialized machine isn't so easy anymore.

As already shown at the beginning of this section, it is absolutely necessary to distribute equally sized subdomains (if feasible) on the available processing nodes. On the other hand chapters 2 and 4 should have also made it clear that it is not possible to simply cut the original real-world problem into "handy pieces" and just distribute them on the computer workhorses.

No matter whether we are working with a standard computer or a parallel machine, in both cases we start out with our real-world problem. On the way from this model to its final representation in the program code, the translation process in between has to be slightly changed when more than one processing node is used later on for the solution of the equation system.

As seen in Fig. 2.17 on page 22, the mathematical representation of the original model corresponds to a specific matrix-vector product. But now having different and smaller subdomains (see Fig. 5.6) which are necessary in case we are working on a parallel computer rather than just one big model used when one processor is involved, we run into the next major problem: How can we now tear apart the matrix-vector product? Or, rephrasing the question: How do we now start out with the situation depicted on the left-hand side of Fig. 4.5 on page 81, to end up with the desired

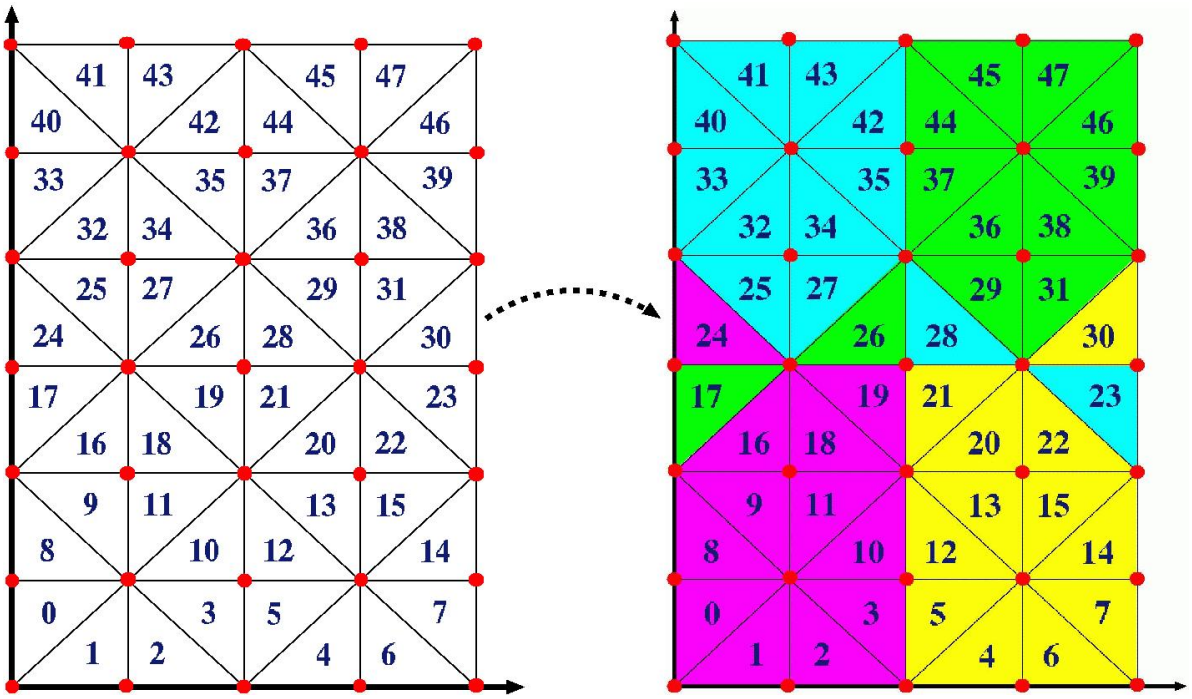


Figure 5.6: A FE presentation of a model and one, exemplary subdivision of it into several smaller subdomains

appearance of the matrix in our matrix-vector product shown on the right-hand side of that figure with a still valid equation system?

We had already run into this problem in chapter 4 at the beginning of section 4.4 on page 80, where we had assumed for reasons of simplification that some algorithm or method exists with which we can slice down our original big model into suitable smaller pieces.

Although this imposed problem may not sound too tricky or complicated right now, we should by no means forget in this situation that the subdomains should be either of the same size or that the subproblem size should have the same computation time. And last but not least, as a further restriction, the communication between the participating processors should also be kept to a minimum, as discussed and shown at the beginning of this section.

Even a non-computer scientist will see at this point that the solution to the complicated problem introduced above is not so easy after all. Now, where shall we cut the model in correctly sized pieces, taking into account all the other important aspects?

If we look at Fig. 5.7, we see a simple finite element model consisting of 20 nodes. Assuming we already had a solution to this intricate problem on how to cut such a model in the optimal way, we decide now to subdivide this model into 4 equally sized submodels. It turns out further that node no. 7 would be exactly on the line where we want to cut. To which of the 4 submodels should this node now be counted, after we have taken apart the original model as demonstrated in Fig. 5.8?

The same problem arises when having a look at, e.g., node no. 6 or no. 12. The only difference to node no. 7 is that here they are shared by just two subdomains instead of four. The importance of how to treat such border nodes becomes even clearer when studying Fig. 5.9 a bit more closely.

This means that when subdividing a given model, such boundary nodes must be treated with utmost care and have to be tagged so that they can be easily recognized later on in the list of all nodes of the model.

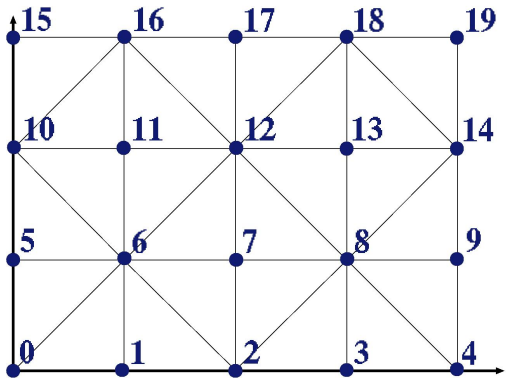


Figure 5.7: A simple finite element model consisting of 20 nodes

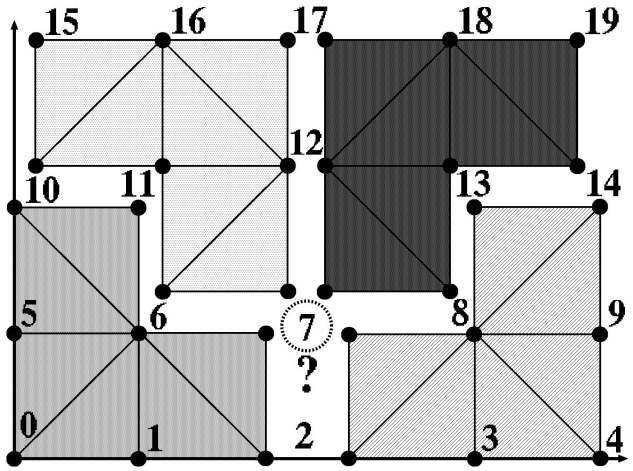


Figure 5.8: Problem of how to assign certain nodes at the borders to one of the 4 subdomains of a simple exemplary model

Most readers who have made it to this section will probably have concluded by now that there is no applicable solution that is able to fulfill all the requirements and take into consideration the aspects introduced above. The good news is that the scientists working in this quite challenging field of theoretical computer science and complexity theory have found ways to come up with useful solutions to this major problem of finding the optimal domain decomposition with the best load balancing and the lowest communication rate at the same time. The bad news is that there is still no perfect algorithm, a unique process which would always succeed in providing the best solution to fulfill all requirements. (Section 5.2, starting on page 99, will provide a closer look into the theory behind the problem of domain decomposition.)

Meanwhile some tools have been developed which can be deployed when dealing with subdomains. Unfortunately, before using such tools, the data and models still require a great deal of attention in regard to data-preprocessing as well as post-processing, after the application of these algorithms.

Two quite well-known tools will be introduced in the upcoming sections of this chapter: *Metis* and *Jostle*, which were both used in the wrapper program. Normally just one of them would already be sufficient, but as it was not clear at the beginning how powerful they would be or how well their results would turn out when used with the existing data sets, both have been implemented and can be plugged into the code selectively by using a switch when calling the wrapper program. (See also Appendix C on page 253 for more information.)

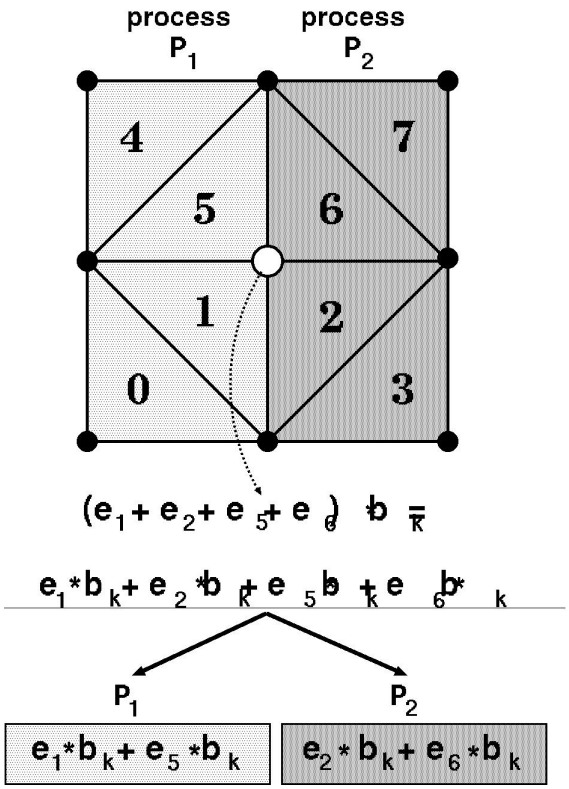


Figure 5.9: The white boundary node b_k will be used by both subdomains in their computations when processing the data of the adjacent elements e_1, e_2, e_5 and e_6

In Fig. 5.10 the whole process of how to prepare an existing mathematical model until it can finally be run on a parallel computer is shown with its main steps. The colorful model representation in the upper right part of Fig. 5.10 is indeed a real outcome of one of the tools mentioned above. It is one of a set of small exemplary test models which were created, read in and used to test a variety of procedures in the wrapper development process. (This example shows clearly that the tools might partition the model in completely different ways than a human being would. The latter would most likely have cut it intuitively into 4 almost similar-looking subdomains.)

The reader might now sigh with relief and think to him- or herself that there is still a lot to do but that there is some light at the end of the tunnel. It seems that the responsible programmer or software engineer just has to sit down, maybe for quite some time, pondering all the aspects and prerequisites and then, after having come up with a sophisticated and thoroughly tested program code, all problems will have been solved, no matter which supercomputer was then being used.

Unfortunately many things also depend on the hardware and architecture being implemented. A pure but big and nicely equipped vector machine has to be programmed in a different way from a general-purpose parallel supercomputer; see also chapter 3. Furthermore, every problem is different. There is no typical "out-of-the-box" solution. Here, an apt comparison by a specialist at the federal high-performance computing center in Stuttgart is fitting. He described the complete software development process in HPC as follows:

> Programming on a normal desktop machine or average non-specialized computer is like using a normal, everyday car, like a Golf, a Mercedes or a BMW. They differ a little bit in their interior, their design, etc. Some of them are faster and more elegant or nicer to look at. Others are just simply "cars" with no special features. Still, they

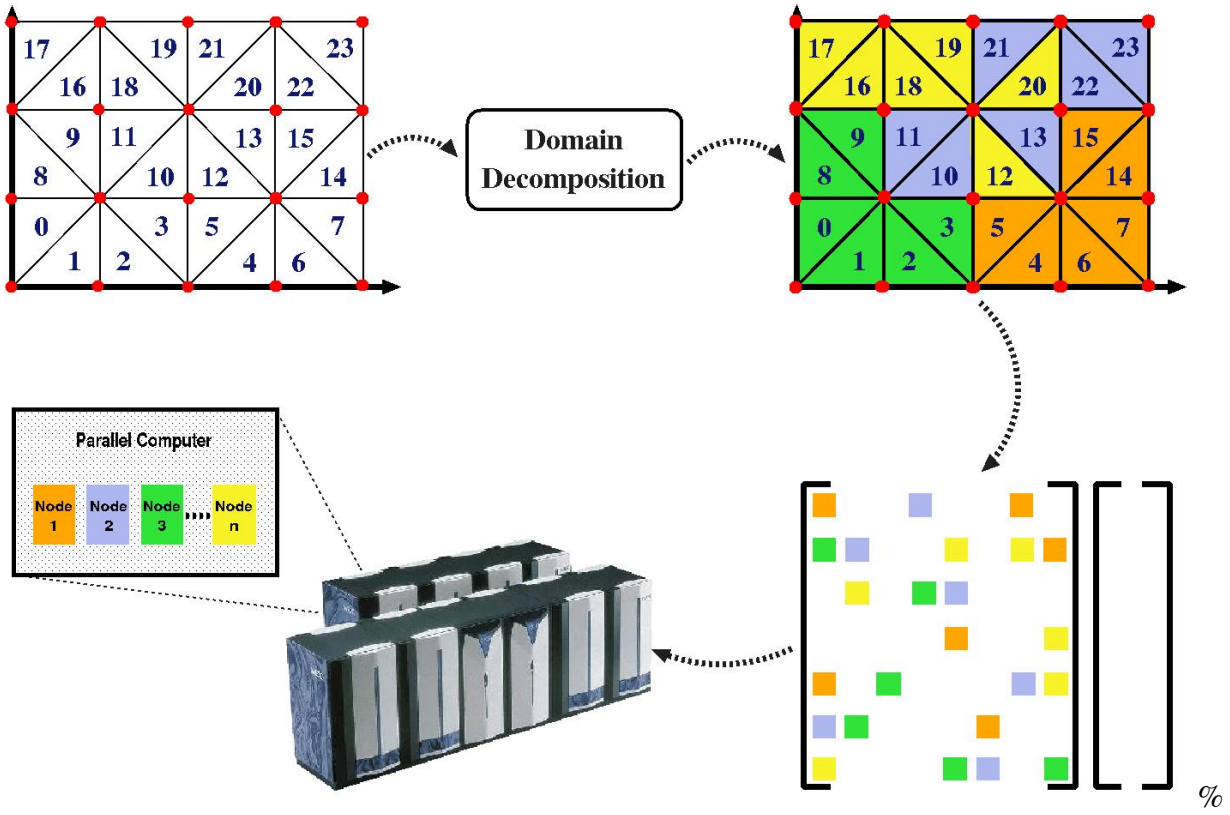


Figure 5.10: The main steps in the process translating a mathematical representation of a model into a form that can be processed on a parallel computer

are all just "cars," machines with which to drive from point A to point B. They are manufactured as a mass product and should suit the average needs of John Public. They should be cheap and everybody should be able to afford at least an average automobile.

Now take a problem that has to be computed on a supercomputer or has to be parallelized. Here you have now a very fast and unique Formula-1 car. It still looks like a car, it also can go from A to B, but every single racing car is a specially designed machine, not a mass product. It is optimized in regard to certain aspects. Although they all look like Formula-1 cars, they still are unique. Their development is kept secret in order to always be ahead of the competing racing teams. There is just a handful of these cars, but a large number of people taking care of every single little aspect.

It is the same with program development for supercomputers. Here you also have to take into account the specialty of the machine, its weaknesses but also its strengths and most powerful features. Every aspect of programming, even those not even considered for an average computer, has to be checked. Therefore the software of the same problem would always look different on each single parallel machine. The code has to be re-written in the worst case any time you are porting it to another supercomputer.

All the previous examples hopefully have made it clear that the aspect of how to partition the overall input data in a reasonable way is a major problem that has to take further difficulties into account like the communication rate and the utilization of every single processing node. Normally load balancing does not always correspond to the most perfect domain composition and vice versa, especially not in regard to communication aspects. Therefore, an optimal way has to be found that

offers the least communication between the nodes and the best load balancing with a reasonable domain decomposition in regard to the input data.

The upcoming sections will deal with the theory behind all this and what intermediate steps are necessary to preprocess the input data in such a way that existing and optimized tools can be deployed.

5.2 A ”Quick Glimpse” into Theory

An average computer scientist would roughly know where to peg the problem of domain decomposition directly, when one tells him or her that this belongs to the *NP-complete problems* (NPC). A theoretical computer scientist working in the field of computational complexity theory would categorize the problem of finding the optimal sets of subdivisions regarding the overall input data as belonging to the *complexity class* NP-complete, which is a subset of NP problems. The latter is the set of all decision problems whose solutions can be solved in polynomial time on a non-deterministic *Turing Machine*. This means, a problem *p* which is in NP is also in NPC if and only if every other problem in NP can be transformed into p in polynomial time.

The problem of interest can therefore be described for computer scientists in the following short but precise way: Given is a graph G with n weighted vertices and m weighted edges. These nodes are now to be divided into p sets in such a way that the sum of the vertex weights in each of the sets is as similar as possible, and the sum of the weights of edges crossing between the sets is minimized. Unfortunately in the simple case where $p = 2$ and the edge and vertex weights are uniform, this graph-partitioning problem is NP-complete.

Until now, no efficient algorithm is known which solves this problem generally, and it seems unlikely that such an algorithm exists. The currently known algorithms and methods are heuristic solutions in which balance may be partially compromised or more typically the minimization is just approximate.

Most likely the non-computer-scientist readers are still wondering what this domain decomposition algorithm is all about and what NPC really means in practical life. A complete introduction to complexity theory is far beyond the scope of this thesis. This is a scientific field in itself within computer science, where legions of computer scientists are working to find the still missing answer to the important question of whether NP = P. But it is also a very important and essential field despite its complexity in which such crucial questions as,

> “When the size of the input to an algorithm increases, how do the running time and memory requirements of the algorithm change and what are the implications and ramifications of that change?"

are of major interest. In other words, complexity theory investigates, among other things, the scalability of computational problems and algorithms. In particular, the theory places practical limits on what computers really can accomplish [Du] [Bovet].

The goal of a classical algorithm is always to guarantee the optimal solution to a given problem with the minimum in runtime. Unfortunately there exist a variety of complex problems in all fields of our daily life, where finding such an algorithm would simply take too long to compute or is just not feasible yet, even with our modern computers.

Unfortunately, even if we have found a suitable and fast algorithm a lot depends on the size of the input data. In a great deal of cases the runtime directly depends exponentially on this data. The algorithm can be as fast and sophisticated as possible, but there is always a limitation in the instance of our problem where a certain algorithm simply is no longer applicable with regard to the input data.

Therefore, scientists abandon or drop some of the high standards and constraints of an algorithm and try to find an applicable solution which is a good compromise between computing efforts, quality in regard to the detected solution and size of input data. The goal is not to find *the* optimal solution but rather one that is fast, effective and close to a perfect solution. Such algorithms are intended to gain computational performance or conceptual simplicity, potentially at the cost of accuracy or precision.

The currently existing algorithms dealing with load balancing all belong to the complexity class NP-complete and implement so-called *heuristic methods*. Depending on the field one is working in, a *heuristic* usually describes slightly different issues. The term *heuristic* is derived either from the old Greek word *heurísko*, meaning "I find," or *heuriskein*, which is "to discover, to find." In the field of computer science a heuristic is a special method similar to an algorithm, which leads to a solution that is reasonably close to the perfect answer to an imposed problem.

One possibility of finding a good heuristic is, e.g., to reduce the solution space by eliminating all such alternatives which seem to be hopeless or unpromising directly from the beginning of the process. Quite well-known heuristics are the *Travelling Salesman Problem* or searching for the shortest paths in existing graphs. Even many commercial anti-virus scanners use heuristic signatures to look for specific attributes and characteristics when detecting viruses and other forms of malware.

The alternative to a heuristic method is the *brute force method*, where all existing alternatives are computed and tested without any exception. The most simple and well-known heuristic is the solution of a problem using the *trial and error method*.

When using heuristics one should always keep in mind that they are fallible, and it is very important to understand their limitations when deploying them. Unfortunately this is often forgotten by the engineer or programmer, as the heuristic might seem to work but is not mathematically proven. One must always respect the given sets of requirements when using such a heuristic. The implemented heuristics need to be tested thoroughly with good test data and the developer should always think about the future data sets or cases that may not exist at the time of development but might be interesting at a later time. The designer must therefore fully understand the rules that generated the output data.

What is meant by this can be shown quickly using a simple example. Let us assume the given sequence of numbers: 1, 2, 4, ... The question is now: What number is next? One heuristic algorithm might say that the next number is 8 because the numbers are always doubling. With this heuristic the sequence might be: 1, 2, 4, 8, 16, 32, and so on. But an equally valid heuristic would predict instead the next number to be 7, as each of the numbers in the given sequence is raised by one interval higher than the one before. With this completely different heuristic the given sequence would be: 1, 2, 4, 7, 11, 16, ...

As shown previously, until now there is no algorithm known which efficiently solves the graph-partitioning problem generally for all possible cases. But a good handful of heuristic solutions has been found so far. In the following, the functioning of the most-used ones will briefly be described.

It is not possible, however, to go into detail for every method here. For this, the reader should refer to the literature which has been published on each of the techniques.

5.2.1 Simple Partitioning Methods

There are three main but simple partitioning algorithms which use either a *linear*, a *random* or *scattered* partitioning scheme. When using the *linear method*, the vertices of the graph are simply assigned to the computing nodes sequentially in accord with the numbering they have in the original graph. In an unweighted graph, consisting of n nodes, which is to be subdivided into p sets, the first n/p vertices will then be assigned to set no. 0, the next n/p nodes to the set no. 1 and so forth. This method produces surprisingly good results because the data locality is often implicit in the vertex numbering.

When using the *random scheme* when partitioning, the nodes are assigned randomly to sets in a way that the balance is preserved. In the last method using the *scattered scheme*, the vertices are handed out in order, with the vertex going to whichever set is at this moment the smallest. In an unweighted case this technique is like dealing out the nodes in a card-like fashion. In practice it was shown that the random ordering produces partitions with a quality in between that of the linear and the scattered methods. An introduction to these basic methods can be found in most publications of partitioning techniques, like in [Fox].

5.2.2 Inertial Partitioning Method

This partitioning technique is also a quite simple one and fast, but it uses geometric information. In addition to the graph itself, it also requires the geometric coordinates for each node in one, two or three dimensions. The vertices are then considered as point masses with their mass values set equal to the vertex weights. Next, the principal axis of this structure is computed. Most likely it will be a direction in which the graph is elongated. In the end the nodes are then divided into sets of equal mass by planes that are orthogonal to the principle axis. Further descriptions of this method can be found, e.g., in [Simon], [Williams] and [Miller].

It could be shown that inertial methods are quite fast but result in partitions of fairly low quality, especially when compared to the *spectral partitioning techniques* described below. Due to the randomized nature of these algorithms, multiple trials are often required (5 to 50) to obtain solutions that are comparable in quality to spectral methods, thus increasing the time. But the overall runtime is still substantially lower than the time required by the spectral methods which are introduced in subsection 5.2.3 below.

Some improvements to the basic algorithms have been made meanwhile, but only in cases with very large problem sizes where the coordinates are available. Such an improvement could be chosen when the emphasis is on low partitioning time rather than on high partitioning quality of the resulting subsets. Another drawback of this method is the fact that it is only applicable if the coordinates are available for the vertices of the graph. For further reading on this topic, the reader can have a look at [Karypis1] and [Chan].

5.2.3 Spectral Partitioning Method

Spectral partitioning algorithms are quite sophisticated methods which take the eigenvectors of a matrix that is constructed from the graph and use these to decide how to partition the graph. It is not possible to give a full accounting of this surprising connection between eigenvectors and partitions here but the interested reader can find further information about this method in the literature mentioned in the following paragraphs.

The simplest spectral partitioning method is called *spectral bisection*. It exists in a weighted and unweighted version and uses the second lowest eigenvector of the Laplacian matrix of the graph to divide it into two pieces. This special eigenvector is known as the *Fiedler vector* in recognition of Miroslav Fiedler. More on this method can be found in [Pothen], [Simon] and [Hendrickson]. More on the pioneering work of Fiedler can be found in [Fiedler1] and [Fielder2].

Instead of bisecting a graph it could also be cut into more pieces using the *spectral quadrisection algorithm*, which divides the graph into four pieces at once using the second and third lowest eigenvectors of the Laplacian matrix. Similarly, the *spectral octasection method* uses the second, third and fourth eigenvectors for a division into eight pieces. Both methods employ more complex metrics to minimize communication costs than the simple bisective technique does. In [Leland2] or [Leland3] further information can be found on these multidimensional spectral methods. Often such algorithms are also referred to as multilevel spectral bisection (MSB).

Spectral partitioning algorithms are usually quite good at finding the right general area of the graph in which to cut but they do poorly in the fine details. Just as improvements have helped to overcome the drawbacks of the inertial methods, they have also benefitted the multidimensional spectral techniques. Here, the necessary improvements depend on the problem. They can result in an improvement rate of 10 to 30 percent, but generally less in larger graphs. Usually MSB algorithms manage to speed up the spectral partitioning methods by an order of magnitude without any loss in the quality of the edge cut. MSB can, however, take a large amount of time. Especially when using parallel direct solvers later in the computing stage when processing the models, the time for computing the ordering using MSB algorithms can be several orders of magnitude greater than the time taken by the parallel factorization algorithm. In such a case, the ordering time can then dominate the overall time to solve the problem. More on this algorithm and its drawbacks can be found in [Leland1], [Karypis1] and [Gupta].

5.2.4 The KL Partitioning Method

The KL partitioning algorithm is one of the more popular methods for partitioning graphs. It dates back to work in the early 70's done by B. Kernighan and S. Lin [Kernighan]. The original idea has been further developed through the years with various extensions and major improvements. One of them is an important linear time implementation due to C. M. Fiduccia and R. M. Mattheyses [Fiduccia], who are often also jointly credited with this algorithm.

At heart, this method is simply a greedy, local optimization strategy. The vertices of the graph are moved between the sets in an effort to reduce the numbers of edges which are cut by the partition. The original algorithm was designed for graph bisection but it was soon extended to the quadrisection case [Suaris].

Unfortunately, the runtime of these algorithms and their memory requirements increase with the partitioning dimension. Practice has also shown that the KL method does not find very good partitions of large graphs unless it is given a good initial partition; see [Leland1].

5.2.5 Multilevel-KL Partitioning Method

Especially for large problems in which high-quality partitions are sought, this algorithm should be the method of choice [Leland4]. The process of finding the optimal partitions is rather complicated and uses smaller graphs of the original graph with which it is approximated slowly. In addition, the KL method is also invoked every few steps to refine the partition.

The algorithm for constructing smaller approximations to the graph relies upon finding a maximal matching in the graph, and then contracting the edges in the matching. Doing this results in a new graph with typically about half as many vertices as the original graph. Edge contraction is intuitively attractive as it largely preserves the former graph topology. When edges are contracted, a single vertex is created out of the two endpoints with weight given by the sum of the weights of its endpoints. In addition, any edges which become coincident have their weights summed and become a single edge. These operations have the effect of preserving the essential properties of a partition as it is moved between the graphs in the hierarchy. More on this can be found in [Leland1].

Experience has shown that the multilevel-KL method gives very high-quality answers in moderate time. It is not as quick as the inertial method plus KL, but it generally produces better partitions. In most cases it results in partitions which are better than those generated by spectral techniques coupled with KL and runs significantly faster than any of the spectral methods. More on the workings and performance of this multilevel-KL algorithm can be found in [Leland4], [Karypis1] or [Ponnusamy], just to name a few. This method is also often referred to simply as multilevel partitioning or k-partitioning algorithm.

Many partitioning tools implement this method nowadays, but some variations in which way the algorithm is realized in software exist, like recursive k-partitioning or iterative bisection partitioning, etc.

5.3 Data Preprocessing - From a 3D Model to a Subdivided Input Set

The previous section was quite theoretical but has most likely not really helped the average non-computer-scientist reader to understand what steps are necessary and need to be considered when implementing domain decomposition in a computer program. For this reason, this section will now deal with the more practical aspects of domain decomposition when using it in a programmer's daily life.

In chapter 2 the important main steps and aspects that are obligatory when working with realisitc 2D or 3D models have already been introduced. At some point of the process, the existing real-life model is translated into a mathematical representation, as shown previously in simplified form in Fig. 2.16 on page 22. Fig. 5.11 depicts now solely the model which was displayed in the lower left corner of Fig. 2.16.

Although the following basics have already been introduced in quite some detail in chapter 2 in section 2.1, starting on page 9, a few of the important terms will be quickly reviewed here in case a reader has skipped the introductory chapters and has plunged right into the more advanced topics of this thesis.

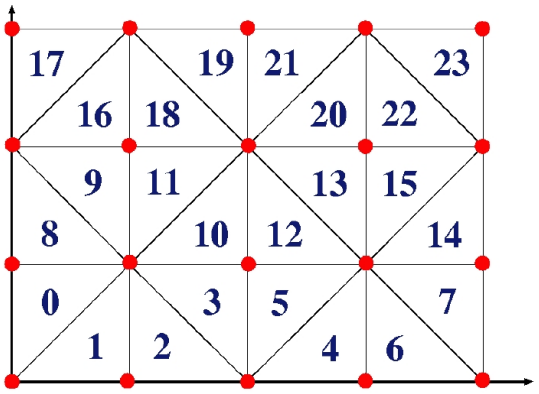


Figure 5.11: A mathematical representation of a simple model with 24 elements and 20 nodes

The model shown in Fig. 5.11 simply consists of *nodes* or *vertices* (the bigger "dots" in Fig. 5.11) and *edges*, which connect the nodes with each other. A certain set of vertices and edges builds a typical geometrical shape, like triangles in this example, but all other shapes like squares, tetrahedrons or cubes, just to name a few, are possible (see also Figs 2.2, 2.3 or 2.4, starting on page 12).

As can be seen in Fig. 5.11, the triangles are all numbered, starting with the number 0 in the lower left corner of the mathematical model, and end with 23 in the upper right corner, which means we have 24 triangles here all together. Such shapes are called *elements* and the consecutive numbers are the corresponding *element numbers*. The models with which the wrapper program was tested in the end consisted of up to 1 mio. elements. The nodes are also numbered (they are just not shown in the figure for readability purposes) and are called *node numbers* accordingly. In the exemplary model in Fig. 5.11 we have 20 vertices all together, starting with node no. 0 and leading up to node no. 19.

Now let us assume that the vertices represent specific locations in a given area in real life, where we have drilled and extracted a bore core. Each of these locations delivers hard facts like, for example, what rocks are present at which depth, which results of special physical measurements were found, etc. But this all is just discrete data. In the end we would like to make a statement on what the entire terrain might look like when talking about the whole area and not just the few spots where we drilled. This means that not just the nodes themselves will be important later on, but also the elements, representing the small areas in between the bore sites, although initially no information on what is going on in a single triangular area is available at all. The vertices are important later on, as they give the values that can then be used for evaluation and interpolation purposes, in order to find a good approximation of the data pertinent to the elements.

Therefore nodes and elements demand equal attention and care in the programming and development process, although it may seem that the nodes are no longer of importance in the upcoming steps. The connection between the element with its building nodes must never be lost.

We haven't talked about the edges so far. In the real-life and exemplary models which were used in this thesis, the edges were of no importance. But there are also cases possible where such edges are weighted depending on what is to be computed or what has to be modelled later on in the computing process. Here, the edges are no longer of any interest. It turned out during the development process of the thesis that it is wise to treat them equally as bi-directional edges with a uniform edge weight of, e.g., the value of 1.

As was shown in Fig. 5.10 on page 98, it is now necessary to translate this mathematical representation in such a way that a matrix-vector system can be generated from the input data for further computations and modelling. For generating the appropriate numbers of submodels, domain decomposition tools are plugged in between. (We had assumed this missing step and its solution for reasons of simplification in chapter 4 in section 4.4 when talking about the changes when converting a formerly sequential modeling program into a version which works in parallel.)

To be able to use such specialized tools which help subdivide the existing model which is built from the input data file, this input information has to be treated in a certain way so that these tools, e.g., Jostle or Metis, can actually work with it. After having called these tools, the original connection between the elements, nodes, edges and the corresponding information must still be existent and retrievable.

The problem is that subroutines which are going to be called when working with domain decompostion tools expect the data in a certain format. Unfortunately the input file format rarely complies with such requirements. This means the data representation of the mathematical model, like the one in Fig. 5.11, has to be changed without changing the original model itself and without losing any relevant discrete information attached to the vertices. Often the hard physical data of the nodes or of the model itself is stored in a different file than the model file is. Some reference number like, e.g., the node number, is then used to retrieve the corresponding information from the different file that belongs to the corresponding vertex. In Appendix D on page 257 a few very short exemplary input files demonstrate in which way the models are stored. How they are going to be processed will be dealt with in chapter 6, which considers solely the implementation aspects of this thesis.

With regard to the computations and interpolations that are done during the processing of the model and its data later on, converting the original *graph* with its vertices and edges into a so-called *adjacency graph* is the most important step. The reason for this is quite simple and explained with a little example. When having a look at element no. 10 in Fig. 5.11, for instance, it would be interesting to know how, during the modeling process, the relevant data is going to change when moving over to the elements no. 3, no. 11 and no. 12, which are directly adjacent to element no. 10. Element no. 23, on the other hand, is already a good distance away from element no. 10. When talking, for example, about the hydraulic head that is present in the area represented by element no. 10, it is more than likely that it is not going to change within the "a snap of a finger" but might rather smoothly shift to a different value when moving to its adjacent elements. But it could already be much higher or lower at element no. 23, as this is already some "element hops" away from our element of interest.

This means that during interpolation and the modelling process, it is of utmost importance which element is a direct neighbor to which other element and which are too far away. Thus, this information must be kept under all circumstances, especially when thinking about subdividing the existing model into smaller subdomains. Such information can be presented nicely using adjacency graphs. In Fig. 5.12 a simple model is shown, which consists of 16 vertices that build 7 square elements. On the right-hand side of this figure its corresponding adjacency graph is shown, which has accordingly just 7 nodes. Every vertex of the adjacency graph relates to exactly one of the seven elements in the original graph. But how is such an adjacency graph built from its corresponding model? Again Fig. 5.12 can serve to illustrate the process of setting up an adjacency graph.

For this, let us start with the square element no. 1 in the lower right-hand corner of the original model shown in Fig. 5.12. This element is a direct neighbor to element no. 2 right above it. But over the upper left-hand corner vertex of element no. 1, this element is also connected diagonally to element no. 3. This means that element no. 1 has exactly two neighbors. When now checking

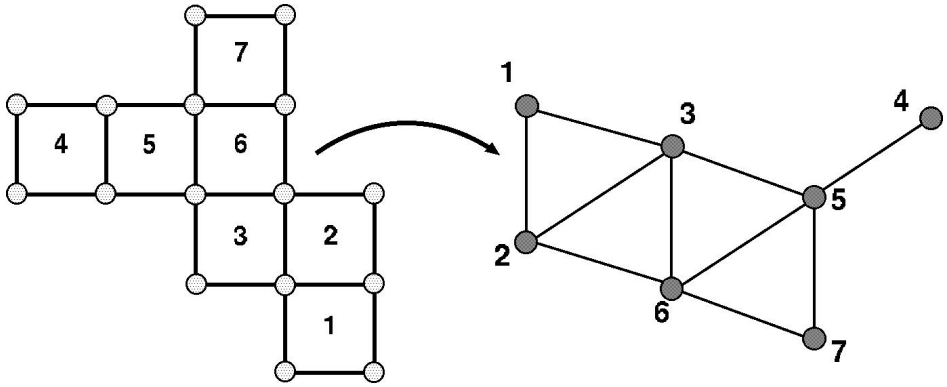


Figure 5.12: A simple mathematical model representation on the left and its corresponding adjacency graph to the right

the adjacency graph we see a vertex no. 1 which is connected by two edges to the nodes no. 2 and no. 3, i.e., element no. 2 and element no. 3. In the original model element no. 1 had no direct connection to any of the other elements of the model. And so it is in the adjacency graph. The vertices with the numbers 4 and higher can't be reached directly from node no. 1 there, just by traversing other vertices in this adjacency graph.

In the course of stepping through the original model of Fig. 5.12 element by element, repeating the same selection process again and again for all elements present, the complete adjacency graph is being built. Element no. 4, for instance, has just one and only one direct neighbor, which is element no. 5. In the graph to the right, we see therefore vertex no. 4 by itself with just one single edge connecting it to vertex no. 5. Node no. 6, for example, has 3 direct neighbors, which are the elements no. 3, no. 5 and no. 7. Element no. 2 is also a neighbor but using the rule with the mutually shared vertex. In the adjacency graph we find therefore the corresponding node no. 6 of the element no. 6 right in the middle, connected directly to the vertices with the numbers 2, 3, 5, and 7.

By slowly converting the original mathematical model representation as it is shown at the left of Fig. 5.12, its resulting adjacency graph is depicted at the right in the same figure. Although the corresponding adjacency graph looks completely different from the original model, the information connected to it is not lost. The adjacency graph is just another form of presenting this data (if the programmer or software engineer was careful enough to still keep the node information).

Unfortunately in real life the conversion between the original model and its resulting adjacency graph is not so easy as Fig. 5.12 might suggest. Much depends on how the data one starts out with is stored on the computer, if the input information is spread among several files and in which way the resulting subdomain data is processed later on using the supercomputer or parallel machine. What might appear here to be a simple step from a mathematical model to its adjacency graph will most likely result in a long listing of code until the conversion on the computer is finally processed. Another important question is the size of the input data sets that have to be kept in mind. Furthermore, one should always remember all the additional prerequisites in regard to load balancing, communication and domain decomposition which were introduced earlier in this chapter. One of the upcoming chapters, chapter 6, deals later with all implementation aspects as they were imposed for this thesis.

Many of the existing domain decomposition tools can be deployed once the adjacency graph has been created. The following two sections introduce two different domain decomposition tool packages, Metis and Jostle, which were implemented into the wrapper program.

5.4 Jostle

Jostle is a graph-partitioning software package that was designed to partition unstructured meshes like, for instance, finite element or finite volume meshes which are to be computed on distributed memory parallel computers (see also chapter 3).

Jostle was created by Dr. Chris Walshaw and is freely available for academic and research purposes. But still the interested user is required to sign a licensing agreement. There is a stand-alone serial as well as a parallel version available. The software can be run in principle on almost any platform where a C compiler is available, like, for example, Linux PCs, Sun or Solaris Workstations, Silicon Graphics Workstations but also Linux PC Clusters, NEC supercomputers, Cray supercomputers, etc.

The Jostle package is well documented. In this section just the very basics are introduced. The interested reader can find more information about Jostle in [Walshsaw01], [Walshsaw02] and [Walshsaw03]. Furthermore, Chris Walshaw has also published a great deal of advanced literature on the topic of domain decomposition: See, for instance, [Walshsaw04] or [Walshsaw05]. For more publications by Chris Walshaw, the reader is referred to Appendix F, page 262, where the bibliography is located.

5.4.1 The Chaco Input File Format

Jostle as well as Metis, which will be introduced in the next section, 5.5, beginning on page 112, both use the Chaco input file format. This is also known as the *compressed storage format* or as CSR format.

Chaco is another but older partitioning software package for UNIX systems which was developed at the Sandia National Laboratories in New Mexico, USA, in the early nineties of the last century. It can be obtained and run under the license of the Sandia National Laboratories and should compile and work correctly on machines that use an ANSI standard C compiler.

Chaco implements the five main classes of partitioning algorithms which have been introduced and discussed previously in this chapter in subsections 5.2.1 to 5.2.5. Unfortunately its functionality is controlled by a fairly large set of parameters, and version 2.0 is about 30,000 lines of code long.

Chaco also works with an adjacency graph representation when dealing with input data sets. The following arguments and parameters are used to describe the graph as input data:

1. **nvtxs:** Integer value which is the number of vertices in the graph. Numbering starts from 1 to *nvtxs* (*nvtxs* = number of vertices).
2. **start:** Integer value which is 0 or 1 depending on what numbering mechanism is used in the graph. In C generally the nodes in the graph start at 0; often in Fortran the numbering starts at 1 instead.
3. **adjacency:** Integer array which contains a list of all edges for all vertices in the graph.
4. **vwgts:** Integer array of length *nvtxs* which specifies the weights for all the vertices in the graph. The vertex weights have to be positive. In case all nodes in the graph are given a unit weight, a NULL pointer is passed instead (*vwgts* = vertex weights).

5. **ewgts:** Array of float which is of the same length as the *adjacency array* and is also indexed the same way. It specifies the weights for all edges (*ewgts* = edge weights).

6. **x:** Array of float and with length of *nvtxs*. It is used to hold the geometric x coordinate for each vertex of in the graph.

7. **y:** Array of float and with length of *nvtxs*. It is used to hold the geometric y coordinate for each vertex of in the graph. If it is set to NULL, the geometry is assumed to be one dimensional.

8. **z:** Array of float and with length of *nvtxs*. It is used to hold the geometric z coordinate for each vertex of in the graph. If it is set to NULL but *y* is not NULL, the geometry is assumed to be two dimensional.

How this lengthy input file format can be used and deployed will be shown below using some simple exemplary input graphs when describing Jostle and later on Metis.

5.4.2 How Does Jostle work?

Jostle uses an advanced multilevel refinement and balancing strategy which is described further in [Walshsaw05]. A series of increasingly coarser graphs are constructed. An initial partition is calculated on the coarsest graph and the partition is then repeatedly extended to the next coarsest graph and refined and balanced there. The applied refinement algorithm is a multiway version of the Kernighan-Lin iterative optimization algorithm which incorporates a balancing flow. How this all is achieved and processed goes far beyond this thesis, but the interested reader should have a look at [Walshsaw05], [Hu] or [Song].

Jostle can also repartition an already partitioned graph. Furthermore, Jostle is able to work with disconnected graphs where the problem of isolated nodes or completely disconnected components can arise. As the data sets used for this thesis and for the development of the wrapper software were required to always be connected, this extension of Jostle was not used or further regarded.

5.4.3 Usage of Jostle

Jostle can be used either as a stand-alone program or as part of another software in which Jostle is then called as a function.

As a stand-alone application Jostle requires two input parameters. The first one is the filename in which the input data set, the adjacency graph, is stored. The second one is the number of subdomains which are to be produced. An exemplary call from the commandline could look like as this:

```
jostle inputfile 8
```

In this case the input file must have a certain format that can be recognized by Jostle. Since this possibility was not intended for such work, it will not be considered here any further.

Jostle is built into the wrapper software like an internal function. This means that when calling and running the wrapper, the user will not see any of the work that Jostle does. The only point in

the whole computing process at which he or she will realize that Jostle is included is when setting a switch for Jostle usage on the commandline. Everything else is hidden from the user. In case another switch for a statistical analysis was set, the program user will get an output with certain facts from the processing.

As indicated previously, when calling Jostle and Metis, respectively, the input data is expected to have a certain format that can be processed by the internal built-in functions themselves. Therefore the input data set to be partitioned is reformated by the wrapper program before the call to Jostle takes place. This process is also hidden from the user of the wrapper. Due to data structures that allow fast processing, even bigger input models of more than 1 mio. elements are reformatted rather quickly so that the resulting partitioning does not take very long at all. For the implementation and the data structures used, the reader can consult chapter 6, beginning on page 119.

Let us assume that we have again the already introduced exemplary model shown in Fig. 5.11 and its corresponding adjacency graph as depicted in Fig. 5.13.

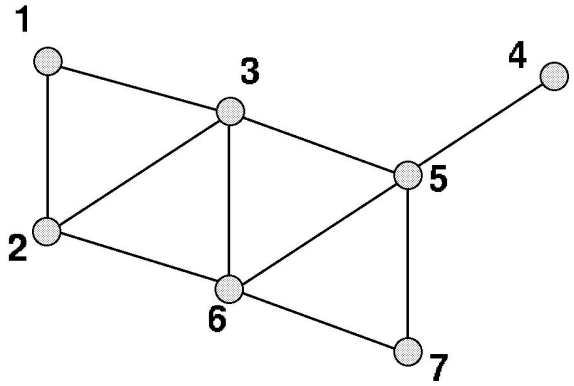


Figure 5.13: A simple adjacency graph consisting of 7 vertices representing 7 elements in its former mathematical representation

Its correct representation as an input data set as Jostle would await it for processing would look like the following:

```
7 10
2 3
1 3 6
1 2 5 6
5
3 4 6 7
2 3 5 7
5 6
```

Comparing this format with the given construction of the input data files that have to be preprocessed by the wrapper program (some examples are shown in Appendix D on page 257) demonstrates easily the necessity of building intermediate data sets before calling Jostle. The final call to Jostle within the program would look similar to the following when working with the programming language C:

```
jostle( &nnodes, &offset, degree, node_wt, partition,
        &nedges, edges, edge_wt, &nparts, part_wt,
        &output_level, &dimension, coords );
```

For the non-computer-scientist user this might all look rather confusing. Therefore, let us step through it using the example given above. As Jostle uses the already introduced Chaco file format, the given scalar quantities of the corresponding data set of the adjacency graph shown in Fig. 5.13 translate to the following:

```
nnodes     7
offset     1
nedges     20
dimension  0
```

`nnodes` is set to 7 as there are 7 vertices in the adjacency graph shown in Fig. 5.13. The numbering starts with 1 for the first vertex. Therefore, the variable `offset` is set to 1. If the number of the first node had been set to 0, as is common when working with the programming language C, `offset` would have been set to 0 instead.

The value of 20 for `nedges` needs a little explaining. When counting all edges in this adjacency graph we get a total of 10. But when taking as an example the edge between vertex no. 3 and no. 6, we could traverse it in two directions, either starting from node no. 6 and crossing to node no. 3 or the other way round. In this example the direction is of no importance but there are cases when it matters from which side an edge is being crossed. Therefore we have to count edges without a given direction twice. As all edges in the example are treated as so-called bi-directional edges, the resulting total is 20 for `nedges`. The variable `dimension` is not used in Jostle and thus is set to zero.

Now let us examine the array structures that Jostle uses when being called. The values using the example above would look as follows:

```
degree      [ 2, 3, 4, 1, 4, 4, 2]
node_wt     (int *) NULL
edges       [ 2, 3, 1, 3, 6, 1, 2, 5, 6, 5, 3, 4, 6, 7,
              2, 3, 5, 7, 5, 6]
edge_wt     (int *) NULL
coords      (double *) NULL
```

The variable `degree` gives the number of edges which are connected to this node. For instance, the vertex no. 2 has 3 edges which connect it to the vertices no. 1, no. 3 and no. 6. Therefore, we find the value 3 in the array, whereas node no. 4 is just connected to vertex no. 5, which is indicated by the value of 1 in `degree`.

`node_wt` is set to zero, as all nodes have the same weight in our example. There are cases when the node weight is important. In case node no. 3 had a weight of 4, the array of `node_wt` would look like `[1,1,4,1,1,1,1]`.

To understand the values in the lengthy array of `edges` we must have a look at `degree` again. There we found a value of 2 for two adjacent nodes which are directly attached to vertex no. 1.

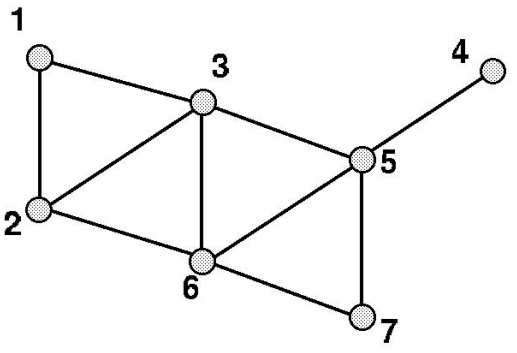
Now, when examining the first two entries in `edges` we find these direct neighbors of node no. 1, i.e., vertex no. 2 and no. 3. Node no. 2 had a value of 3 in `degree` and thus, its directly connected neighbors are given as nodes no. 1, no. 3 and no. 7, and so on. Counting the total of the entries in `edges` results in a value of 20. Counter-checking this value with `nedges` above, we see the same value of 20 that indicates the correct entries of `edges`.

With the value of `edge_wt` we have the same situation as with `node_wt`. In our exemplary graph in Fig. 5.13, all edges have the same weight, which could be 1 for simplicity reasons. Therefore, all edges are treated the same and no further consideration of the edge weight is necessary. But when assuming a different weight of 3 for the edge between vertex no. 2 and no. 3, for instance, the entries for the array of `edge_wt` would change to

```
[1,1,1,3,1,1,3,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

The value of 3 appears twice in `edge_wt`, as we have to traverse the edge from node no. 2 towards node no. 3 and then again in the other direction. The last array, `coords`, is not used in Jostle and therefore, it is set to zero.

The total of entries in `degree` and `node_wt` is of length `nnodes`, which was 7 in our example. Correspondingly, there should be as many entries in `edges` and `edge_wt` as the value of the variable `nedges` gives, which is 20 in this exemplary graph. Fig. 5.14 offers a short summary of how the main data structures would look in the little example.




```
nnodes = 7
--> Graph contains 7 nodes

offset = 1
--> Graph starts at one instead of 0

nedges = 20
--> Sum over the number of edges

degree = [2,3,4,1,4,4,2]
--> Array of edges to neighbor nodes

edges = [2,3,1,3,6,1,2,5,6,5,
         3,4,6,7,2,3,5,7,5,6]
--> Neighbor nodes per node
```

Figure 5.14: A simple adjacency graph consisting of 7 vertices with the corresponding values of the Jostle data structures

As indicated previously, Jostle is also able to treat disconnected graphs and already partitioned data sets. In such a case, more array structures must be filled and additional environment variables have to be set for a correct function call of Jostle. For the work of this thesis, such cases are not relevant and are therefore not further examined. The interested reader can check [Walshsaw01], [Walshsaw02] and [Walshsaw03] to learn more about the handling of such data sets.

In a later version of the wrapper program a statistic tool was integrated and, depending on the partitioning tool which was used, certain parameters can be checked and printed out for the interested user. Jostle offers 4 additional subroutines in regard to the partitioning process which can be deployed:

```
int     jostle_cut();
double  jostle_bal();
double  jostle_tim();
int     jostle_mem();
```

The integer value of `jostle_cut` returns the total weight of cut edges when the partitioning is terminated. The function of `jostle_bal` gives the final partitioning balance as the ratio of the maximum subdomain weight over the optimal subdomain weight. A return value of 1.0, for example, would mean a perfect balance. With the value of the function `jostle_tim` the user is informed about the total runtime in seconds of Jostle when partitioning the input data. The last function of `jostle_mem` returns the total memory used by Jostle in bytes during the partitioning process. More on such statistics will be explained in the implementation chapter beginning on page 182.

5.5 Metis

Metis is, like Jostle, a software package which was implemented primarily to partition large and irregular graphs or meshes. In addition it can also be used to compute fill-reducing orderings of sparse matrices.

Metis was created by George Karypis and Vipin Kumar of the University of Minnesota, Department of Computer Science, and the version used for the software of this thesis is version 4.0, published in September 1998 [Metis]. (*"Metis"* is the Greek word for wisdom, but Metis was also a titaness in Greek mythology. She was the consort of Zeus and the mother of Athena. She presided over all wisdom and knowledge.)

Metis is freely available and consists of three different packages. The first major package is called simply `Metis` and can be used for serial graph partitioning as well as sparse matrix ordering. It is available both as a set of stand-alone programs and as a library. The second major package is called `ParMetis` and it is a parallel version which provides static graph and mesh partitioning as well as dynamic graph partitioning, and the fill-reducing ordering of sparse matrices. It is just implemented as a MPI-based library. (For MPI see also 3.9.1 in chapter 3.)

There is also a third but very specialized package available which deals with serial hypergraph partitioning. This one is called `hMetis`. It is specifically optimized for partitioning large VLSI circuits and is available both as a set of stand-alone programs and as a library. As VLSI circuits are not of any interest for the thesis, `hMetis` will not be considered any further here.

Metis is written entirely in ANSI C, and version 4.0 consists of over 22,000 lines of code. As most Unix systems provide a C compiler, like the GNU C compiler which is commonly used, Metis is portable onto most Unix systems. It has been extensively tested on AIX, SunOS, Solaris, IRIS, Linux, HP-UX as well as BSD and Unicos. There is a file called `INSTALL` which comes with the distribution, in which detailed instructions are given on how to build, compile and install Metis for the system in question. Improvements and further information in regard to Metis can be found on its homepage; see [Metis].

The Metis software package is nicely documented. This section just gives an overview of the very basics. Many aspects resemble Jostle, which was introduced previously in section 5.4 on page 107. The interested reader can find further information about the algorithms implemented in Metis in

[Karypis1], [Karypis2] and [Karypis3]. All other information about Metis and its documentation can be found on the official homepage [Metis].

5.5.1 How Does Metis Work?

The algorithms used in Metis do not operate directly on the original graph as the traditional graph partitioning algorithms do. Metis is rather based on the multilevel partitioning methods as they have been introduced in subsections 5.2.4 and 5.2.5 in this chapter.

Generally, `pmetis` and `kmetis`, the two main programs within the stand-alone version of Metis, implement two different methods, but both focus on partitioning an unstructured graph into k equally sized parts. The partitioning algorithm used by `pmetis` is based on multilevel recursive bisection (see also [Karypis1]), whereas the partitioning algorithm used by `kmetis` is based on multilevel k-way partitioning, which is described in [Karypis2].

In the routines that can be accessed using the library interface, called *Metislib*, the same algorithms as in the stand-alone version of `pmetis` and `kmetis` are implemented. Just the usage of these library functions in Metislib is a little bit different.

5.5.2 Usage of Metis

The usage of Metis depends significantly on the application to be computed. Both `pmetis` and `kmetis` are able to produce high-quality partitions. However, depending on the application, one program may be preferable to the other. `pmetis`, or the equivalent Metis library function, should be chosen in those cases where just a small number of partitions are to be created out of an input graph. But if more than 8 subdomains are needed, `kmetis` or the corresponding function in Metislib should be used instead. Both programs, `pmetis` and `kmetis`, can partition a given graph or mesh into an arbitrary number of subdomains.

The library interface rather than the stand-alone versions has been used in this thesis. This way, Metis can directly be accessed from a C or Fortran program over the stand-alone library Metislib. Furthermore, Metislib extends the functionality provided by the stand-alone programs through the option of altering the behavior of the various algorithms implemented in Metis. Also, additional routines are available.

The stand-alone version of Metis is used in very much the same way as the stand-alone version of Jostle. It can be invoked by the following command, depending on which of the two programs is being called:

```
pmetis input_file number_of_subdomains
kmetis input_file number_of_subdomains
```

In order for `pmetis` and `kmetis`, respectively, to be able to process the given input file, it must have a certain structure, otherwise it cannot be recognized. Since the stand-alone versions of Metis are not pertinent to this thesis, they will not be considered here any further.

As Metis is implemented into the wrapper program over its library interfaces, the user does not see anything of the work that Metis does. The only point in the whole computing process at which he

or she will realize that Metis is going to be used in the program run is when setting a switch for the usage of Metis on the commandline. Everything else is hidden from the user. In case another switch for a statistical analysis was set, the user will get an additional output with certain facts from the processing upon program termination.

Similar to Jostle, the Metis library functions implemented in Metislib use the CSR input format (see also subsection 5.4.1 on page 107) to process the adjacency graph that is being built from the input model data. This means, according to the process using Jostle, in Metis there are also two important arrays that have to be fed into the Metis functions: one array, which holds the number of adjacent vertices from a certain node, and a second one, which gives a listing of all node numbers that are directly attached to a specific vertex.

The function call in Metis is a little bit different from the one in Jostle. The construction of the arrays to be used later on by Metis also varies slightly in comparison to Jostle. But the arrays that are used for the function call are also constructed by exploiting the adjacency graph. (The reader should keep in mind that this graph is built upon the elements and not from the original nodes and edges resembling the input model! See page 105 for a refresher on this topic.) Fortunately, the differences between the two domain decomposition tools are just minor.

Normally the arrays used in Metis are set to be from a C type of `int`, but if necessary, on certain platforms they can also be changed to a C type of `short`.

Calling the Metis functions implemented in Metislib from a program would now look as follows, depending on how many subdomains are to be created by this function (the cryptic names that are used throughout the Metis documentation have been replaced by names that are more descriptive to the reader):

```
METIS_PartGraphRecursive(&total_of_elements, metis_vertices,
                metis_neighbor_listing, metis_vertex_weights,
                metis_neighbor_weights, &metis_weight_flag,
                &metis_offset, &number_of_partitions,
                metis_options, &metis_edge_cut, metis_partition);

METIS_PartGraphKway(&total_of_elements, metis_vertices,
                metis_neighbor_listing, metis_vertex_weights,
                metis_neighbor_weights, &metis_weight_flag,
                &metis_offset, &number_of_partitions,
                metis_options, &metis_edge_cut, metis_partition);
```

The first of the two calls should be used when the input model is to be subdivided into 8 or less partitions, whereas the second one is the correct call to choose when subdiving the data set into more than 8 subdomains.

For reasons of completeness, let us use the first function call as a little example. Assuming the same adjacency graph depicted in Fig. 5.13 on page 109, the two important arrays with which Metis would work when using the first function would have a structure like the following:

```
metis_vertices          [ 0, 2, 5, 9, 10, 14, 18, 20 ]
metis_neighbor_listing  [ 2, 3, 1, 3, 6, 1, 2, 5, 6, 5, 3,
                          4, 6, 7, 2, 3, 5, 7, 5, 6 ]
```

The values in the array called metis_neighbor_listing are identical to the ones in the array edges used by Jostle, as shown on page 110. Only the array metis_vertices differs in comparison to the array degree when using Jostle. On closer look one will quickly see that the information given in the array metis_vertices used by Metis is exactly the same as that given in degree of Jostle.

When examining the values in metis_vertices one realizes that the values are continuously growing, starting with a value of 0 and terminating with a value of 20, which is exactly the maximum number of edges in the exemplary adjacency graph in Fig. 5.13 on page 109. In Jostle the array degree had the following structure: [2, 3, 4, 1, 4, 4, 2]. When now adding the value of 2 in the degree array of Jostle to the value of 0 in metis_vertices, the value 2 results. Adding the second value in degree of Jostle to the second value of metis_vertices the total is 5, which is indeed the correct entry in the metis_vertices array. Now the scheme should be clear: In Jostle the total number of adjacent vertices is stored in the array of degree, whereas in Metis the position in the array of metis_neighbor_listing is given at which the adjacent nodes belonging to a certain vertex in the adjacency graph can be found.

In Fig. 5.13 on page 109, node no. 4 is just connected to vertex no. 5, and node no. 4 is also the fourth node that is stored in the arrays. When counter-checking the array structure, one finds at the fourth position in metis_vertices an entry with a value of 9. Now using this value of 9 to find the correct neighbors of node no. 4 in the array metis_neighbor_listing, there is just one value of 5, which is indeed the only adjacent vertex to node no. 4. There cannot be any more neighbors, as the next entry in metis_vertices for node no. 5 of the adjacency graph says explicitly that the following values, beginning at a position of 10 in the array metis_neighbor_listing, already belong to node no. 5 and no longer to node no. 4.

There is one important difference, however, when comparing the two ways of storing the information. In Jostle, the array degree has exactly as many entries as there are vertices in the adjacency graph. In the Metis array structure metis_vertices, there is always one more entry than there are nodes in the adjacency graph. In the exemplary arrays given above, there is a value of 20 in metis_vertices. It is the 8th entry of the array but there are just 7 nodes in the adjacency graph. One could use this information as a control value. At the position of 20 in the array metis_neighbor_listing, there should not be any more entries. Starting from 0 and counting up to 19 already comprises 20 different values. At a position 20 in the array metis_neighbor_listing, one would find a 21st value; this cannot be, as the adjacency graph has an edge count of just 20 and not 21!

Metis also offers, as does Jostle, the handling of weights for the nodes in the adjacency graph as well as its edges. If no weights are used, the array holding the weights can be set to NULL. Otherwise a specific array is filled with values for the weights between zero and greater than zero. (As weights are not used in the thesis, the handling of weights by the library calls are left out here.)

With all this information in mind, the corresponding variable structures for the call of the Metis library function for the exemplary adjacency graph of Fig. 5.13 on page 109 would look like the following:

```
total_of_elements       7
metis_vertices          [ 0, 2, 5, 9, 10, 14, 18, 20 ]
metis_neighbor_listing  [ 2, 3, 1, 3, 6, 1, 2, 5, 6, 5, 3,
                          4, 6, 7, 2, 3, 5, 7, 5, 6 ]
metis_vertex_weights    (int *) NULL
```

```
metis_neighbor_weights   (int *) NULL
metis_weight_flag        (int *) NULL
metis_offset             1
metis_options            0
```

The reader will notice the similarities between Jostle and Metis. For tuning purposes in Metis, in very special cases the user can set an additional option variable to a certain value. None of them are of any interest when working with the data sets of this thesis. Therefore they are not further discussed here. Upon a successful completion of the library call the variable `metis_edge_cut` will store the number of edges in the graph that are cut by the resulting partition. The last variable, listed in the function call under the name `metis_partition`, holds the partition as it was created by Metis.

5.6 Comparison and Future Improvements

In the previous sections the main differences between Metis and Jostle have been shown. It is fair to say here that a real comparison between the two tools is rather difficult, as Jostle was developed much earlier than Metis. Furthermore, Jostle resulted from the early work by Chris Walshaw and has not really been improved since then, as he works on it only in his free time. Metis, on the other hand, resembles something like a professional tool, and a bigger supporting and actively working team stands behind it.

A comparison of the effective outcome when using the two tools shows few differences. A complete discussion in regard to measurements and results when applying Metis and Jostle to the input data sets will be given in chapter 7, starting on page 201.

At the stage of the implementation and writing of this thesis, the parallelization process of GS/RF was just at its very beginning. For practical and testing reasons the preprocessing process was decoupled from the main program. This way the domain decomposition tools could be freely tested and the preprocessing routines which produce the necessary subdivided data sets could be improved without affecting the parallelization process of the solver and main routines.

The whole preprocessing process in which the original discrete model data is read in and which produces the necessary intermediate model data for subsequent computation by the supercomputer can be regarded as a stand-alone application. This approach has its pros but also cons.

The benefit of the decoupling is clearly that time is no longer of any importance. The preparation in regard to the data sets can be done whenever it seems suitable. One can even produce different domain decomposition files of the same input model with just a different number of subdomains in advance. The processing of these intermediate files does take some time. But as this work is completely independent from the main computation on the parallel computer, the parallelization is not affected by the preprocessing. Testing and improvements during the programming phase of the preprocessing can be done without risking a program failure later on when using the expensive resources of the number cruncher.

Furthermore, the preprocessing is machine or platform independent. There is no need for the creation of the intermediate data sets to be done on the parallel computer. The domain decomposition and the data preparation can be run on a single-processor machine as well as on any other type of computer. This is an important aspect as nowadays the computing time that is consumed on a big

supercomputer or specialized parallel machine is charged by the facility. When such computers are then used for tasks that can also be accomplished on cheaper environments the users save quite a large sum of money.

The clear disadvantage of this method is not of major importance but concerns the decoupling. When in a later stage of the parallelization process the parallel solver produces results that show that slight variations of the subdomain models are necessary, these important changes cannot be made dynamically during the parallelization itself. Instead, the process has to be stopped, the preprocessing has to be initiated with these slight variations and only then can the parallelization be restarted again with the new subdomain model data. Without a decoupling of the preprocessing, expensive computing time on the parallel machine might be consumed for simple tasks but the whole parallelization process itself need not be interrupted, and one can react immediately by varying the input parameters. With a good programming, the computing time that is consumed by the preprocessing can be minimized.

As nowadays, especially in the big HPC centers, the time slots for using the specialized parallel machines are quite narrow, an interruption during the parallelization process for generating new input data sets offline may mean waiting for hours or days to get back into the computing queue.

As a solution to this problem, future improvement definitely lies in parallelizing the preprocessing itself. This is possible as, e.g., Metis offers a parallel version that can be implemented into the preprocessing routine. There are other software packages available similar to Jostle and Metis on a single-processor environment which can then be deployed for a parallel preprocessing.

In this way a little bit of expensive supercomputer computing time is still consumed by the preprocessing task, but as the preprocessing procedure itself can be accelerated through its parallelization, it can be integrated directly into the main parallelization process. In case slight changes are necessary during the parallel processing of the model data, the responsible engineers can dynamically vary the parameters directly on the supercomputer and an interruption is no longer necessary. Time slots need no longer be lost, as the full computing time that was granted can now be used without interruptions.

The clear drawback of this improvement is the fact that when there are problems during the preprocessing stage, the main parallelization process itself is also affected. Testing and improvements during the development and programming phase of the parallel preprocessing routines directly show their effects on the main parallelization run. In case the integrated preprocessing part of the whole parallelization program runs into problems or even terminates with a program crash, the complete time slot or computing time is wasted.

5.7 Summary

Large-scale numerical simulations using unstructured graphs or meshes to be processed on parallel computers require a thorough distribution of the data to the processors. This distribution must be done in such a way that the number of elements assigned to each processing node is the same, and that the number of adjacent elements assigned to different processors is minimized. The goal of the first condition is to balance the computations among the computing nodes of the parallel machine. The goal of the second condition is to minimize the communication resulting from the placement or the number of boundary interface nodes per subdomain.

Graph-partitioning algorithms and tools can be deployed to successfully satisfy these imposed conditions by first re-modeling the meshes or graphs by special graphs suited to these partitioning problems, and then subdividing the model data into almost equal parts. Various tools and libraries are available which implement optimized heuristics to find a good partitioning. Two known and often-used packages for this purpose are Jostle and Metis.

Chapter 6

Implementation of the Domain Decomposition Wrapper Application

6.1 Overview

This chapter is the core of the thesis. It describes the implementation of the preprocessing. Here the problems encountered as well as possible solutions in regard to computer-science aspects are discussed.

One major issue is the enormous amount of input data of real-world models which needs to be handled when performing calculations and program runs on a supercomputer or multiprocessor environment. How can such big models be stored on a computer effectively in regard to memory consumption or access time during computation? Where do tradeoffs have to be made between comfortable, easy programming and convenient handling with respect to the often enormous amount of data? Where is it advisable to rather think about performance and speedup and "forget" about the handling and programming? Is there a middle way? Which solution turns out to be the best? – These and other questions have to be considered and then answered when thinking about an optimal data structure for the input data. Ultimately, three different data structures seemed to be convenient in one aspect or another but just one of them turned out to be efficient and is now finally implemented in the current version of the wrapper program.

Storing the given input data is one aspect, but preprocessing it is another issue. Before the domain decomposition can be carried out, the given model data needs to be preprocessed so that it can be handled by necessary tools, like, for example, those described in chapter 5. Most likely, some sorting is necessary until the domain decomposition tools like *Metis* (see subsection 5.5, starting on page 112) or *Jostle* (see subsection 5.4, starting on page 107) can be used. How can this sorting be achieved most conveniently, also in regard to the chosen data structure? And after the preprocessing, what further and necessary steps have to be realized in preparation for the main program run of GS/RF (see chapter 4.2)? Answers to these questions will be dealt with in the sections following the discussion of the data structures and memory aspects.

It would be interesting to investigate the differences between the data fragmentation returned by *Jostle* and that returned by Metis. For this reason an optional statistic function was implemented. In this way, a convenient comparison between the tools and their results is possible. As it is also important to know where runtime is consumed during the execution of the wrapper, the author added a function which measures the processing time in different parts of the program. This

facilitates finding where to start the correct routines or parts of the program during the debugging process or in case of code optimization, for instance.

The last part of this implementation chapter deals with the commandline interface of the wrapper: How can it be used? What switches should be set, and which ones are just optional but quite convenient? - These questions will be answered in regard to the handling of the wrapper program.

As the problem of choosing the right data structures was one of the most important questions, this will be the one discussed firstly in the following sections after a short introduction to the problem of finding the "correct neighbor" of an element in a model.

6.2 How to Choose the Adjacent Neighbors

In section 5.3 of chapter 5, starting on page 103, the necessary conversion from a mathematical model representation to an adjacency graph was already introduced and explained. What was omitted there were the steps "in between," or the method itself, for deciding which elements in the original model representation are adjacent to others and which are not.

As the setting up of a corresponding adjacency graph and its implementation is obligatory for the usage of the aforementioned tools like Metis (see section 5.5) and Jostle (see section 5.4), this process is an important aspect of preprocessing. Therefore this short section will provide a closer look at the transitional steps in between.

In Fig. 5.12 on page 106 a simple example was already given: The mathematical presentation presented there had squares, and it wasn't difficult to spot which elements are adjacent and which ones are not. But now let us have a look at Fig. 6.1.

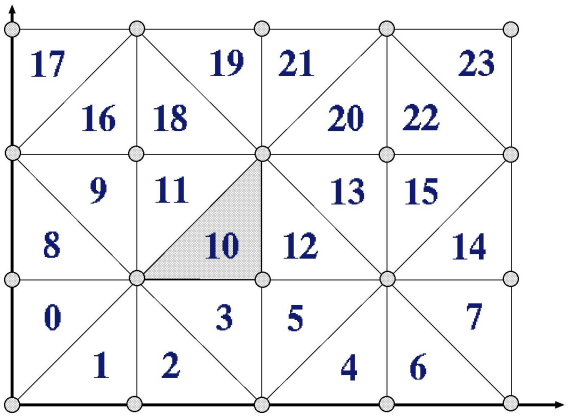


Figure 6.1: A mathematical representation of a simple model with 24 elements and 20 nodes with element no. 10 as the starting point for finding its adjacent elements

Let us assume that we are right in the middle of the process of setting up the corresponding adjacency graph of this example model in Fig 6.1 and that the next element we have to check for among its surrounding neighbors is element no. 10. Which of the adjacent elements should we now pick in this case? Those in its immediate vicinity are clearly the elements no. 3, 11 and 12. Element no. 10 and the three aforementioned ones all share one mutual edge. This approach for setting up the adjacency graph would lead to a result for the adjacent elements of element no. 10 as shown in Fig. 6.2.

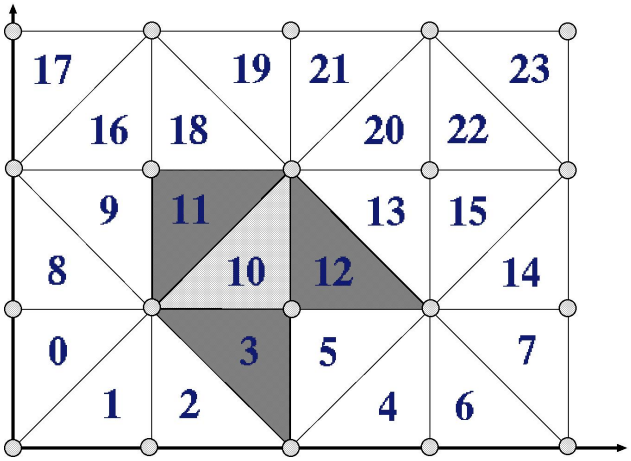


Figure 6.2: Elements no. 3, 11 and 12 are in the immediate vicinity of element no. 10

Now let us consider the implications of the foregoing: When discretizing our real-world problem as described in chapter 2, starting on page 9, each one of the elements will finally hold certain parameter values, like a material property or the hydraulic head, etc. When assuming, for instance, a continuous flux from one point to another within our problem domain and thus, also in our discretized model later on, it is very likely that such properties, like a hydraulic head, normally will not change by huge amounts within a given distance. We also recall that we tried to discretize our model in such a way that we still have a very good and clear picture of the actual situation that we are going to model without neglecting memory and runtime issues.

Now bringing all these aspects together in our discretized model or later on in its mathematical presentation, a hydraulic head, to stay with this example parameter, most likely just varies a little bit within certain ranges. We do have a definite value for the hydraulic head of element no. 10. Now, when examining its adjacent elements, their hydraulic head should not differ appreciably from the value given for no. 10. This holds even more when the elements no. 3, 11 and 12 share a long mutual edge with element no. 10.

So far the procedure for finding adjacent elements using a mutual edge seems to be working correctly. But on second thought, let us now have a long look at element no. 5. Its center point is definitely closer to element no. 10 than the rightmost node of element no. 12, which – unlike with element no. 5 – we designated as an immediate neighbor to element no. 10. And on a critical third look, we could now point out that if we assume in the real-world setting a gradual change in value from element no. 10 to element no. 12, the same would be definitely true for element no. 5 and therefore, of course, also for the elements in the model that describe the real-world scenario which we are going to model and compute.

This means that our first simple approach to assembling the adjacency graph should be optimized in such a way that also such “close” elements as elements no. 5 and no. 10 in the example would be detected, although they don’t share a mutual edge. The solution to this problem was already hidden in the previous example shown in Fig. 5.12 on page 106. There the direct adjacent elements to square no. 1 were given in the corresponding adjacency graph as elements no. 2 and no. 3. But when examining the model on the left-hand side of Fig. 5.12, we find that square nos. 1 and 3 do not share a mutual edge, although they share a mutual node. And this is exactly the correct solution.

When coming back to the example given here with triangular element no. 10, on our second try we are not going to concentrate on the mutual edges but rather on the nodes with which the element

is configured when setting up the adjacency graph. Let us take element no. 10 again: We start with the leftmost vertex of element no. 10 and check for all those surrounding elements that are connected to element no. 10 over this one vertex. The result of this search is depicted in Fig. 6.3.

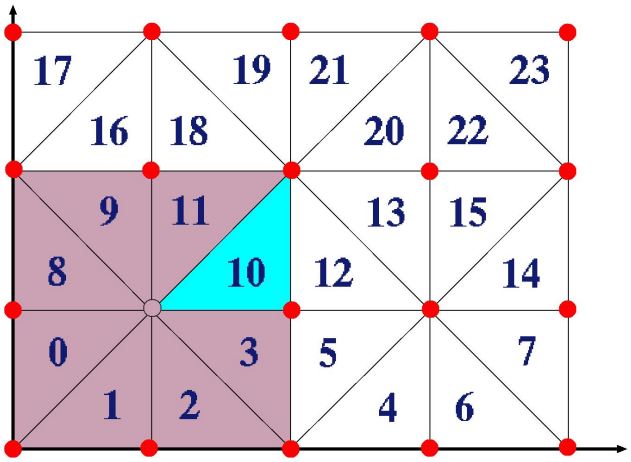


Figure 6.3: All immediate neighbors of element no. 10 connected over its leftmost vertex in the triangle

When proceeding the same way for the remaining two vertices of element no. 10 we will have to expand our list of immediate neighbors quite a bit, as Fig. 6.4 demonstrates quite clearly.

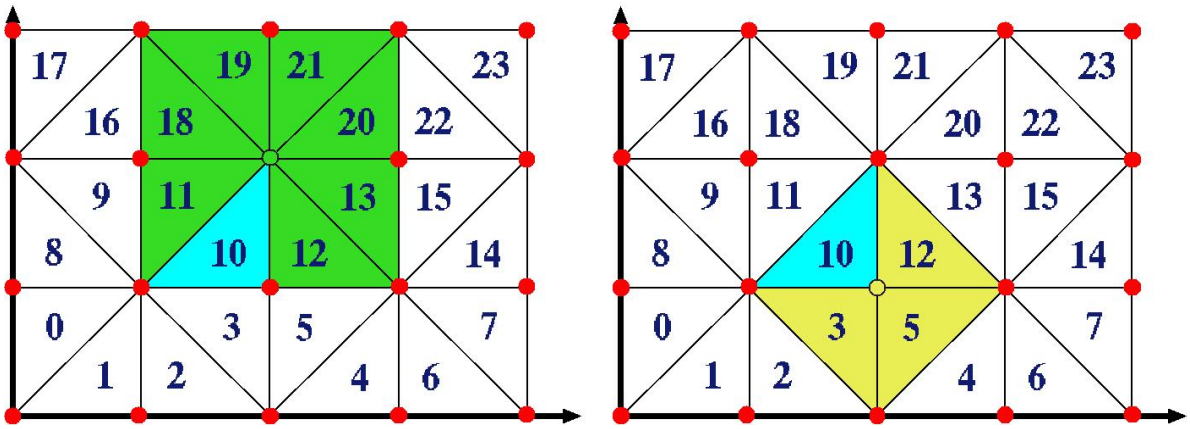


Figure 6.4: All immediate neighbors of element no. 10 connected over its upper and its rightmost vertex in the triangle

When setting up the adjacency graph of a model representation we have to proceed step by step as we just did for element no. 10 in our example model. After performing all necessary steps for element no. 10 and combining the resulting adjacent elements to it, we have the result which is summarized in Fig. 6.5.

As can be seen from Fig. 6.5, the list of neighbors of our element no. 10 does not consist simply of just 3 elements numbered 3, 11 and 12, as we had stated after our first approach, but indeed of 14 elements with element nos. 0, 1, 2, 3, 5, 8, 9, 11, 12, 13, 18, 19, 20 and 21. When recalling the small example model representation consisting of squares on the left-hand side of Fig. 5.12 on page 106 and now checking its adjacency graph on the right-hand side of the figure, we were satisfied, as this adjacency graph looks neither too complicated nor complex.

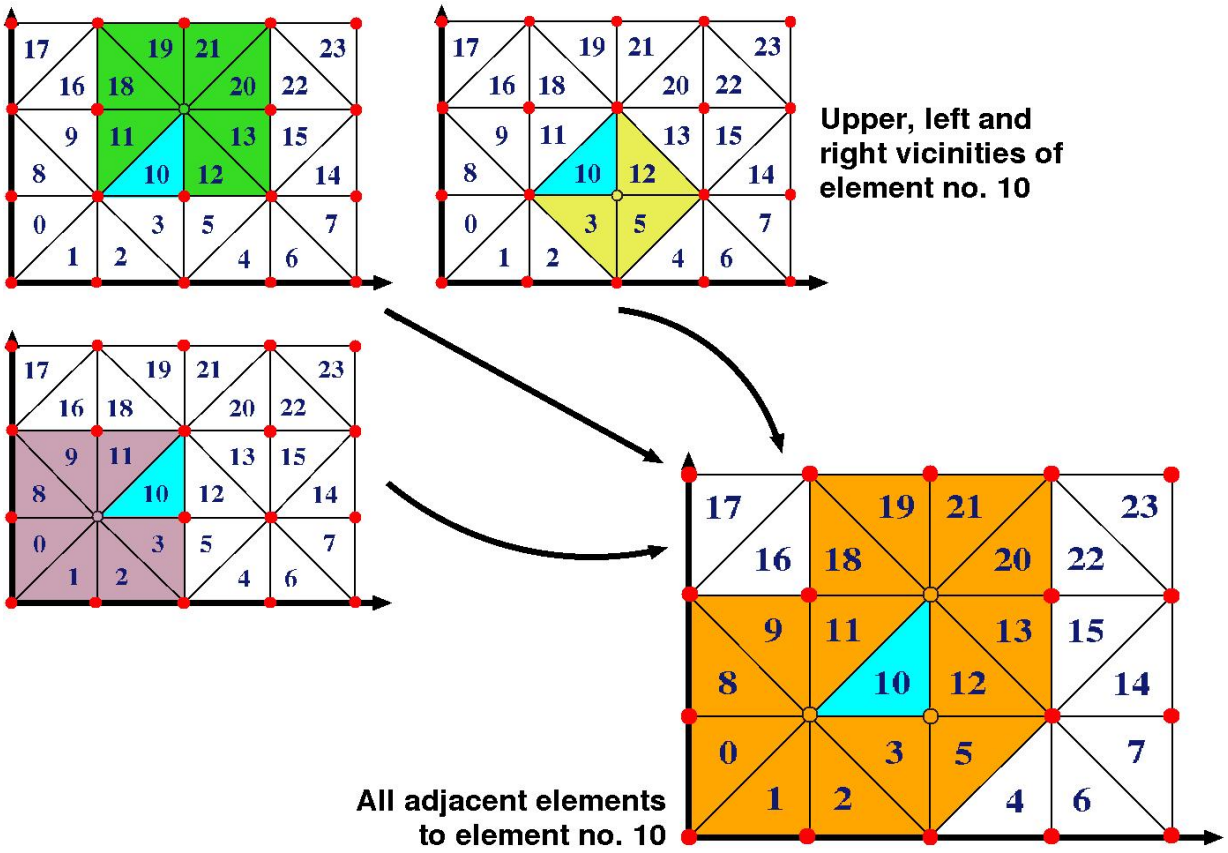


Figure 6.5: All immediate neighbors of element no. 10 which are connected over mutual vertices

When now setting up the adjacency graph for the equally simple model representation in Fig. 6.1 with triangular- instead of square-shaped elements, its resulting adjacency graph is already much more complex and a great deal larger than that for the model with the square elements. In fact, it is indeed so complex that it is already impossible to display it here without considerable effort. Why is this so?

The answer is quite simple and can also easily be seen by just comparing the two models shown in Fig. 6.6. Although they differ slightly in regard to the number of elements they contain, the model consisting of squares is obviously much less complex than the model with the triangles. The triangles are interlocking and notch almost into each other, whereas the squares are just simply stacked on top of each other or side by side.

The reason for the difference in complexity can directly be seen in Fig. 6.6: Whereas in the representation with the squares the elements always share edges and thus, at least two nodes with another, directly adjacent element, the vicinity in the presentation using the triangles is established simply over mutually shared nodes. For this reason, the stacking pattern in the model with the squares arises and the triangles are much more interwoven with each other.

Now why is all of this important enough for someone in charge of preprocessing to “waste” so much time on it? The answer is again very simple. During the preprocessing phase but before calling the domain decomposition tools, the model representation, stored in a file, must be processed. This means that it has, for example, to be read in in a first step. This necessary model information is given through geometrical information about the nodes and which of them form which element.

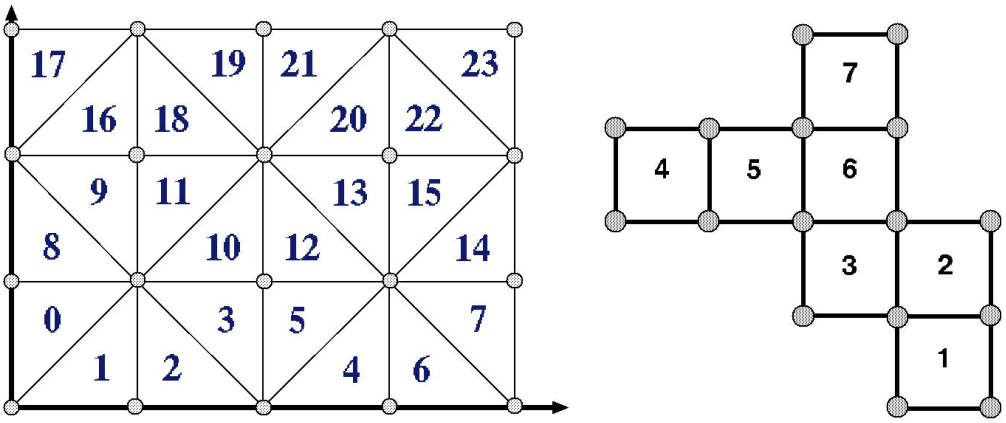


Figure 6.6: Two simple model representations, each consisting of differently shaped elements

For further processing of the elements, specifically this information about the constitution and shape of the elements is important and must be stored in an appropriate way. (This will be discussed in the following sections on data structures.) When now setting up the adjacency graph as one of the next major preprocessing stages, the programmer or engineer rarely knows in advance what shape the elements in the model will have – if they will be rectangular, triangular or even tetragonal, etc. They can even be a mixture of various different shapes. Such models are not only possible but actually exist. However, as was pointed out and demonstrated in the two previous example models with the triangles and squares, the complexity increases directly in dependance of the geometrical shape of the element.

The amount of information that has to be stored for an adjacency graph resulting from a model built of triangular elements is much greater than from a model with square elements. If even a small model with just 24 triangular elements like the one shown in Fig. 5.11 on page 104 is already so complex that it can't be easily displayed, what would such an adjacency graph look like if 1 million elements or even 20 million elements were now to be processed? Obviously, this problem is not trivial.

The original input data provided has to be stored conveniently so that it can be easily accessed. This also holds for the adjacency graph that must be assembled at some point during the preprocessing phase. Furthermore, all the auxiliary information necessary for setting up the adjacency graph or sorting the elements, etc. must be stored as well. If one is unconcerned about how this tremendous amount of information is going to be stored and then processed by using the appropriate data structures later on, he or she will definitely experience a rude awakening when computing bigger or really complex models. And one should not forget that this is just the preprocessing stage, to say nothing of what will happen if the choice of data structures used for the real computing stage was equally unfortunate!

In the upcoming sections we will have a closer look at the correct choice of a suitable data structure for such a model. But one should remember: Everything always depends on the real-world problem or its scenario and the resulting model itself. There is no general recipe. (This was nicely explained in chapter 5 on page 97, where a Formula-1 car was compared to an average, run-of-the-mill car.) Therefore, no matter what model or scenario is to be modeled and implemented at some stage, a thorough analysis of the input data and the intermediate data is definitely obligatory.

6.3 The Problem of Choosing the Appropriate Data Structure

Numerous different reasons can lie behind the decision to port a primarily sequential program onto a supercomputer or multiprocessing system. Often, one motive is the gain in processing speed when many complicated and lengthy mathematical operations are needed to successfully produce a result. Especially when specialized hardware like a vector computer (see section 3.5.2) can be used, necessary changes in the program directly lead to an improvement, but only if the code can be well adapted to the underlying vector machine architecture.

Quite another important factor for deciding to develop a parallel version of an existing sequential software application (see also section 4.4, starting on page 79) could be the amount of input or model data which needs to be processed. Still nowadays, despite the fast and powerful desktop computers and PCs, hardware limitations restrict the amount of data given, as was briefly mentioned in chapter 3, starting on page 25.

As a possible solution one could decide, for example, that the input model data should be reduced. This would then lead to a less accurate and more inprecise result of the computation of the model data and its equation system, but the program run could terminate in a reasonable time. But maintaining the original size of input information and parameters instead would result in program runs that process for weeks in a normal hardware environment. However, in respect to the computed model, the final solution would definitely be much more accurate. Up to which point would a less exact model be acceptable when at the same time the processing time could be reduced? What would be the maximum time that is tolerable for a program run and what effects does this have on the model and its input data? Depending on the scientific area, neither inaccurate models nor endless processing times are acceptable at all.

Especially when dealing with an enormous amount of input data in simulations or mathematical models, HPC often is the only solution. Here, a multiple of processing units but also generally better, faster and bigger hardware peripherals and processing components can be found. But such advantages can easily be offset by inefficient and inappropriate data structures and data handling. Furthermore, working in such an environment requires sophisticated knowledge; nor should one forget that computing time and usage of the HPC hardware costs money!

If it is possible to divide the model or simulation data into independent parts without affecting the final result, as discussed already in chapter 5, starting on page 89, demanding scientific computations can suddenly become solvable. A very typical but good example for this is the weather forecast problem which was briefly described in the motivation chapter (see page 1).

Even when simulations might suddenly be possible through fragmentation of the model data, there is still the problem of how to store this amount of information. In the end, 10 or 20 million elements of a model simply remain 10 or 20 million elements, no matter whether the input data could be "chopped" into smaller chunks or not! In many cases where such complex mathematical computations are necessary, the model information has to be computed all together, but through fragmentation it can now be spread among several processing nodes during program execution instead of having one "poor node" compute the complete model alone. And yet, even if the data can now be distributed, these 10 to 20 million or more model elements have to be read in and processed; and most likely a similar amount of information has to be written back and stored somewhere (see also section 4.3 on page 74 in chapter 4).

When working in a multiprocessor environment, how the data is stored and in which way it is accessed is very important! This aspect is often completely neglected by the average software

developer working on simple desktop computers. Programming techniques and data structures which seem very convenient and favorable in a desktop programming environment could have very adverse effects on a HPC architecture. For instance, *pointers* and *structs* in C programming can be taken here as a simple example. Such data structures are common in C and are found throughout an average C program in general. They often make things easier during programming, and elegant C code can be written this way when applied reasonably.

On a multiprocessor machine, fast data structures like *arrays* which can be read in quickly into the caches for processing without pointer dereference are preferable. This holds all the more if a great deal of similar but simple computations have to be processed, like integer operations. Supported by hardware on these specialized computers, the arrays are "slurped" one after another into the internal register banks of the individual processing nodes and are available directly for the computation and the specialized hardware components without any additional preprocessing, unpacking, pointer resolution, etc. Were the data stored in *structs* instead, cumbersome and painstaking pointer operations would now be necessary. In addition, possible speedup would be offset by ineffective additional internal operations, required for the pointer resolution but normally hidden from the software user.

One could say now that this extra work takes so little time nowadays, even on supercomputers. This is completely correct, but is only true when dealing with a few hundred input elements on maybe one processing unit, for instance. But if millions of input elements are spread on various processing nodes, this additional little bit of time for extra internal hardware-related operations and administrative work suddenly becomes measureable! If the main reason for porting the primarily sequential program onto a supercomputer was originally to cut down processing time and increase speedup, such inappropriate data structures could ultimately foil all former efforts!

Let us take a simple example and assume that it takes 1 ms for 50 real numbers which are stored in a simple data structure to be processed by a specialized pipelined hardware component. At the same time, 10 real numbers, stored in using pointers and structs like in C, are processed within 0.5 ms. All operations are done on the same machine. If 1000 real numbers are now to be computed, it takes 20 ms for the version using the simple data structure and already 50 ms for the version with the pointers. When processing 100,000 input values, the fast pipeline variant with the simple structures only needs 2 s, whereas the pointer version now runs for 5 s. As modern data models have millions of input elements, an assumption of just 1 million real numbers in our simple example seems realistic. With the simple data structure, 1 million numbers would be computed in 20 s for just one command but they would run 50 s with the pointer structure. If one takes into account that a program does not consist of just one but of countless computing steps and of additional loading and storing operations which also consume time, it quickly becomes clear on the basis of the above example containing only a few numbers that wise decisions with respect to a suitable and fast data structure are necessary.

When choosing a data structure, another important aspect is memory consumption. Arrays are normally stored in consecutive and contiguous memory locations, whereas linked lists (see also section 6.4 on page 128) are quite often distributed within the memory. To access the list elements a little extra memory space for the pointer to the following list member must also be kept. If doubly linked lists are used which implement pointers not only to successor the but also to the predecessor, even more of the memory space is consumed. And again there is a difference between storing just a few hundred and a thousand elements of a data model or even 20 million elements. Storing 20,000,000 additional pointers in the memory during processing or just keeping the model information in a simple array without any overhead will definitely have consequences during program execution.

Again a simple example: We assume a 32-bit architecture, which means the word length on this computer is 4 bytes. Everything being stored on this machine consumes 4 bytes of memory automatically, even if it just needed only a couple of bits. If our model now consists of 20,000,000 elements and each of them is stored in 4 bytes from the very beginning, an array which holds all this data would then occupy directly 80,000,000 bytes, which is approximately 80 MB (depending on how one "defines" 1 MB, see also table 6.2). Using a linked list instead, each of the 20,000,000 elements of the model needs at least one additional pointer in a simple linked list and two pointers in a doubly linked list. A pointer also allocates 4 bytes in the memory so that the simple version of the list would immediately consume roughly 160 MB of memory and a doubly linked list, about 240 MB. As some old laptops sometimes have just 256 MB of RAM, just the storing of the lists alone would require half or most of the memory (depending which type of list had been used) without having computed one single number of the model.

On the other hand, arrays might not always be a good solution! Within a sparse matrix, for example, there are hardly any non-zero values. In such a case it would not make any sense to store information in a 2D array if finding the position of a non-zero matrix value takes more processing time than resolving the required pointers when storing the small amount of non-zero information of the matrix in a list instead. (This is just an example. There are of course many special data structures for sparse matrices and other cases which can be implemented, depending on the situation and the problem to be coded.)

Which data structure should be implemented in the end depends mainly and very closely on the problem which has to be computed and solved by the code and on what type of input data or temporary data has to be processed. Therefore it is necessary to analyze such questions thoroughly at the beginning of the coding process. Especially when exploiting HPC architectures, the convenient and elegant data structures might not always be the best ones (an unfortunate fact that especially typical desktop programmers often forget or ignore!).

At the time of this thesis it was originally planned to process finite element (FE) models up to 20 million elements and even more in the near future. These need to be preprocessed and sorted before they can be finally split into different submodels using special domain decomposition tools as discussed in sections 5.5 and 5.4 of chapter 5 starting at page 89. Not only must the elements themselves be read in and processed but also additional auxiliary variables and further information must be stored and computed for the realization of the preprocessing.

In the following sections different data structures are analyzed in regard to issues related and important to the model data. The first data structure, which was implemented in the first version of the wrapper program, were so-called bit arrays. It turned out that they did have some serious drawbacks despite the fact that they are fast and simple data structures. When analyzing the problems encountered, lists of structs seemed to be a second interesting approach to the given general conditions. But after closer examination they dropped out of the list of possible candidates of suitable and appropriate data structures. The third and last data structure, which is now implemented in the current version of the wrapper, are dymically allocated arrays.

For reasons of readability and also logical dependencies the order of the different data structures presented here will be changed somewhat in regard to their timely application. The list structure is introduced first. Next to follow are the bit arrays, which already show clearly the major benefits of an array structure. But they also have certain serious disadvantages. The discussion of the different data structures will be concluded by the dymanically allocated but simple arrays. (The comparison between the different data structures and their advantages but also drawbacks in the situation of the given model data of GS/RF will follow in the upcoming chapter 7, starting on page 201.)

6.4 Linked Lists

In section 6.2, beginning on page 120, the process of how to find the adjacent elements to one particular element was already introduced and described. When we recall the example given there using element no. 10, 14 different elements were finally selected to be stored in the immediate vicinity of element no. 10. A closer look at the listing of these 14 resulting elements with the numbers 0, 1, 2, 3, 5, 8, 9, 11, 12, 13, 18, 19, 20 and 21 reveals that the element numbers always occur just once. This is due to the fact that each element is unique in the model and its representation. As another consequence, the adjacent elements to one particular element can be sorted, either in an increasing or a decreasing order. Unfortunately there might be gaps in the consecutive list of numbers, as the neighbors of element no. 10 nicely demonstrate.

There is another fact that is hidden in the geometry of the element type. As an adjacent element to a certain element is connected over a mutual node, there is only a limited number of elements that can be attached to each other this way. The reason is simply the amount of space that the elements can consume. In section 6.2 the element type was a triangle in 2D, whereas in Fig. 5.12 on page 106, squares were used, also in 2D.

Even without any profound mathematical or geometrical analysis, it is directly clear by just examining either such a triangular element in the example in section 6.2 or one of the squares in the example in Fig. 5.12 that after having placed a certain number of elements around one triangle or one square, the available space around this geometric element is consumed and completely filled by its surrounding adjacent elements. This means the list of immediate adjacent neighbors to one element is finite. That is also true for all other kinds of elements, no matter of their geometrical shape or appearance.

Now adding all these given facts together, one of the first data structures that immediately appears suitable to the "inner eye" of a computer scientist in such a situation is a so-called *linked list*. In the following subsections the basics of such linked lists and their implementations are briefly introduced. A short discussion of their pros and cons will conclude this section about linked lists.

6.4.1 The Basics of Linked Lists

A linked list is one of the fundamental data structures in computer science. It is a simple *self-referential datatype*, as such a *list node* or *list element* contains always at least one pointer or one link to another datum of the same type. Thus, a linked list is a typical data structure in which the objects are arranged in linear order. It consists of a sequence of elements, each of which contains arbitrary data fields and at least one reference, the *link*, that points to the next node in this list. The linked list gets its overall structure by using pointers to connect all its nodes together like the links in a chain. Fig. 6.7 shows an example section of a simple linked list.

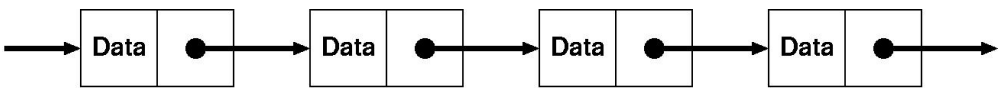


Figure 6.7: A section of a simple linked list

As linked lists are a very uncomplicated yet often effective data structure (this depends strongly on the given situation), there are programming languages where such lists are already directly

incorporated as a built-in data structure including the operations that are used to access and work with them. Such languages are, for instance, Lisp and Scheme. In other programming languages, often so-called procedural or object-oriented programming languages like C, C++ or Java, linked lists are set up over mutable references.

A list always has a *head* and a *tail*. The head is the beginning of a list and is generally a reference that points to either the first list node if the list is not empty, or is set as an "empty reference" which is a `NULL`-pointer in C or C++, for example. The tail is the last element in a list and its reference can be set to `NULL` or some other value (see the different types of lists below).

6.4.2 Types of Linked Lists

There are various types of linked lists. The most simple one is the *singly linked list*, as shown in Fig. 6.8. In such a list type there is just one reference per list element that points only to the next node in the list, its successor.

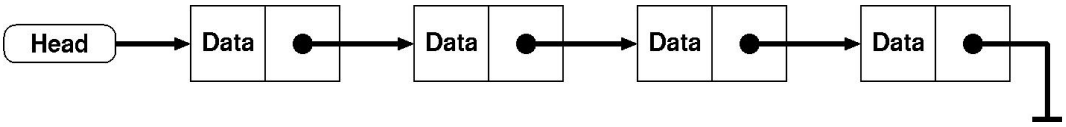


Figure 6.8: A simple singly linked list consisting of 4 list elements

If there is no successor or in case of an empty list, the reference is set to `NULL` as an empty reference; see Fig. 6.9, where such an empty list is shown.

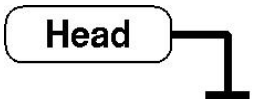


Figure 6.9: An empty singly linked list with just an "empty reference"

In a *doubly linked list* each list element holds two links: One link points to the previous list element, the predecessor, and the second one to the successor. It can be compared to two singly linked lists that have been fused into one list, where each of the pointer references in the singly linked lists had been pointing in the opposite direction. A simple doubly linked list is depicted in Fig. 6.10.

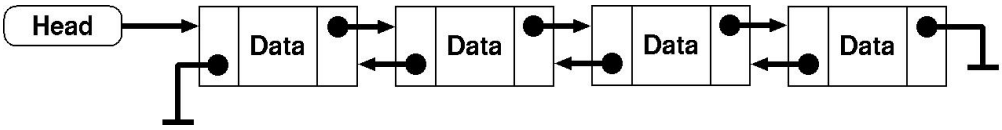


Figure 6.10: A simple doubly linked list consisting of 4 list elements

The doubly linked lists are also sometimes called *two-way linked lists*. In any case one has to take special care with empty doubly linked lists or when working with the first or last list node in such a list. The singly linked and the doubly linked lists are both *linearly linked lists*. Unlike the latter, the *circularly linked lists* do not have an explicit end or beginning once one is moving within such a structure.

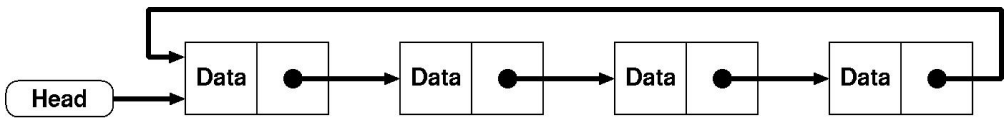


Figure 6.11: A singly circular linked list consisting of 4 list elements

In the *singly circular linked list* shown in Fig. 6.11, the reference of the tail node which would be set to an "empty value" in a singly non-circular list is pointing to the very first element of the list again. This way a ring structure is created in which one can move endlessly in one direction. Similarly in a *doubly circular linked list* the same applies but here the back reference in the first list element is pointing to the final list element and vice versa. In such a circular list one can move now in both directions with an end. This is depicted in Fig. 6.12.

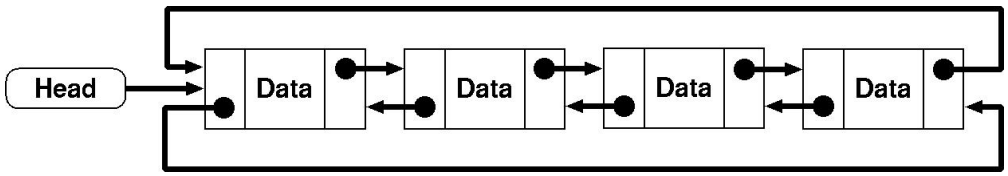


Figure 6.12: A doubly circular linked list consisting of 4 list elements

No matter whether using a singly or doubly, circularly or non-circularly linked list, the lists can be either sorted in regard to a certain aspect (like increasing or descending value, for example, when using integer values in the data field) or it can also be unsorted. This depends on the situation and reasons for choosing and implementing such a list.

In the remainder of this section, we assume that the lists with which we are working are sorted and linear singly linked.

6.4.3 Basic Operations on Linked Lists

Regardless of the problem to be solved by using linked lists and the type of list implemented, there are certain basic operations on this data structure that are necessary in order to work with it. These elementary procedures are explained now briefly without going into great detail.

To set up and/or to increase a list in size the `insert` operation is needed. In case the list does not yet exist when the first list node is inserted, the head of the list, which means a special reference, must be created and the first and only list element is inserted by simply having it referenced by the list head. Once the list exists, the list element to be added just has to be placed at the correct location in the list in which it should be inserted. To add a new list node the appropriate references of its predecessor and its successor need only be corrected. Such an operation is demonstrated in Fig. 6.13, where a fourth list element is to be inserted between the node containing the value "B" and the one with the value "D."

It has to be noted here that each single list element always has its own reference by which it is accessed, so that this node can always be reached and used. In C or C++ this is done over a simple pointer. It is the only possibility to access such a list element. The very second it is smoothly integrated into the list structure, its own reference pointer can be neglected, as its list predecessor is now pointing to it and thus, it can be reached any time over this reference.

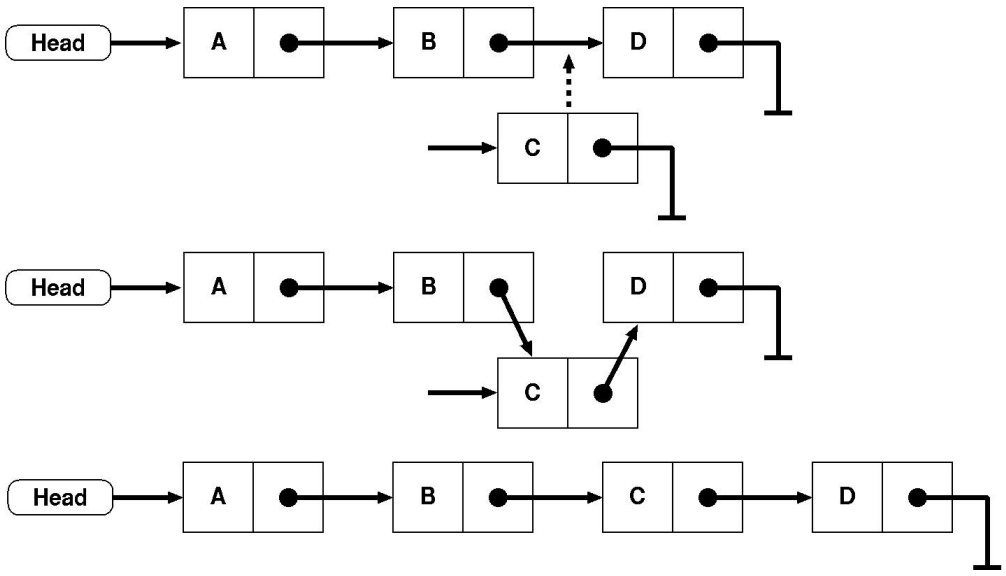


Figure 6.13: A fourth list element is to be inserted into a singly linked list

The other important operation on linked lists is the `delete` operation. Instead of adding another list member, one is now taken out. In case it was the very last node existing in the list, the head reference of the linked list structure must be set to an "empty reference." The standard steps in case of a node deletion are depicted in Fig. 6.14.

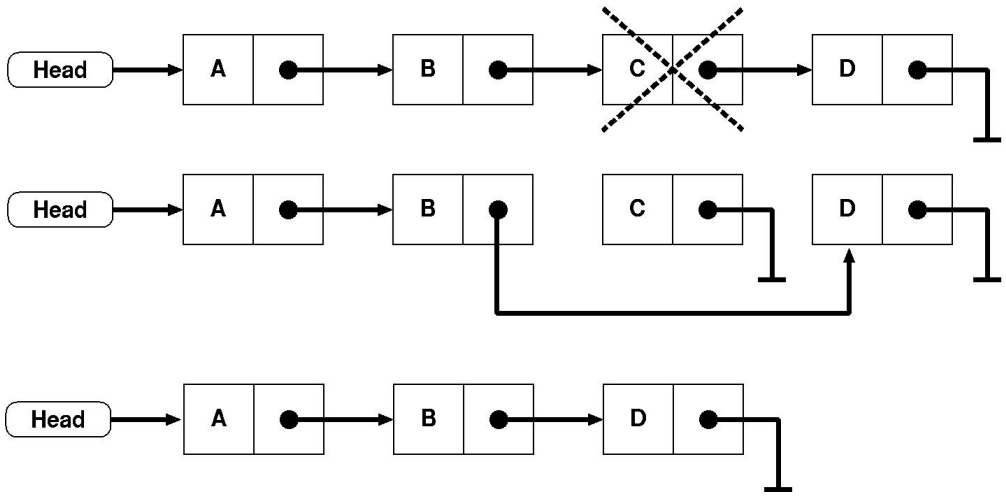


Figure 6.14: A list member is taken out of a singly linked list which automatically decreases in size after the deletion

Both operations, `insert` as well as `delete`, are modifying operations. There are other basic operations in addition to the two most important ones introduced above. A non-modifying operation is the traversing of the list element by element. This is an essential procedure when searching a special data value within the list nodes. As an ordering of the data is not obligatory when working with linked lists, it is more than likely that there will be situations when information stored in the list structure must be retrieved. It can only be found by searching the list elementwise from beginning to end in the worst case. For the traversing, one has to check each reference in the list element to find the successor node until a certain value has been found or the end of the list is reached.

Other interesting operations on linked lists could be, for example, finding a minimum or a maximum value stored in the data field in a list element. But a closer look reveals that this is nothing more than just a simple traversing operation combined with a search over the list nodes. It is also possible to link together two different lists. Either one can be appended at the end or beginning of the other or one list can be split and the other one can be linked in between the gap. And here again, we still have similar operations to be performed. In case, for instance, one list is to be appended at the end of the other one, this simply involves having the last reference of the first list no longer point to `NULL` but to the first list element of the second, the appended list.

Linked lists are one of the basic data structures, no matter whether they are singly or doubly linked or linear or circular, and can be applied in various situations to solve given problems. Further information about this topic can be found in almost any basic computer science algorithm (text)book. A good start would be, for instance, [Darnell] for complete beginners of this topic, or either [Cormen] or [Sedgewick01], one of the "bibles" in the wide and interesting field of algorithms.

6.4.4 Implementation of Linked Lists

When working with linked lists in C it is obligatory to know exactly which operations using so-called pointers are possible and allowed, and what problems can arise. It is also necessary to have an understanding of how a memory is organized and how pointers are handled and stored there in comparison to other data types. Most likely a non-computer scientist or a non-C programmer might not have this knowledge; therefore, a brief introduction to a computer memory and its organization will be given.

Computer Memory Organization and Access

Anyone who has worked with a computer knows that generally the hard disk, nowadays equipped with gigabytes of storage capacity, is responsible for permament storage. Furthermore, one will probably have experienced a notable slowdown of his or her computer when its built-in cache (normally just a percentage of the size of the hard disk) is too small. The question now arises as to which of the aforementioned storage devices one actually is working with when using pointers. Is it the hard drive or is it the faster but very small cache?

To add to the confusion, the answer is: neither of the two. The reason for this is that there is normally a special device within the computer responsible for all memory activities, called the *memory managing unit* (MMU). It can be either implemented in software or in hardware. But what is its function?

The answer is actually quite simple: The MMU helps both the user and programmer of a computer. In section 3.2 of chapter 3, starting on page 26, the principle of a cache hierarchy was introduced. When combining different fast and slow components, i.e., the registers in a processor, a L1 and L2 cache and a hard drive, the user or programmer can no longer organize and manage them efficiently, no matter what the computer's size may be.

The MMU takes care of the necessary hard drive accesses as well as writing back register or cache entries, etc. It sits "in between" the CPU and the other devices. Describing the complete tasks and organization of the MMU in full is far beyond the scope of this thesis. Therefore just a simplified view will be given. All interested readers should have a look at [Patterson], the so-called "bible" of computer architecture.

To both the CPU and computer user, the storage space, simply called “the memory,” appears to be an “endless” storage space starting at a virtual memory address of 0 and going up to kilo-, mega- or gigabytes, depending on the specific case. One could picture this as long row of equally sized storage boxes. All the boxes or the storage locations in the memory have a basic size, which depends on the computer or processor architecture with which one is working. Nowadays, this is either 4 bytes on a 32-bit architecture or even 8 bytes on a newer and more modern 64-bit architecture.

If a datum that needs to be stored is smaller in size than the minimum size of the storage space in this virtual memory, the leftover space is “padded.” If something is bigger in size, the datum will be distributed consecutively in a certain way. How this is accomplished is not important here. The MMU takes care of such things, too. Fig. 6.15 shows a simplified view of a virtual memory 2000 bytes in size.

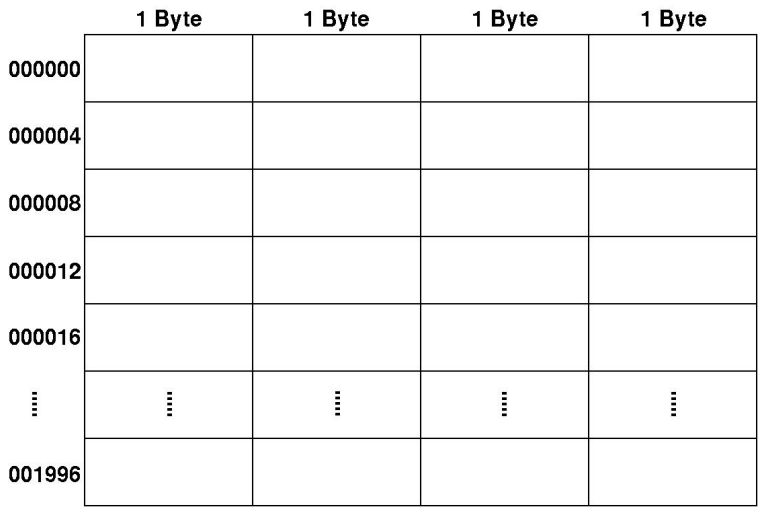


Figure 6.15: A virtual memory with a size of 2000 bytes, organized in 4-byte blocks on a computer with a 32-bit architecture

As can be seen in the picture, the virtual memory starts at the virtual address of 0 and ends at an address of 1999, which is exactly 2000 bytes when starting to count at an address of 0. The next thing to note is the fact that the memory is not organized in byte size but in the aforementioned blocksize of 4 bytes on the 32-bit architecture used in Fig. 6.15. When something is now to be stored in one of the memory locations, only addresses with a multiple of 4 can be used, i.e., 0, 4, 8 and so on. The last possible memory address in the example depicted would be 1996. On a 64-bit architecture everything would be the same, only that there are now 8 bytes in a row instead of 4, as in Fig. 6.15. On a 64-bit architecture the virtual memory would also start at the virtual address of 0 but then the next address would be at 8, then 16, etc., as one needs a multiple of 8 in this case.

Whenever something has to be stored in the memory during the execution of a program, it always will be stored within this big block of virtual memory. On what device and where the information is then stored there in reality, is hidden from the user and the CPU and is performed in the background by the MMU.

This explains why some list elements might be stored contiguously and consecutively on the real storage device while others might be stored separately and singly at some other location. For the user and the CPU this is neither evident nor visible.

References in General and Pointers in C

In subsection 6.4.1, on page 128, linked lists were introduced as a so-called *self-referential datatype*, which is indeed a perfectly fitting name for this kind of data structure, as their strength lies in the usage of references.

A linked list is a data structure, which means its elements hold values that are stored somewhere in the virtual memory. A reference in regard to the memory could be understood as something similar to the address of a house, for example. A reference is therefore a small identifier from which it is possible to find a potentially much larger object.

A pointer in C or some other programming languages is nothing but a simple implementation of a general reference data type. This means that a pointer is a data type whose value refers directly to or "points to" another value that is stored elsewhere in the memory by using the address where this other value is stored. Hence, a pointer stores just the address of an object in the memory but not its true value. Obtaining or requesting the value of the object referred to by the pointer is called *dereferencing* the pointer.

To the reader unexperienced in programming this approach might sound rather complicated and nonsensical. In fact, the usage of pointers can result in a very fast and elegant code and improvements in performance, especially for repetitive operations such as traversing strings or tree structures. The following example will hopefully demonstrate the power of pointers.

The pointer was compared to the address of a house. Finding this house and its inhabitants on the basis of this address is analogous to dereferencing the pointer. Now suppose one leaves a forwarding address in the old house each time he or she moves. A person who wishes to visit the person who has moved and has just the old address, could start out at the first house and then simply follow the forwarding address to the next house, and so on, until the visitor finally finds the current address. This is exactly how pointers are used when working with linked lists.

Now let us assume that one wants to be able to easily locate the people in his or her street on the basis of their last names. One completely impractical way of doing this could be by using a large crane and physically lifting up and then rearranging all the houses on the street according to the last names of their residents. A much more practical and also easier solution is to make a list of the addresses of the people living on that street and then sort this list by the last names of the residents. When using pointers and working with them as references, the programmer does the same: He or she is now able to manipulate references to data values in the memory without actually having to modify the stored data itself. Depending on the circumstances, this method can be much more efficient than working on the values directly.

Unfortunately, as elegant as the code produced by using pointers might be, their application can be just as dangerous as elegant. In many programming languages pointers allow largely unprotected access to memory addresses, and the risks and damages caused by false pointer dereferences or memory corruption resulting in program crashes and/or incorrect computations are high. Even worse, such types of program bugs are generally very difficult to track.

In Fig. 6.16 an example is given of how a singly linked list could be stored in the virtual memory. In fact, the list taken as an example here is the upper-most list in Fig. 6.13 with the three list elements, where a fourth element is to be inserted. This example demonstrates that all of the list elements can be located anywhere in the virtual memory. The single list members need not necessarily have to be stored consecutively or in the right order, thanks to the use of the pointers that link the list nodes to each other no matter where they are found within the memory. In case a

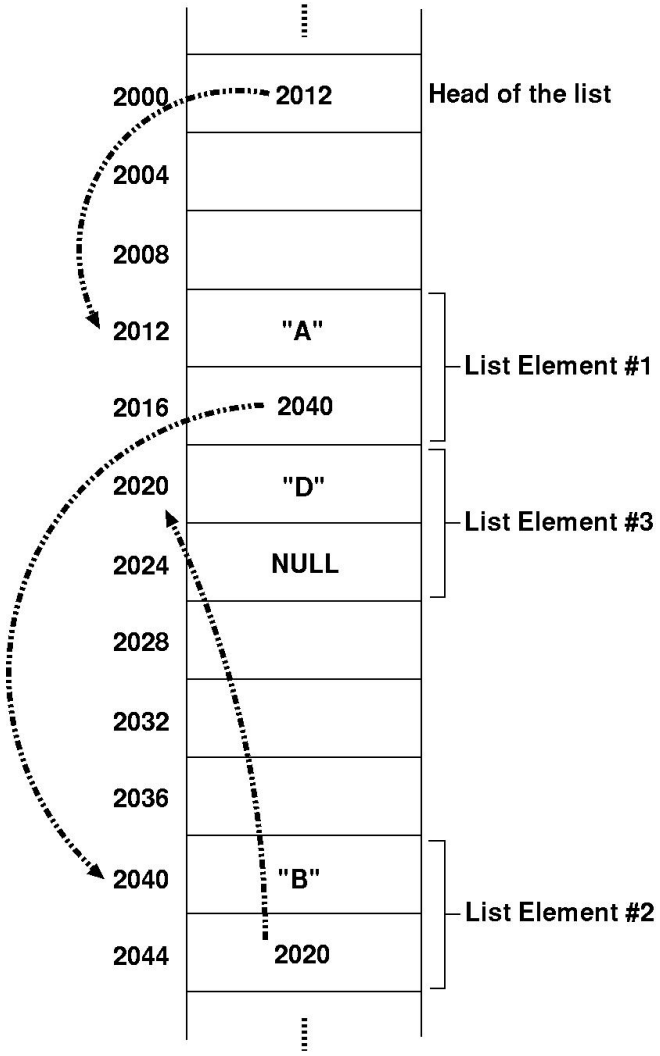


Figure 6.16: An example of how a short singly linked list could be stored in the virtual memory

fourth member with a data value “D” is now inserted, as in Fig. 6.13, it could, e.g., be stored either in the free space between list element no. 3 and no. 2, as the first element being followed by list element no. 1, or at a completely different memory location.

Should we now wish to delete the second list node of the linked list in Fig. 6.16 with the value “B,” it is obvious how simple this is. One must only correct the pointer value in the first list member with the value “A” which points to address 2040 by changing the stored address from 2040 to 2020, at which location the remaining list element with the “D” can be found.

6.4.5 Linked List in the Preprocessing of the Model Data

At this point all necessary information has been provided to focus on the implementation and usage of linked lists in regard to preprocessing the input model data. The problems encountered in preprocessing, no matter what the underlying applied data structure and their solution are, are not of interest in this subsection. They will be discussed separately in one of the following sections, as they are experienced independently of whether linked lists or another data structure are used.

When examining the general structure of the input data files shown in Appendix D, starting on page 257, it is immediately apparent that a linked list is needed to hold the node information for each of the elements setting up the model. Here the problem of the varying number of nodes per element has to be solved, as this number depends on the geometrical shape of the element.

Setting up these lists is straightforward. When reading in each line with specific element information, the corresponding list per element with the node numbers can be set up directly. As the node information provided per element is given in sorted order, the emerging linked list over the nodes is also sorted.

It needs to be noted at this point that not only the list which has to be created for the node information, but also those that will result from later stages in the preprocessing can be either singly or doubly linked linear lists. This depends on the personal gusto of the programmer. As the handling and administration for the doubly linked lists is much more complicated and implies more effort in its usage, singly linked lists should be preferred, as there is no necessity for doubly linked lists here. The application of circular lists, be they singly or doubly linked, generally does not make sense in this case, no matter which stage of the preprocessing process is regarded.

After having read in the input data the next major stage in the preprocessing process is setting up the adjacency graph using the information provided and stored up to this point in the program. When recalling the algorithm of how to find the vicinity of a certain element in the model introduced in section 6.2, starting on page 120, one realizes that up to this point we have dealt with per-element information. Now, in the process of finding the adjacent elements to a certain element, we need additional information on a node-to-element basis as well as on an element-to-element relationship basis.

Here, on first sight, a linked list again seems to be a suitable solution because just a finite number of elements which are directly adjacent to another element has to be stored, and because just a certain finite number of elements share one single mutual node. The information that is going to be gained and needs to be processed in this stage of the preprocessing is not necessarily sorted. So the responsible programmer should try to set up and fill the lists directly in some sorted manner, either with increasingly or decreasingly ordered values.

The last stage in the preprocessing of the input data where linked lists could be a possible and suitable solution is after having called the domain decomposition tools when processing their returned information. In accordance to each of the resulting subdomains, the corresponding border nodes and inner nodes have to be determined and printed out. The structure of such an output file, a so-called `ddc-`file, is shown examplarily in Appendix B, starting on page 241. As the domain decomposition tools work upon the basis of elements but the returned information is focused on the domains, new lists for each of the domains have to be created where the corresponding inner and outer nodes must be stored. It is reasonable to also immediately order the stored node information correctly when setting up these lists. A linked list comes in handy here, since, like in the situation of setting up the adjacency graph, the number of inner and outer nodes per domain which has to be stored is finite.

It is clear that in addition to the various lists essential to the preprocessing, many more different auxiliary data structures are needed. They may also be linked lists, but also other basic and common data structures known in C can be used here. As these very specific implementational details do not add in any way to a further understanding of the implementation of linked lists, the reader will be spared these further particulars.

The setting up of linked lists, adding more list nodes, etc., are done exactly as described in sub-

section 6.4.3, starting on page 130. A full discussion of the benefits but also the major drawbacks when implementing linked lists in the preprocessing stages will follow in chapter 7, starting on page 201, where the different possible and suitable data structures for the preprocessing will be compared and analyzed with regard to their general suitability in the given situation of model data preprocessing.

Through the analysis of the different preprocessing stages and during the implementational process, it soon became evident that linked lists unfortunately are not the most suitable data structure, and that others better adjusted to the given data and existing prerequisites were needed. Alternatives are the subject of the next large sections of this implementation chapter, where not only arrays in general but also more specialized types of arrays, bit arrays and dynamically allocated arrays, are introduced and discussed.

6.5 Array Structures

In the previous section *linked lists* were described and analyzed in detail as one possible and well-suited data structure in this particular case. The other alternative is not dissimilar from such linked lists when looked at from an experienced programmer's point of view. Indeed, many C programmers even consider them to be identical: We talk about pointers (that are used to set up linked lists) and the classical, simple *array* in C.

The following sections and subsections delve into the various aspects of arrays in general as well as into two very specific types of arrays, called *bit arrays* and *dynamically allocated arrays*. The comparison and discussion of the advantages and drawbacks of the three possible data structures will follow in chapter 7, section 7.2, starting on page 209.

6.5.1 The Basics of Arrays

In subsection 6.4.4, on page 132, the basics of computer memory and its organization were already introduced and explained. The storage area which is "visible" to the CPU and to the user is nothing but a *virtual memory* which is administrated in the background by the MMU. It can be compared to an "endless" row of storage boxes that all have the exact same size. Depending on the underlying computer architecture, this "box size" (which is referred to as the *word size* of the computer) is either 4 bytes on a 32-bit architecture or 8 bytes on a 64-bit architecture. But still, each of the single bytes in this virtual memory, simply called the "memory," has its own memory address although they are normally organized and accessed in bytes of 4 or 8, depending on the word size. Fig. 6.17 "zooms" into a virtual memory showing memory addresses that range from 924 to 940.

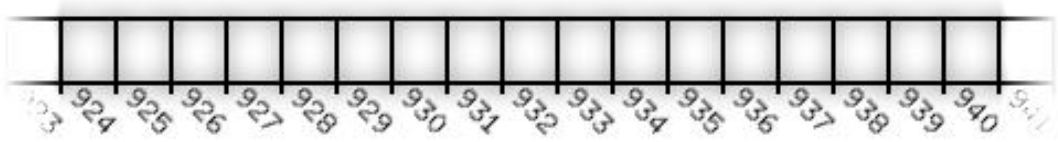


Figure 6.17: Excerpt of a computer memory ranging from memory address 924 to 940

For the following text, we will assume a word size of 4 bytes (but the reader can easily transfer the examples to a word size of 8 bytes). When starting to count the addresses in the memory from address no. 0, the next valid memory location would be at 924 in Fig. 6.17, as this is the next location that is divisible by 4; this is obligatory when working on a 32-bit architecture. (See also Fig. 6.15 on page 133.) The last valid memory address in the excerpt shown in Fig. 6.17 is no. 936. (Address no. 940 is also a correct memory location according to the word size but it no longer completely fits into the memory excerpt depicted.)

Now let us leave the memory for a second and think about a simple but daily situation: We are interested in the development of the temperature during the year, measured every day at noon. Depending on the accuracy of our thermometer we most likely will end up with 365 floating-point numbers that have to be stored.

Should we now wish to process these temperatures on a computer, we could "simply" define a variable for each day when a measurement was taken: `january_01 = -1.2;`, `january_02 = -2.7;`, and so forth. It quickly becomes evident that this might not be the best way to store the values, as now several similar commands in the computer program lines have to be coded for each of the temperatures, if they are to be processed further. This is neither efficient nor fast. Furthermore, with every additional line of code, the likelihood of all kinds of errors will increase statistically.

We already learned about linked lists in the previous section. So why not use this data structure instead? It is definitely a more practical approach to the given problem for processing the temperatures than coding 365 single variables. This would allow us to start at the head element of our list of temperatures and walk through the list elements, step by step, repeating the necessary C commands and simply exchanging the list node with the temperature.

Still, this might not be the most efficient nor best solution, depending on the given situation and what we want to do with our measured temperatures. We need to keep track of the location of the pointer; we have to make sure that we are not "overstepping" the end of the list, and we always have to find the next valid list element by evaluating the pointer information. All these single, extra steps from one list element to its successor take precious time and additional coding lines.

The most convenient solution would simply be to store all the measured temperatures one after another without any extra overhead for accessing them, as the processing routines are the same for each of the values. And this is where the data structure called an *array* now comes in handy. When we store such a long row of 365 temperatures in an array, they are literally placed successively in one of the "storage boxes," (still keeping in mind the organization in the memory, based on the word size). Fig. 6.18 shows an example how the memory might then look.

In C, an array is a collection of variables of the same type which are stored at a memory location in a contiguous and consecutive way. Each of the variables in an array is called an *array element* and can easily be accessed by giving the name of the array plus an index expression, called a *subscript*. A subscript value of 0 identifies the first or initial element of the array, a value of 1 identifies the next, second element, and so forth.

It may seem confusing on first sight to have arrays begin at 0 instead of 1, but this reflects C's philosophy of staying close to the computer architecture. Here, zero is a much more natural starting point for computers, even though it may be a bit more inconvenient for human beings. There are programming languages, like Fortran-77, where arrays begin with a subscript of 1. Although this method may be more intuitive, in fact it often is more costly because the compiler, which translates

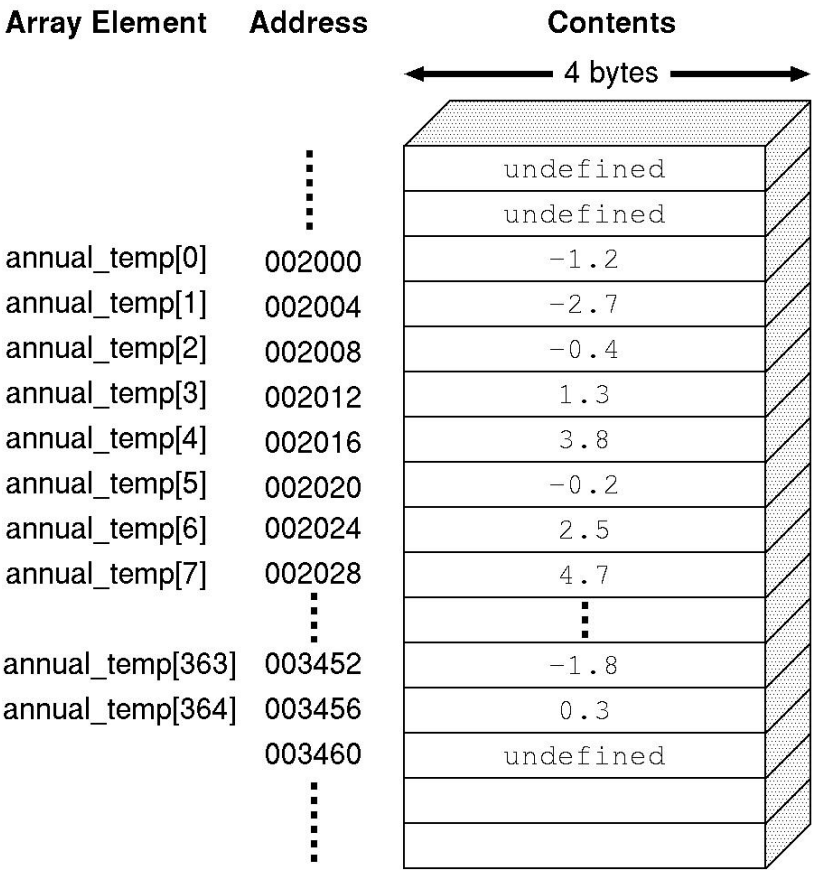


Figure 6.18: Excerpt of a computer memory in which 365 different daily temperature values are stored consecutively

the program into machine code, must now subtract 1 from each of the subscripts to get the true internal memory address of an element. The C method results in a more efficient code.

The basic purpose of arrays is to store large amounts of related data that share the same data type. Before one is able to use an array it has to be declared at the beginning of the program. Through this declaration of the array data structure an exact amount of storage space depending on the data type of the array elements is reserved automatically where the single elements are then stored later on. For the array that was used in the example shown in Fig. 6.18 the declaration will look as follows:

```
float annual_temp[365];
```

With this an array called `annual_temp` consisting of 365 floating-point elements is created. The temperatures that have been measured throughout the year can be entered by assigning the value to the corresponding day:

```
annual_temp[0] = -1.2;
annual_temp[1] = -2.7;
annual_temp[2] = -0.4;
...
```

To avoid confusion, it should be noted that the highest subscript of an array is always one less than the size that was declared for the array! This is because the subscripts start at 0 instead of 1 and therefore the last array element of `annual_temp` is accessed as `annual_temp[364]`. Nor should one ever confuse an array declaration and an array element reference that is used in an assignment as shown above. They might look almost the same but they have completely different functions.

What makes such array data structures now so efficient and so fast when they are used in a piece of code? The following example of a C program straightforwardly demonstrates the strength of arrays (for reasons of simplicity we assume that we have already assigned the daily temperature value to every array element):

```
#include <stdio.h>
#include <stdlib.h>

#define DAYS_IN_YEAR 365

int main(void) {
    int j;
    float sum = 0.0;
    float annual_temp[DAYS_OF_YEAR];

    /* We assume the single daily values are already
       read in here into the array elements and are
       ready for further processing after this point */

    for (j = 0; j < DAYS_OF_YEAR; j++)
        sum += annual_temp[j];

    printf("The average temperature for the year is %d. \n",
                sum/DAYS_OF_YEAR);

    exit(0);
}
```

Even to those readers who are not familiar with programming, the intention of this short C program should be clear: The stored daily values are added together and from this annual total, the average temperature of the year is computed and finally printed out on the computer display. Although we are processing 365 different temperatures with this program, its core part is rather short (when leaving out the procedure for reading in the array elements).

In fact, the total number of days in a year appears just once at the very beginning of the program in the form of a macro called `DAYS_OF_YEAR`, which is simply an alias for 365 days. It could also be replaced by 30 and then be called, for example, `DAYS_OF_MONTH`. Also the calculation of the average temperature consists only of two lines of code in the example above and not 365 or more C code lines.

The reason for this is the fact that by using an array structure a simple counter can be employed, and just incremented step by step; in the program example it is called `j`. `j` is initialized to a value of 0, and as long as `j` has a value that is smaller than 365, it is increased by 1 after computation of the code line consisting of `sum += annual_temp[j];`. Here, the temperature value that is stored for the array element accessed by the subscript of the value of `j` is added to the annual total of all temperatures.

At the very start, when `j` has a value of 0, the array element `annual_temp[0]` is added to the variable `sum` which has an initial value of 0. After having processed the C code line `sum += annual_temp[j];` the first time, `sum` now holds the value of `annual_temp[0]` which is -1.2. In the next "round," when `j` is increased to 1 (0 + 1 = 1), the array element `annual_temp[1]` is accessed and added to the value of `sum`, which results in -1.2 + (-2.7) = -3.9, and so forth.

What happens now when we reach the last array element, which is `annual_temp[364]` (not `annual_temp[365]`!!!)? Let us examine this case again step by step. Our counter variable `j` is set to 364, the value of `annual_temp[364]`, i.e., 0.3; this added to the total stored in the variable `sum`. With this step we have now added all 365 temperature values and have no further temperature values for this specific year. As previously, the counter variable is increased again by 1 and is now set to 365 (which is one day more than our year has days!). In the C code in the example we find an expression of `j < DAYS_OF_YEAR`. When now inserting `j` and its value of 365 into this expression, it evaluates to FALSE, as 365 is clearly not smaller than DAYS_OF_YEAR, which has a value of 365. Therefore at this stage in the program the whole process with incrementing a counter and computing a new total suddenly stops.

What would happen if we didn't have a control statement to take care of terminating such a loop in the program? The answer is: Anything could happen! It depends on the programming language and the underlying system. In Fig. 6.18 we find a value that is undefined at the memory location that would resemble the array element `annual_temp[365]`. `undefined` means that the value for `annual_temp[365]` is unpredictable. In the worst case the program would run forever, continously adding more and more non-existing temperature values to the variable `sum` until a memory overflow would be the lethal result. On some platforms a typical error is a so-called *segmentation error*, which is a classical error which occurs when something within a program tries to access parts in the memory which it is not allowed to access, for example, an operating-system memory location or a memory address behind a legally defined array, etc.

Although arrays are one of the simple and basic data structures, one should be always very careful when working with them. Especially processing undefined memory locations within, before or after an array often produce some of the most pesky bugs, as they can cause different results each time the program is executed. Such undefined values in the memory are often called *garbage* or *trash* in the programming world, and the contents of these memory locations can contain whatever is leftover from a previous program execution. They might have harmless values, such as zero, most of the time; yet in rare circumstances, they may acquire harmful values that can in the worst case cause the program to fail. Frequently, these bugs are not noticed until after a product is switched over from the test environment to the production platform or has been shipped to a customer, and the harmful values then turn up at the customer site. For this reason arrays should always be initialized. This ensures that at least the values of the array elements within the array have a declared starting value.

6.5.2 Multidimensional Arrays

The arrays introduced so far have all been one-dimensional arrays. In addition to the aforementioned ones, there are also multidimensional arrays. They are simply arrays of arrays. At least two- and three-dimensional arrays still can be imagined by a human being. A typical 2D array is a matrix that consists of more than one line or one row. Another suitable example for a 2D array is shown in Tab. 6.1.

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Table 6.1: A matrix that is called the "magic square," as its rows, columns and also diagonals all have the same sum

Zooming into Tab. 6.1 and just considering the first line of it, we are dealing with a simple one-dimensional array consisting of 5 elements. These array elements alone would be successively stored in the memory. When now working with the complete "magic square" shown in Tab. 6.1, i.e., having five rows instead of one to deal with, actually nothing would change. These 5 lines of the matrix are stored consecutively and contiguously in the memory, starting with row 1, then row 2, and so forth, until the 5th row is stored as the last. In the memory the matrix is then just visible as a 1D structure, which is 25 array elements one after another, as if we had just been working with a 1D matrix that consisted of one row of 25 single matrix elements. This might even become clearer when considering the following C statement which is both the declaration and initialization of the magic square:

```
int magic_square[5] [5] = { {17, 24,  1,  8, 15},
                            {23,  5,  7, 14, 16},
                            { 4,  6, 13, 20, 22},
                            {10, 12, 19, 21,  3},
                            {11, 18, 25,  2,  9}
                          };
```

To access an array element in a multidimensional array, one has to specify as many subscripts as there are dimensions. To read the array element with the value `20` in the `magic_array` above, and store it in another program variable called `temp`, one has to specify the two following subscripts as follows (note: the subscripts start with a value of 0 when counting and not 1!):

```
temp = magic_square[2][3];
```

The only important aspect to remember is that multidimensional arrays are stored in *row-major order* in C. (There are also languages that store such arrays in column-major order, occasionally leading to quite some confusion and misunderstandings.) In an array that is stored according to a row-major order, the last subscript is the one that varies the fastest.

Everything still holds true when now working with 3-dimensional arrays. One can easily imagine such an array as a cube or cuboid that consists of several 2D arrays or matrices that are only stacked upon each other. With the following C statement of

```
int 3d_array[3] [4] [2];
```

an array is declared (but not yet initialized!) that consists of 3 2D arrays, each of which again consists of a matrix with 4 rows and 2 columns. When this 3D array is stored in the memory, the first 2D matrix with 4 rows and 2 columns is stored consecutively and contiguously, then the second one behind and then the third one. And again, we have a simple 1D structure mapped in the memory, although on the "outside" we are dealing with a very complex 3D structure.

As these basics are very important in regard to the upcoming sections on bit arrays and dynamically allocated arrays and their handling, here is another simple example with the related C declaration and intialization:

```
int simple_array[2] [3] = { {0, 1, 2},
                            {3, 4, 5}
                          };
```

`simple_array` is a 2D array consisting of 2 rows with 3 elements each. Fig. 6.19 shows an excerpt as an example of the memory where `simple_array` will be stored.

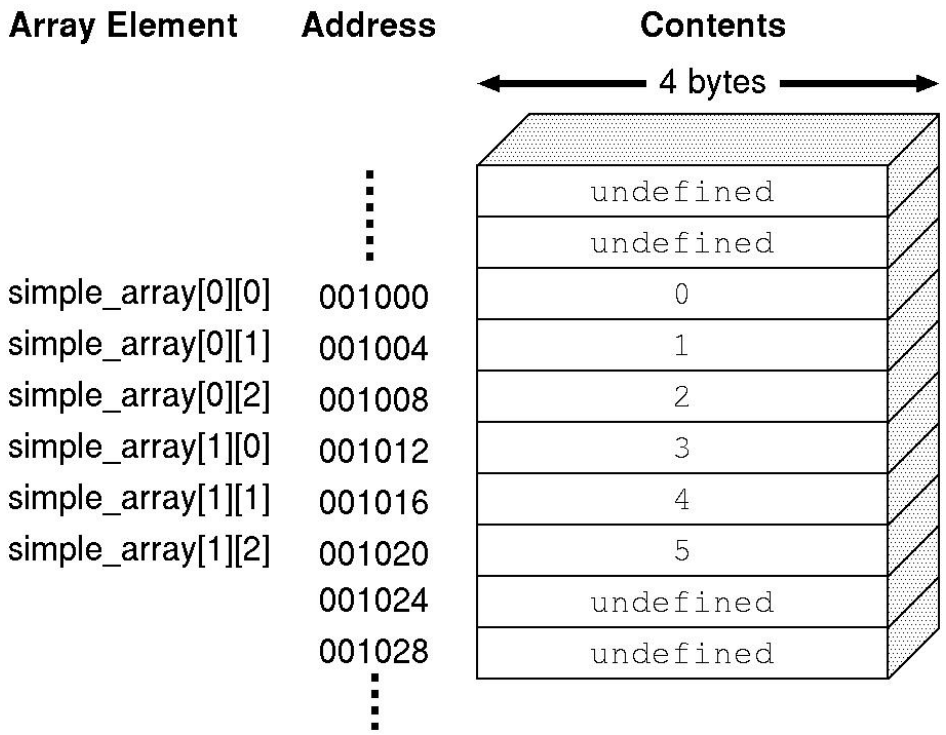


Figure 6.19: An excerpt as an example of a memory where a simple 2D array consisting of 6 elements with 2 rows of 3 elements each is stored

When examining the way `simple_array` is stored in the memory in Fig. 6.19, the following C declaration would also be correct:

```
int simple_array[6] = {0, 1, 2 ,3, 4, 5};
```

When comparing the example memory in figures 6.19 and 6.20, it is found to be completely identical. Only the "naming" or "addressing scheme" of a 1D- or 2D-array structure is different.

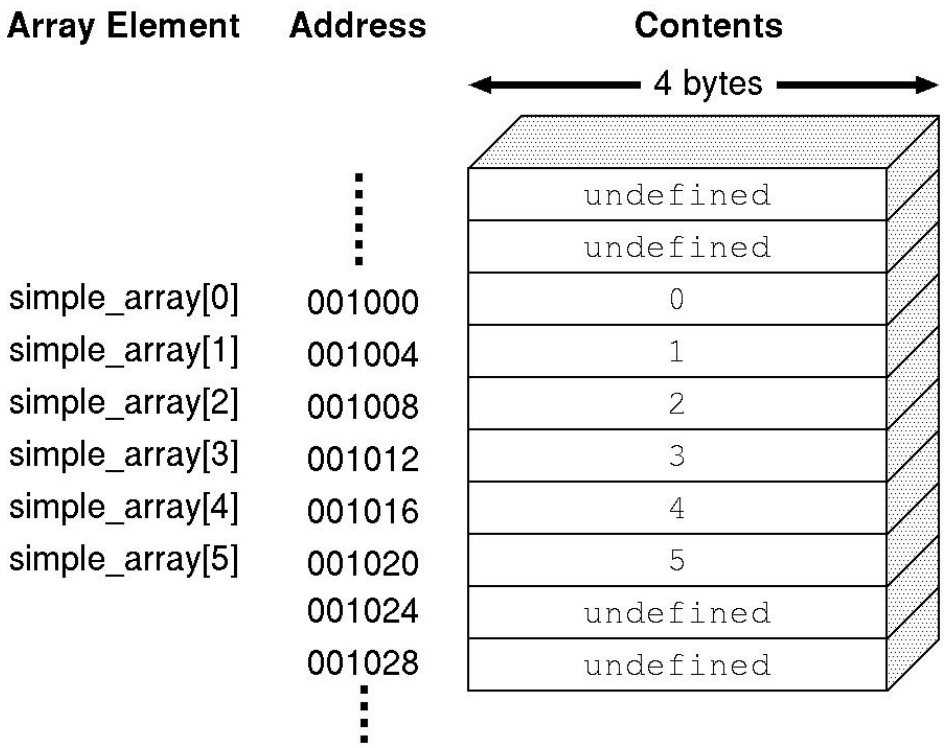


Figure 6.20: An excerpt as an example of a memory where a simple 1D array consisting of 6 elements is stored

Normally the average programmer hardly ever has to worry so much about basic circumstances like these. In a multiprocessor environment or when switching to parallel computing, all of a sudden these things become important, as programming and working in such an environment is much more hardware dependent than on an average desktop machine. Of course, the same principles that are applied in such a sophisticated and high-end environment can be used on a simple, out-of-the-box machine, but often the extra effort is not worth the improved results that can be achieved. Nevertheless, a good programmer and engineer always thinks about the most appropriate data structure for his or her problem that is valid for both worlds, on a desktop PC as well as in a HPC environment.

Up to this point we have been discussing arrays in general, in which typical data types could easily be stored, like integers or floats. But there are also some specialized types of arrays. Some of them are rarely used, like the bit arrays, which are discussed in detail in the upcoming section, because their handling is rather complicated and requires a great deal of knowledge about memory organization and hardware and computer basics. Others, like the dynamically allocated arrays, are more common but nevertheless not easy to apply.

Independent of which type of array is finally implemented, all of them build upon the same basics and rules introduced in this section. As arrays are one of the most fundamental data structures in almost every programming language, this introduction has just scratched at the surface of the topic. As this thesis is not intended as a textbook on programming, the interested user can find much more valuable and interesting information about arrays in all good programming books, not only on C but on all other programming languages which have arrays implemented. For C, which was discussed here, a good starting point would be [Darnell], [Ritchie] and [Kinzel] or, for the more advanced reader, also [Linden] and [Stroustrup].

6.6 Bit Arrays

6.6.1 Introduction – Problem of Accessing Single Bits Directly

As explained in section 3.2.1 of chapter 3, the smallest unit in computer science is one byte consisting of 8 bits. A short description of the cumbersome hardware processes that are needed to transfer data from the memory into the CPU and back again was given in section 3.2.2. To optimize these time-consuming procedures nowadays, not single bytes but so-called *words* or *memory words* are loaded and transported within a computer (CPU, cache, memory and registers). Such a word consists of a certain minimum number of bytes, the basic data unit within the hardware architecture. Usually word sizes of either 4 bytes (= 32 bits) on common 32-bit architectures or today even 8 bytes (= 64 bits) on 64-bit architectures of modern desktops and laptops are the standard.

This information reveals that it is not at all easy to operate directly on bit levels with an average application software. How would one access, for example, bit *32,435* out of megabytes or gigabytes of data when the minimum data transfer size lies at 4 bytes and 8 bytes, respectively? How does one know if this bit *32,435* is even still within the range of 1 megabyte, for instance? (To add to the confusion and for reasons of completeness here, one must not forget the various address translations which are necessary when data is "pumped" through the different memory or cache hierarchies within the computing system!)

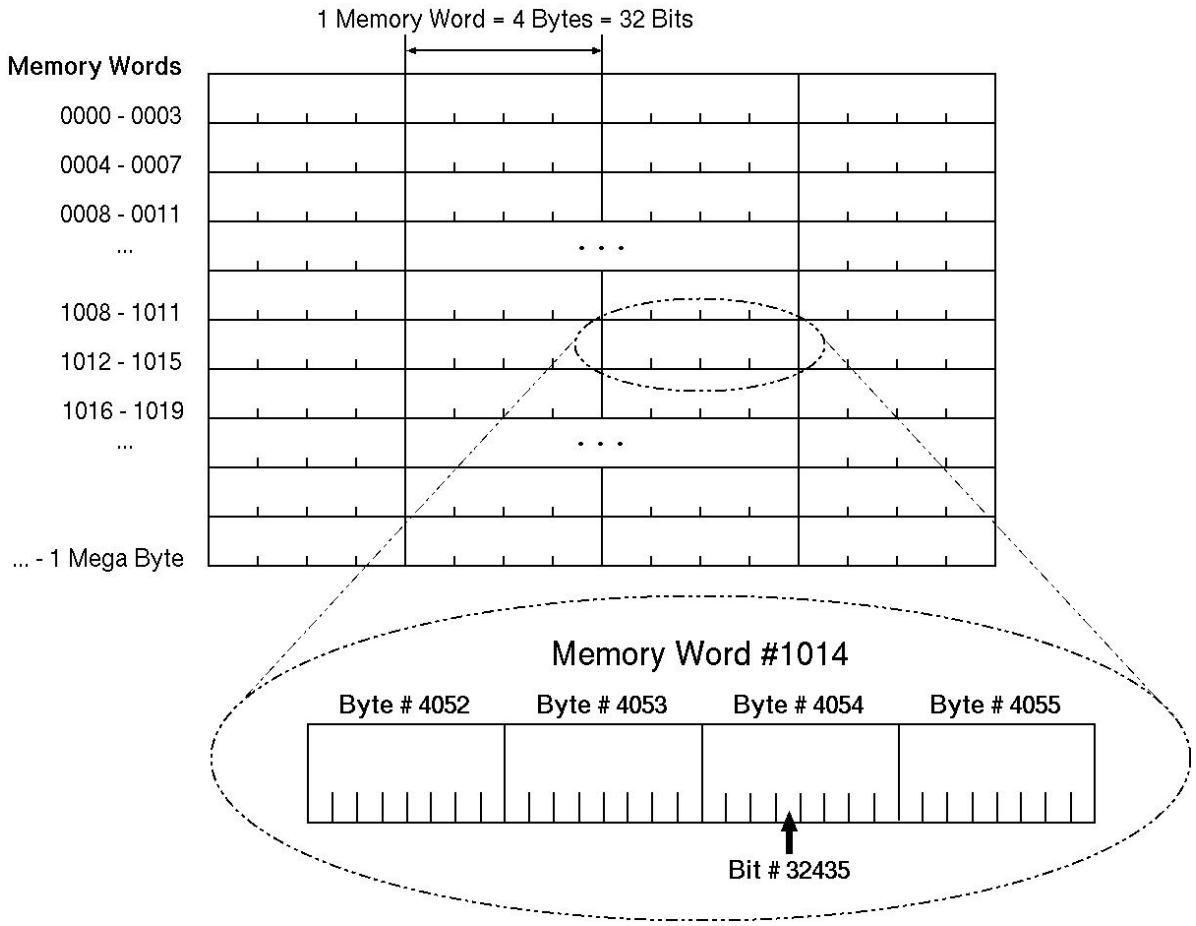


Figure 6.21: Position of bit number 32,435 within 1 megabyte of data

To answer these questions and to provide a useful foundation for the technical discussion of the

implementation of bit arrays, the following subsection offers a hopefully not too dry introduction to *congruences* and *modular arithmetic*, with an emphasis rather on practical aspects than complicated theory, which is not needed to such an extent here.

6.6.2 Congruences - Introduction into Modular Arithmetic

When opening up a math book and looking up "factoring" or the basics behind "modulo computation" (which all belongs to the field of *number theory* in mathematics), most people probably would be shocked to see the plain mathematical descriptions. An understanding of the math behind this all is definitely important but it might not be of too much help here.

By carefully observing our daily life we can easily find good examples of the usage of modulo computation (see [Neale]). Therefore, let us consider one with which everybody is certainly familiar as a convenient starting point for a short mathematical introduction to the theory of modulo computation.

Let us have a look at figure 6.22, which shows the face of a standard analog clock:

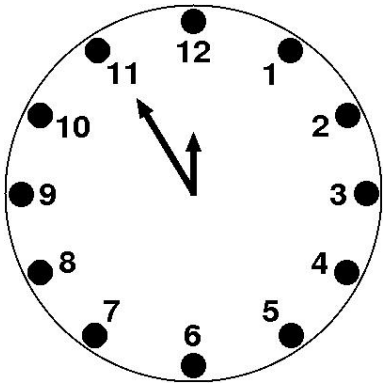


Figure 6.22: Face of a clock with 12 digits

The digits run from 1 to 12. But what happens when we have revolved 12 hours on our clock? Our standard analog clock starts counting at 1 again although our day consists of 24 hours, as shown in figure 6.23.

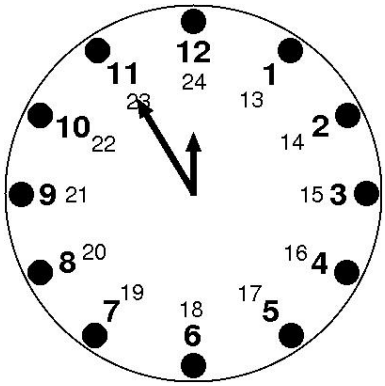


Figure 6.23: Face of a clock with 24 digits

To solve this problem we often refer to 1 o'clock in the afternoon as 1300 hours or, in some languages, as 13:00 o'clock. 2400 hours is midnight; then we start counting at time 1 of the next morning again. The next day, noontime is referred to again as 12 o'clock in the morning. Nobody would actually say it is now 3600 hours since midnight yesterday.

What we have here is nothing but two counting systems, one with exactly 12 numbers (from 1 to 12) and another with exactly 24 numbers (from 1 to 24). In our first system with 12 numbers, 1, 13, 25, 37, ... are all obviously the same. In the second system 4, 28, 52, ... are seemingly also all the same. For reasons of simplicity, let us stay with the numbering system consisting of 12 digits for the following paragraphs.

When we check our clock and see that it is now 1 in the afternoon, what we are really saying then is *13 = 1 + some multiple of 12*; or the next day, at 2 in the afternoon it is *38 = 2 + some multiple of 12* or, alternatively and in a more sophisticated way, *the remainder when you divide 13 by 12 is 1* and *the remainder when you divide 38 by 12 is 2*. The way we write this mathematically is

$$13 \equiv 1 \ mod \ 12 \qquad or \qquad 38 \equiv 2 \ mod \ 12,$$

and so on. This is read as "13 is congruent to 1 mod (or modulo) 12" and "38 is congruent to 2 mod 12."

Our counting systems do not necessarily have to work only in *mod 12* (the technical term for it). For example, we could work in *mod 7*, or *mod 46* instead if we wanted to. In such a case we need only think of clocks numbered from 1 to 7 and 1 to 46, respectively; every time we get past the biggest number, which in this system would be 7 or 46, we reset to 1 again.

Let us go back to the normal clock face with the numbers 1 to 12 on it for a moment. Mathematicians usually prefer to put a 0 where the 12 would normally be, so that we would usually write (for example) $24 \equiv 0 \ mod \ 12$ rather than $24 \equiv 12 \ mod \ 12$, although both of these are correct. That is, we think of a normal clock face as being numbered from 0 to 11, instead. This makes sense: We'd normally say that 24 leaves a remainder of 0 when we divide by 12, rather than saying it leaves a remainder of 12 when we divide by 12!

More formally, we can say that in general when working *mod n* (where n is any whole number), we write $a \equiv b \ mod \ n$ if *a* and *b* leave the same remainder when divided by n. This is the same as saying that we write $a \equiv b \ mod \ n$ if n divides $a - b$. When having a look at the earlier examples with the clock, one can see that this definition is correct. (We are still working in the numbering system with an n of 12, i.e., the numbers 1 to 12.) Let us now take 1 o'clock in the afternoon which is 13, and 1 o'clock after midnight two days later, which is the same as 49 when continuing to count the hours. This results in $49 \equiv 1 \ mod \ 12$ as well as $13 \equiv 1 \ mod \ 12$. Now let us check the difference of 49 and 13: $49 - 13 = 36$; 36 is clearly divisible by 12 without remainder.

After introducing the notation let us do some math, and see how so-called *congruences* (the term for what was described above) can make things a bit clearer.

Congruences have some useful properties: for one, we can add them. That is, if $a \equiv b \ mod \ n$ and $c \equiv d \ mod \ n$, then $a + c \equiv b + d \ mod \ n$. That seems a bit odd at first sight but $a \equiv b \ mod \ n$ is just $a = b + kn$, where k is an integer, and similarly, $c \equiv d \ mod \ n$ is just another notation for $c = d + ln$, where l is an integer. So

$$a + c = (b + kn) + (d + ln) = (b + d) + (k + l)n$$

and thus, $a + c \equiv b + d \ mod \ n$. For an example with real numbers, we consider $17 \equiv 4 \ mod \ 13$, and $42 \equiv 3 \ mod \ 13$, which results in $17 + 42 \equiv 4 + 3 \equiv 7 \ mod \ 13$. We should keep in mind that both of the congruences being added are $mod \ n$, and so is the answer. A caveat: We do not add the moduli!

Congruences cannot only be added, as shown above, but also subtracted. This means if $a \equiv b \ mod \ n$ and $c \equiv d \ mod \ n$, then $a - c \equiv b - d \ mod \ n$. We can also prove that something similar is valid for multiplication: If $a \equiv b \ mod \ n$ and $c \equiv d \ mod \ n$ then $ac \equiv bd \ mod \ n$. The same rules used to prove addition apply. Again, a caveat: Both of the congruences we are multiplying are $mod \ n$, which means that we do not multiply the moduli!

For proving the correct rules for division, we must be very careful. An example shows why: Apparently it is true that $10 \equiv 2 \ mod \ 8$. But if we "divide both sides by 2," then $5 \equiv 1 \ mod \ 8$, and this is clearly nonsense. To get a true congruence, we would have to divide the 8 by 2 as well: $5 \equiv 1 \ mod \ 4$ is acceptable. If is not clear why this is so, we can consider the fact that $a \equiv b \ mod \ n$ means nothing but $a = b + kn$ for some integer n, which is simply a normal equation. If we wish to divide a by some factor, then we have to divide *all* of the right-hand side by this factor as well, including kn. In our example this means we have to also divide 8 by 2, which results in the correct congruence. As a general rule, it is best not to divide congruences. Instead, one should think what they really mean and start working from there instead of using the shorthand. In [Schmeh] more theory behind modulo division and other operations like root or potentiation is given and illustrated by simple examples with real numbers.

Readers interested in learning more about number theory in general, congruences and the fascinating math with factor rings and fields behind them should have a look into [Quattrocchi], [Beth], [Markowski] and [Stallings]. Many of these books are from the field of computer security and cryptography, as some of the most popular encryption and decryption algorithms are based on such simple congruences.

6.6.3 Bit Arrays under Implementational Aspects

The term *bit array* by itself already suggests what is hidden behind these not quite common data types: it is simply an array of bits - nothing more. A rather nasty and nitpicking computer scientist could add now that every array, no matter what type, always consists of bits, as an array is built of bytes, and 8 bits form one byte, and that one should be more precise here. But a second computer geek, an even bigger nitpicker, might then indicate that in our case, this would be perfectly fine, as the bit arrays we are examining and working with here are simply a long row comprised of one bit after another. The data type itself is redundant.

Once this difference of opinion has been “overcome,” the question could arise as to what this is all good for. Didn’t we assert at the beginning of section 6.6 that the minimum word size of a contemporary computer should be at least 4 or 8 bytes? Whatever is moved back and forth somewhere within the computer hardware would not fall below this minimum size! So, how does this all make sense then? It seems that we do not have direct access to the single bits within such a memory word at all, yet we are claiming that a long listing of single bits is a good solution to work with.

The truth is: All of the above are correct! By the usage of bit arrays we are still retaining the restriction with the minimum size of information of 4 or 8 bytes in memory word length. At the same time though, we indeed are operating on bit level in the memory of the computer. How this

all is achieved will be explained in a way that even a non-computer scientist or a person who is not a big "hacker" can still understand what is being done when implementing such bit arrays. (It should be noted here, though, that it is neither reasonable nor the goal of this text to end up with the complete code listing with thousands of lines of C code.)

Motivation – A Simple Example

To bring some light into this darkness let us have a look at the following example, which starts out with a snippet of pseudo code:

```
for i = 1 to amount of input elements do
    if input element[i] exists and is loaded then
       set switch[i] to 1
    else
       set switch[i] to 0
    endif
endfor
```

A further assumption in regard to the above code fragment is an amount of input data of 20,000,000 elements of a model that is to be computed. As is immediately evident from the given instructions, the result of the verification of whether an element exists and is loaded will be stored in an array or array-similar data structure, indicated by the "expression" `switch[i]` in the code fragment above. (We do not make a statement as to how the input data is stored and administered here, nor is it of interest at this moment.)

Assuming a memory word size of only 4 bytes, this array for the switches per element will occupy $4 * 20,000,000 = 80,000,000$ bytes in the memory. According to table 6.2 and how one "defines" the size of 1 MB, this is approximately 80 MB.

When processing real-world models in simulations, a great deal of different information has to be stored in addition to the elements themselves. The code fragment above was just a simple example. Generally, not only 4 bytes are needed per element but many more, depending on the simulation and computations. During a normal code execution and program run, many auxiliary data structures, like the intermediate results or other temporary variables, have to be stored. In this way, quickly a few hundred megabytes or even gigabytes of data have to be held in memory, in addition to the elements.

One should not forget that the available space for storing such information is not exclusively reserved for such a program run. Background processes of the operating system, administrative tasks and maybe even other user processes are running in parallel, depending on the platform where the code is executed. Let us take, for instance, an average desktop or laptop computer with 512 MBytes of memory as the machine on which our example code above would be executed. Not only holding the 20,000,000 elements in an array similar to the *switches array* from the code snippet and just one single, additional auxiliary data structure in the same size to process these elements would occupy roughly 160 MBytes or a little less than 1/3 of the whole memory.

On HPC platforms generally more memory is available but in most cases the operating nodes are shared among several different users and their programs. Given the fact that simulations on such multiprocessing systems are very demanding in regard to processing time and memory, it is directly clear that only fast and effective data structures should be used.

Size/Unit	Common Shortcut	Description
Byte	–	2^3 Bits = 8 Bits
KiloByte	KByte	2^{10} Bytes = 1,024 Bytes
		= 2^{13} Bits = 8,192 Bits
MegaByte	MByte	2^{10} KBytes = 1,024 KBytes
		= 2^{20} Bytes = 1,048,576 Bytes
		= 2^{23} Bits = 8,388,608 Bits
GigaByte	GByte	2^{10} MBytes = 1,024 MBytes
		= 2^{20} KBytes = 1,048,576 KBytes
		= 2^{30} Bytes = 1,073,741,824 Bytes
		= 2^{33} Bits = 8,589,934,592 Bits
TeraByte	TByte	2^{10} GBytes = 1,024 GBytes
		= 2^{20} MBytes = 1,048,576 MBytes
		= 2^{30} KBytes = 1,073,741,824 KBytes
		= 2^{40} Bytes = 1,099,511,627,776 Bytes
		= 2^{43} Bits = 8,796,093,022,208 Bits
PetaByte	PByte	2^{10} TBytes = 1,024 TBytes
		= 2^{20} GBytes = 1,048,576 GBytes
		= 2^{30} MBytes = 1,073,741,824 MBytes
		= 2^{40} KBytes = 1,099,511,627,776 KBytes
		= 2^{50} Bytes = 1,125,899,906,842,624 Bytes
		= 2^{53} Bits = 9,007,199,254,740,992 Bits
ExaByte	EByte	2^{10} PBytes = 1,024 PBytes
		= 2^{20} TBytes = 1,048,576 TBytes
		= 2^{30} GBytes = 1,073,741,824 GBytes
		= 2^{40} MBytes = 1,099,511,627,776 MBytes
		= 2^{50} KBytes = 1,125,899,906,842,624 KBytes
		= 2^{60} Bytes = 1,152,921,504,606,846,976 Bytes
		= 2^{63} Bits = 9,223,372,036,854,775,808 Bits

Table 6.2: Standard units of data sizes after *IEC*

Saving Memory Through the Usage of Bit Arrays

When again using the example above of our 20,000,000 elements and their processing, it is clear that there is no way around reducing the amount of memory for just storing all the elements. They most likely will be integers or floating-point numbers, which means depending on the hardware environment, compiler or programming language, a certain minimum amount of memory space is consumed when storing them as input data for our simulation. Typical values for certain data structures are on 32-bit architectures or in most programming languages and compilers like the following (see for example [ISO 9899]):

- **Integer values:** 32 bits
- **Long integer values:** 64 bits
- **Floating-point values:** 32 bits
- **Floating-point values with double precision:** 64 bits
- **Floating-point values with quadruple precision:** 128 bits

In case the 20,000,000 elements are integers or floating-point values, they occupy more than 76 MBytes in memory. In case of 64-bit values this is even more than 150 MBytes of space. If now additional auxiliary variables have to be stored during processing as described in the subsection above, a great deal of memory is needed during program execution.

Quite often during processing simple *yes*- and *no*-values have to be kept for the elements. If such information is stored in a simple integer variable, it still occupies 4 bytes of memory space. When using only a data structure like a *char type* in C programming, still 1 byte is reserved. But a *yes/no* value could be stored in just one single bit! Comparing these results, either 76 MBytes (4 bytes data structure), 19 MBytes (1 byte data structure) or a little bit more than 2 MBytes (1 bit data structure) of memory are used.

Another improvement can be achieved when further examining the input data. In FE models the elements are numbered consecutively. Gaps in between the model data caused by missing elements due to incorrect discretization, for example, are not allowed. Information and data in connection to an element are often just bound in regard to the element's number in its model. But if the element itself has no meaning, and just its number is of importance, then only the index is of interest. This means there is no need to store 20,000,000 elements, for instance, as they are not needed. Instead many auxiliary variables can be accessed using the element number as an index. Combining this "trick" with the idea of storing *yes/no* values in just one bit of data, memory consumption can be reduced to the maximum extent possible. Fig. 6.24 shows the difference in memory space occupied when storing values in 4-byte, 1-byte and 1-bit data structures.

Short Introduction to Boolean Algebra

Before finally "diving" into the implementational secrets of bit arrays, a brief introduction to *Boolean algebra* is needed, as the operations that are performed on the bit arrays are based on these laws. The standard, numeric mathematic operations of addition, multiplication and negation can be also expressed by logical operations. This was shown in the late 1830s by an Englisch

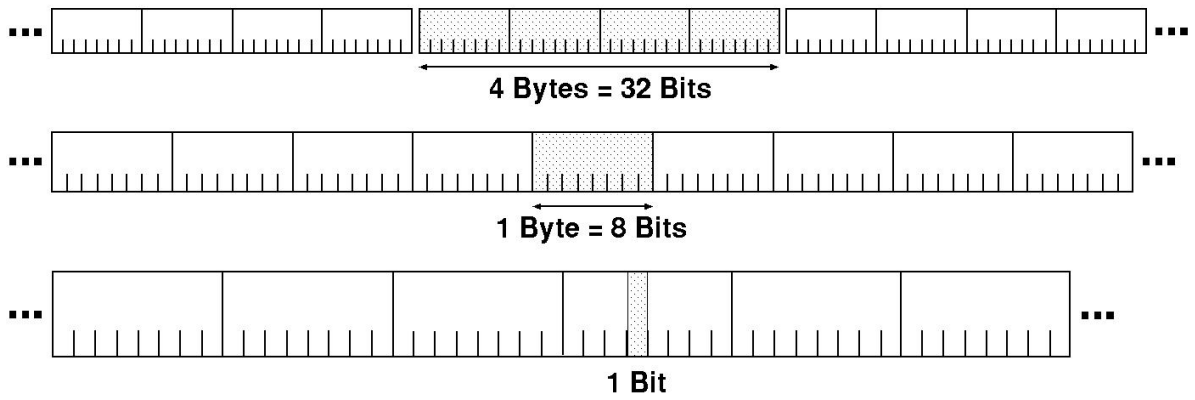


Figure 6.24: Difference in memory consumption using 4-byte, 1-byte and 1-bit data structures, respectively

mathematician called George Boole. The Boolean operations operate on the set of {0,1}. Often also the values TRUE and FALSE are used interchangeably, instead of 0 and 1.

The Boolean Algebra is so important because the processing in a computer is based on it. In former times when they still had big electronic components like gates or tubes, computers were constructed in such a way that by coupling these components internally, information could be processed all at once: Depending on the switching, for example, either current was flowing or not; and, later on, with magnetic components, magnetization was switched on and off. Thus, information like yes/no, TRUE/FALSE, no current flows/current flows, etc. could be used and be processed all at once, according to the laws of Boolean algebra. By coupling certain gates in a typical way, for example, the Boolean operations could be built and, by arranging certain sets of gates, even complicated Boolean equations could be assembled.

The three Boolean operations are a logical AND (also often written as $\vee$ or in C as &), logical OR (also $\wedge$ or in C as |) and NOT as the negation of a value (also written as $\neg$, $\sim$ or in C as !). In Fig. 6.25 the truth tables for these 3 operations are shown.

$\wedge$	**0**	**1**
0	0	0
1	0	1

$\vee$	**0**	**1**
0	0	1
1	1	1

	$\neg$
0	1
1	0

Figure 6.25: Truth tables for the 3 Boolean operations AND, OR and NOT

As the truth tables might be somewhat difficult for the unexperienced reader to understand, here three different descriptions of how the AND, OR and NOT work are given when assuming two variables p and q, which could either hold a value of 0 or 1:

- **AND:**
  Only when both variables, p and q, are set to 1, the result of the AND operation is also 1.
- **OR:**
  In case at least one of the two variables p and q is set to 1, the result of the OR operation is also 1.

- **NOT:**
  In case the variable is set to 0, applying NOT to it will switch the value to 1 and vice versa.

In Fig. 6.26 a different, more commonly used way of writing the truth tables is shown. Below them a drawing is provided to demonstrate this relation when using two sets.

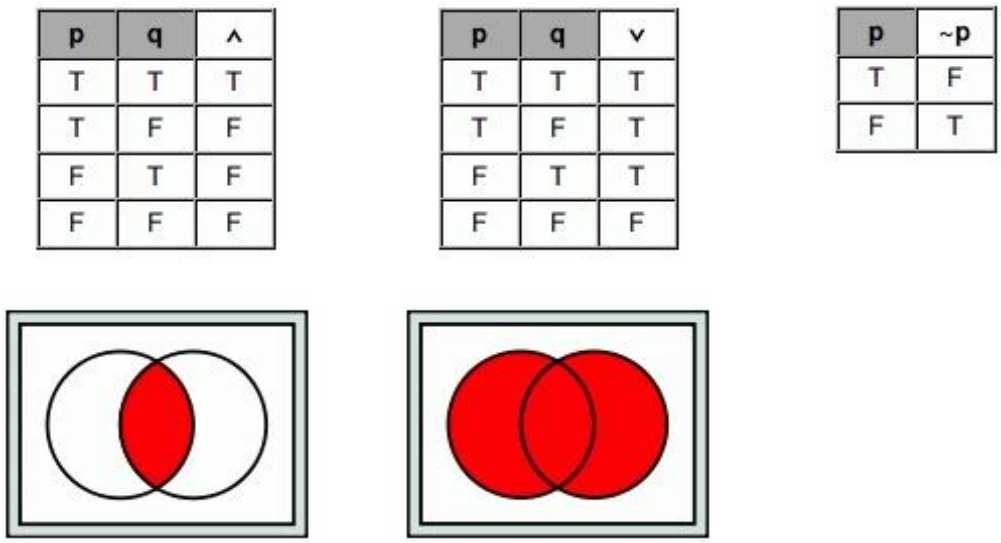


Figure 6.26: Truth tables for the 3 Boolean operations AND, OR and NOT

Provided with this little "Boolean toolbox," we are actually now able to do math! In Fig. 6.27 this is shown as an example using the AND operation upon two numbers, X and Y.

```
      x:  10001101
      y:  01010111
x AND y:  00000101
```

Figure 6.27: Adding two binary numbers, X and Y

At this point we are now readily stocked with the necessary basics to finally focus on how the bit arrays are actually used.

Usaging the Bit Arrays

No matter what is to be done with bit arrays, there is always the problem of how to determine where the bit of interest is located in the corresponding data structure. In subsection 6.6.2 the basics of congruences and how to use them were briefly discussed. Here, in this subsection, we now need precisely such information.

Previously we stated that a bit array is nothing more than an endless row of bits, in which one can store either the value 0 or 1. Such a bit array, starting with index 0, is depicted in Fig. 6.28.

As can be seen from Fig. 6.21, each of the bits in the array has its very own subscript or position, similar to the subscript in an array with which an array element is determined.

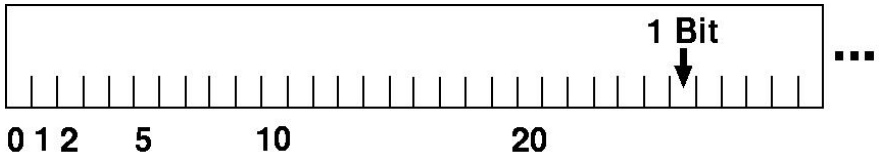


Figure 6.28: A simple bit array starting at bit 0

The only problem we have with this bit array now is that computers do not work with bits but with their word size, depending on the underlying architecture. There is no way to directly implement such a condensed data structure as a bit array in C. Simply no function or command exists which directly accesses bit xy from such an specialized array. Somehow we have to "map" the bit array onto the smallest possible structure of the computer where it can be implemented which is either in blocks of 32 bits when working with a word size of 4 bytes, or in blocks of 64 bits when working with a word size of 8 bytes, respectively. For the rest of the section we are assuming a 32-bit architecture but everything can be easily transferred to a 64-bit architecture by basic mathematic operations.

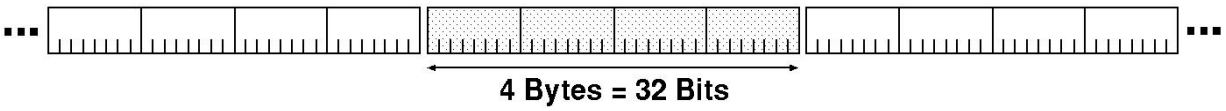


Figure 6.29: A bit array organized in bytes and blocks of word size on a 32-bit architecture

As can be seen from Fig. 6.29, a bit array is organized in blocks of word size which consist of 32 bits each, or 4 bytes of 8 bits. To access one bit within the bit array we have to give two indices: One specifies the block of word size the bit is located in and the second one gives the precise position within this 32-bit block of word size.

Just as n input values stored in a normal, static array from 0 to n-1 are accessed by their subscript (for example, input element 5 has subscript 4, counting from 0) and stored in the 5th position in the array, so now a bit within a bit array can be accessed by determining the number of the block and the bit position within this block from its true value.

Let us assume we have n input elements and would like to store input datum k with $0 \leq k \leq n-1$ (only integer values allowed). To find the corresponding block b_k of word size in which the bit is set, the original number of the datum k that is to be stored has to be divided by 32.

Now we have to recall the congruences already introduced in subsection 6.6.2 of this chapter. When dividing k by 32 we need an integer number as a result of this operation, as the blocks only have integer numbers and no floating point numbers. Therefore we have to do a division without remainder to determine the correct block number b_k.

Only in case our value k was a multiple of 32 does a "pure" integer number result; in all other cases we are "cutting off" a certain remainder when performing such an integer division. But what happens, in case we do *not* have a multiple of 32? Let's say we get a result of 4.15 from the division? Here we encounter the congruences again! The remainder gives us the exact position within the 32-bit block b_k. As any of the 32-bit blocks has a bit position 0, a bit position 1, a bit position 2, and so forth, we can simply start counting again at zero when having reached bit 31 (we start counting at 0!) in the block and shift to the following 32-bit block where we start counting

again from 0. Thus, we are operating not with mod 12, as we did in the example with the clock previously, but now with mod 32. Any number mod 32 results in a value between 0 and 31. Et voilà! We have our bit position.

Unfortunately, matters are not this easy on computers. They still require their basic operational width, which is 32 bits! (It is possible, though, to write single bytes, but this does not help us here and going into more detail would only foster confusion and not further understanding!)

As we are not able to write or read precisely one single bit but always bits in the number of word size, we have to find a way to still select our appropriate bit. Now here, the Boolean operations, introduced earlier, come in handy. By using some kind of "masks" we can manipulate the single bits precisely at these positions we are interested in.

Assuming we have 8 bits with an unkown value of XXXXXXXX and we want to set the 3rd bit from the right to a value of TRUE. The operation XXXXXXXX | 00000100 results in a binary value of XXXXX1XX. Using the OR relation, we have manipulated precisely bit no. 3 from the right and this is set to TRUE, as intended. The remaining 7 bits stay completely untouched.

The same operation is applied when it is necessary to delete a bit, which means setting its value to FALSE. We do the same thing, use a mask as before, but now we apply the AND operation and not the OR. By doing so we result in XXXXXXXX & 00000100, i.e., XXXXX0XX.

The harder life is made by the C language when working with bit arrays, the more comfortable it is to set up the mask and to position the value of TRUE in it, as was done with 00000100 in the example above. Here the *shift operator* helps, which is "«" for shifting a bit one position to the left and accordingly "»" for shifting a bit one position to the right.

Unfortunately, we are still missing one little detail in order to be able to conclude this subsection with a helpful, practical example. We have found a way to determine the exact position of the bit within the bit array, i.e., in the appropriate block b_k. We have seen that we can use a simple mask to manipulate the bit within the block, but we do not know how to deal with the division, the remainder problem and the fact that we are working mod 32. But fortunately, things aren't too complicated here, either, as we can see by having a look at Fig. 6.30.

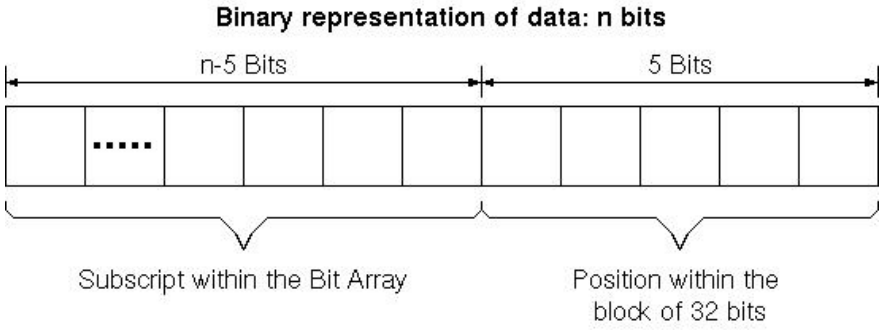


Figure 6.30: A binary representation of the value n showing the two necessary indices for accessing its bit in a bit array

When we perform a division without remainder on a binary representation of a number, we actually shift the whole number of binary digits to the right. In our case, when we perform a division of 32, which is 2^5 and thus 5 bits, we cut off the 5 rightmost bits and no longer regard them. The block b_k simply results from the remaining bits after the division. (Now it also is clear why we cannot

work with a floating-point number in this case but only a true integer number.) This is depicted in Fig. 6.31.

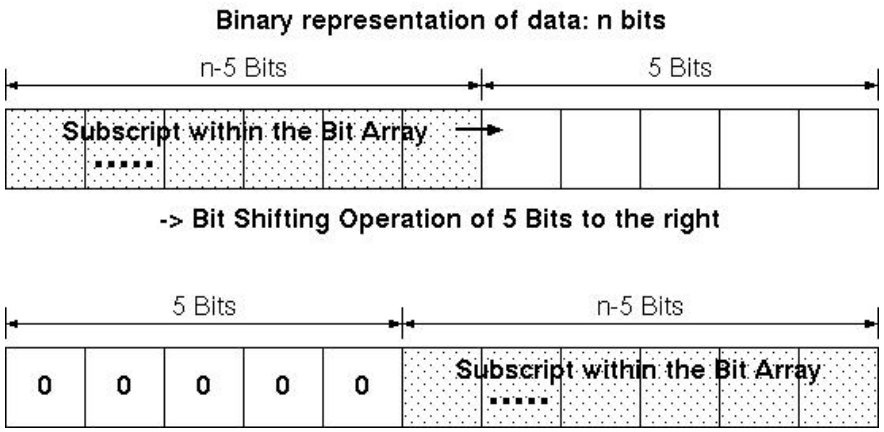


Figure 6.31: Performing a division of 32 without a remainder on a binary representation of a number n

As we saw also in Fig. 6.31, our remainder, which is "cut off" when doing the division by 32, is the 5 bits on the very right. When just determining their simple value, a binary 5-digit number tells us exactly which bit position within a 32-bit block is to be flipped or regarded. The block b_k in which this bit is located is given by the other n-5 bits of the binary representation of this number.

For the reader interested in how this is accomplished using C, the necessary short C statements follow. Here we assume that we have a bit array called `bits[]`, n being our datum in which we are currently working:

```
bits[n/32] |= ( 1 << ( n & 31 )); /* for setting bit in array */
bits[n/32] &= ~( 1 << ( n & 31 )); /* for deleting bit in array */
```

In case we would like to check if a bit is set in the bit array or not, the following C conditional statement is needed:

```
if ((bits[n/32] & ( 1 << ( n & 31 ))) != 0 ) {
     ....
} /* end of if-statement */
```

It should be noted here for the sake of completeness that a normal integer in C is always a signed integer value, which means it is either clearly set to a positive value or a negative value. This way the integer does not have 32 bits with which one can work but only 31, as one bit is saved to store the prefix. As we require 32 bits to work with the bit arrays, in order not to add to the already enormous amount of complexity, we "skip" the prefix by not working with signed integers but with unsigned integers. Thus, the prefix is discarded and its position is now also available to store bit values.

As we finally have all our "ingredients" together for working with a bit array, the last subsection of this section will present a brief but significant example.

Example of Working with a Bit Arrays

In contrast to the previous subsections, which were quite theoretical, this concluding subsection now presents a short example in which we assume that the number k is set to the example value of 133, which is 00010000101_2 in binary notation. The bit array we are working with is called `bits[]` and is declared in C as `unsigned int bits[64]`.

First we have to find out which of the 64 different `unsigned int` blocks is the correct one to operate on for our k of 133. Therefore, we have to divide 133 by 32, which equals the previously described bit-shifting operation of 5 bits to the right. In C notation this is `133 / 32` as we are using integer division, or, in binary notation, $00010000101_2 \gg 5$. This results in 4.15 but as we are using integer arithmetic, this is rounded to a value of 4. This holds true when considering the value of the shifting operation that turns out to be 000100_2. As the bit array `bits[]` starts at a subscript of 0, we are actually working within the 5th `unsigned int` block, i.e., `bits[4]`.

The second and last step is now to find out which of the 32 bits within the block of `bits[4]` is the correct one and is to be set to one. For this we finally use our $mod\ 32$ operation. In C notation this is written as `133 % 32`. Working in binary mode, we will achieve the same by a bitwise AND operation: 00010000101_2 & 00000011111_2. The result of this equals 5 or 00000000101_2, which indicates that a value of 1 has to be placed at the 5th bit within `bits[4]`. In Fig. 6.32.

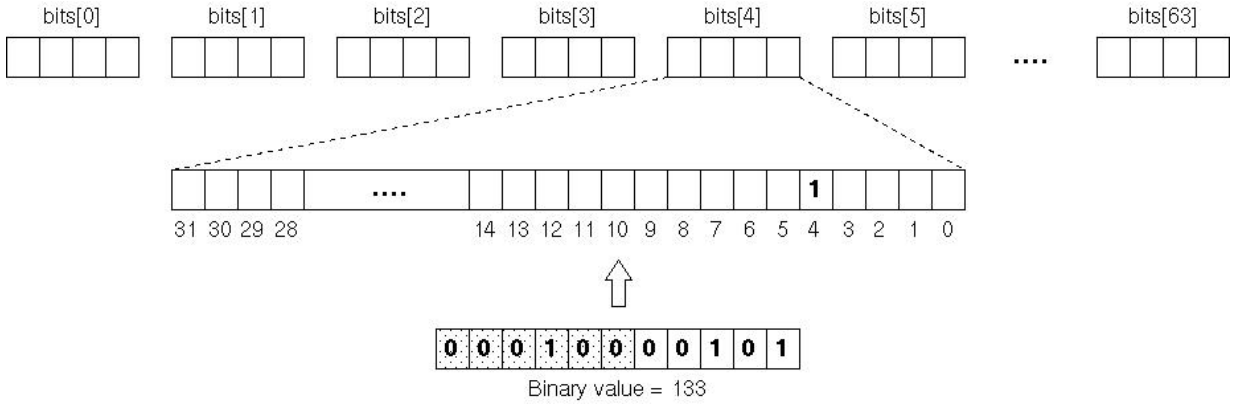


Figure 6.32: Determining the correct bit position in a bit array for a binary value of 133

It is obvious that the general basic operations when working with bit arrays are not especiallay complicated. Unfortunately things start to become extremely confusing and unclear the very second more than one dimension is involved and the necessity of numerous additional auxiliary and temporary variables arise.

In the first version of the wrapper a 2-dimensional bit array was implemented. The first dimension had to be set up as a normal, static integer array over all elements or nodes, for example. The second dimension then, could be realized as a bit array. Things became very complex by keeping the wrapper independent of the underlying architecture. Thus, it didn't matter if blocks of 32 bits or 64 bits in size were used as the word size. When determining the bit position now, not only the word size had to be determined but also the element or node for which the bit had to be examined or set and, finally, the bit array in which the single bit, stored to this variable, had to be located.

Although the bit arrays could be implemented and tested thoroughly and finally ran without any

problems, they had to be replaced by a better suited data structure, as they had one serious drawback. A full discussion of this is left for section 7.2 in chapter 7, starting on page 209.

6.7 Dynamically Allocated Arrays

The reader who has made it up to here, through all the implementational facts and details, will already have learnt a good deal about arrays in general (see subsection 6.5.1) and more specifically, about *bit arrays* (see previous section 6.6).

The upcoming section deals with another type of array, the so-called *dynamically allocated array* (DAA). To avoid leaving the interested but non-experienced reader here alone in the dark, the section begins, as usual, with an introduction to what is hidden behind these arrays in general and gives an overview of the very basics of these arrays. The following subsections will then dive more deeply into the details of the problems that arose and what finally made the dynamically allocated arrays the almost "perfect" data structure now implemented in the wrapper program. (One should note that "the" perfect data structure does not exist; there are only well-suited ones, as is the case here with the DAA).

6.7.1 Introduction to Dynamically Allocated Arrays

At first look, a DAA is not any different from the "normal" array structure introduced and described in subsection 6.5.1, starting on page 137. Even most of the handling of DAAs during their usage for storing elements is identical. They are accessed in the same way and when using them, one does not notice that he or she is dealing with a special kind of array. So wherein lies the difference and what makes them so special in comparison to standard arrays?

When working with a standard array data structure the programmer has to declare it at the beginning of the C program, exactly as was necessary for the 1D arrays in the example of `annual_temp` on page 139, in the example of a C program on page 140, for the multidimensional arrays with the `magic_square` on page 142 and the example of the `simple_array` on page 143. In any of these cases, at some point the programmer or engineer had to specify how many elements were to be stored in the array or in each of the dimensions of the array.

It is obligatory that the underlying routines or devices, like e.g., the MMU (see also subsection 6.4.4, starting on page 132), be able to reserve a certain amount of space in the memory when the program is started and run. When declaring an array, everything is "prepared" in the memory so that at the very second the data is going to be stored in the array during the program execution, sufficient space is available. Arrays are *directly* accessed, unlike a list structure that uses pointers, as explained in section 6.4, starting on page 128, where the pointer structure first had to be dereferenced in order to find the true value that is stored somewhere in the memory. This means when there is no space immediately available in the array to place a datum and to keep it there, the program is going to crash (unless the programmer is well experienced and has a good exception handling installed).

Arranging for such space in the memory and reserving it is called *the allocation of memory*. It is something that is done once at the beginning of the program before processing; therefore it is called *static*. There are no more changes during the program execution once the memory has been allocated.

Let us return to the DAAs with this little explanation in the back of our minds. Just the name, *dynamically allocated arrays*, already indicates how such DAAs differ from "normal," static arrays. Instead of being declared once and "forever" before the main routines of the code, such DAAs are able to change their size. This might read pretty "neatly," but in reality involves many actions that carry out this change in memory space in the background, invisible to the user, during execution of the program. This can prove dangerous if the user does not know exactly what needs to be done or what to take care of.

In most cases, the DAAs start off as a normal static array when the program is being run. At some point during the execution the memory size for the array is *reallocated*, which means a change in size compared to the original size with which it started out. It could either shrink (which is rather unlikely to happen) or the array will grow in size; how much or in what way always depends on the case or the situation this technique is used in. There is no general recipe as to how much it should grow or shrink nor a perfect time when the reallocation has to happen during the execution. It always depends on the specific situation or programming project.

When such a reallocation is necessary, the underlying routines of the operating system and sometimes the hardware components involved (depending on the realization of such routines on the given architecture) have to ensure that enough "new memory space" is available. If so, the entire "old array" already existing which needs to change its size is taken and copied completely into a new memory location that was reserved through the reallocation process. The old memory location, where the array was stored before it was copied, is then freed and can be reused.

Generally, if possible, the underlying system tries to move and copy as little data as possible. So, given the case that there is still enough free space behind the already existing array, more free memory space is "appended" to the already existing array. But one cannot rely on this. It may be that when a machine with little memory is heavily used by various different users at the same time, the array is really copied back and forth to a brand-new location.

No matter what now really happens – invisible to the user of the program being executed – this action requires extra time and additional administrative work in the background. The more often such reallocation is necessary, the more of this extra work is performed and as a logical consequence, the more the execution of the program is slowed down. There are cases when such a reallocation is the only reasonable solution to a given problem; the responsible programmer or engineer has to sit down and thoroughly examine the situation in order to determine how often and to what extent such a reallocation has to be performed. If possible, it should be avoided.

6.7.2 The Necessity of Dynamically Allocated Arrays

Although DAAs involve much hidden extra work and thus a probable increase in runtime and harbor a certain risk that something might happen during their reallocation, they are nevertheless used. But when does their implementation and usage make sense? Shouldn't the programmer or engineer thoroughly analyze the data *before* the coding and implementation process and so determine the exact amount of information which has to be "shoveled" back and forth and ultimately stored? The answer here is clearly *yes* but there are still cases in which the true amount cannot be foreseen despite the utmost care and analyzation.

On page 123 two different model representations were depicted in Fig. 6.6. As pointed out in section 6.2, starting on page 120, the programmer can provide for the best possible algorithm in setting up the adjacency graph as well as a very good domain decomposition but he or she never

knows in advance what type of model is going to be processed. Is it a model that consists of various differently shaped geometrical elements or is it a model that is built solely from cubes? Maybe the elements are triangles instead?

Even with the best analysis the programmer cannot predict what model has to be processed. The model with purely triangular-shaped elements most likely will still be more complex – even with a smaller number of elements – than a model that consists only of cuboids. The engineer can arrange for the size of input data at the very beginning of the program execution and can make sure that the initial data structures that will be used for storing the input data sets are well suited, large and fast enough. But in the further processing of the models the data will be considerably transformed. New information, either temporary or critical for the final result upon program termination, will be generated and has to be stored; furthermore, the data that has already been read in is not required to stay "static" without alteration.

Keeping to the example of the model with solely triangular-shaped elements, problems already start to arise when establishing how many neighbors an element has. In section 6.2 the painstaking method for determining the vicinity of an element was described, in the example given there, element no. 10 ended up with 14 direct neighbors. Taking element no. 23 instead would suddenly lead to only 5 directly neighboring elements, and element no. 1 would have a total of 7 elements surrounding it. So, what should the programmer now do? This was just an example in 2D, what numbers would be likely for the vicinity of an element in case we switched to a 3D model?

It is clear that there is a certain minimum of direct neighbors; the good programmer has to make sure that the data structure that is going to hold the vicinity of an element is at least this size. But when setting up the model, determining the true maximum size for such a data structure is almost impossible, as there are countless combinations of all kinds of mixtures of geometrical shapes for the elements. Of course, one could establish the absolute maximum of direct neighbors that must not be exceeded, but what about the cases in between? And again, there is not just one element for which the vicinity has to be stored, but maybe millions.

A simple calculation demonstrates the difficulties just described: Let us assume that 5 is the maximum number of direct neighbors an element can have and that the element numbers are stored as simple integer values. When now setting up the adjacency graph for 1 mio. elements and assuming this maximum of 5 for all of them, even if they had fewer direct neighbors, we would have already had to reserve 1 mio x 5 x 4 byte (on a 32-bit architecture), or 20,000,000 bytes, in our memory space. That is roughly 20 MB. Now changing the possible maximum to 15 instead of 5, and repeating the calculation, we end up with 60 MB, which is three times the storage space. And we still haven't processed anything else; neither an auxiliary variable nor anything else has been stored up this point. Just a few years ago, a normally equipped desktop computer had 64 to 128 MB of memory; then the difference between 20 MB and 60 MB was extremely important! And what if only 1 % of the elements really needed the memory space for 15 elements and the rest had a number of direct neighbors which was much less than 15? We would have reserved the whole memory completely in vain!

This simple example clearly depicts the dilema each programmer has to face: Is it always wise to directly allocate the maximum amount of space in the memory or rather to adapt the storage space dynamically as the need arises during program execution? The answer to the question strongly depends on the given situation, the underlying hardware, the computing environment as well as the goal of the computation. Should an enormous amount of memory be available to the only user on a big number cruncher, the usage of the memory would be of secondary concern. Here one probably would concentrate more on the optimization of the underlying algorithms to efficiently

use the given hardware.

In contrast, for someone who is sharing with numerous other users a small cluster which is insufficiently equipped with memory, the problem of memory usage and reallocation is a major consideration; consequently, his or her resulting code will differ considerably from the code of the person using the number cruncher. As described on page 97, when program development in HPC was compared to the Formula-I car development, in every case an individual solution had to be found.

There are some well-known fundamental facts, like the computational environment in which someone is going to develop the code. In HPC the resulting code is in general much better adapted to the underlying hardware than it would be on an average desktop computer. Normally the number of users simultaneously working on that computer is also approximately known. The type of model to be processed and its input data can be analyzed in advance; thus, the the type of computation for solving the model is clear from the very beginning.

This all helps the coding person to come up with a good framework for the program to be written. But open questions remain that probably could be answered theoretically but the mathematical calculations involved are too demanding and/or time consuming. Instead, in such cases like those with the direct neighbors in the above example, the programmer or engineer has to balance different techniques, all of which may increase either the runtime or memory consumption of the program or show other negative effects. There is practically never an optimal solution; if one does exist, it is generally not affordable and/or feasible under normal working conditions or within the given financial or timely frame.

6.7.3 The Usage of Dynamically Allocated Arrays in the Wrapper

The initial starting point for the implementational work for the preprocessing was very similar to the situation described above. The bit arrays, as described already in section 6.6, starting on page 145, had been implemented already and it had turned out that they had some unfortunate drawbacks (as will be described in chapter 7, starting on page 201). The data structure of linked lists (see section 6.4 of this chapter, starting on page 128) also proved to be an ill-suited alternative. So a new data structure had to be found that matched the given specifications and prerequisites.

One major requirement for this new data structure was that it needed to be fast; thus direct access was obligatory. Unfortunately no perfectly suited basic data structure in C exists that is fast, simple to use and at the same time shows individual adjustability to a certain amount of data by being “close” to given hardware requirements. The only data structure to almost meet all these requirements is an array. However, the first attempt to implement bit arrays showed that a statically used array was not practicable at all, due to its inflexibility in regard to varying size.

Experiences with the bit arrays soon made it clear that it was not the size for an individual datum itself that was critical in regard to memory constraints but the overall memory allocation for the whole structure. An almost perfect solution would have been the implementation of a dynamically adjustable bit array. The overhead for this as well as the quite complicated routines to access the bit array made this solution impracticable.

The necessity for DAAs emerges several times during the preprocessing. The different locations are discussed in more detail in section 6.10, starting on page 178, further on in this chapter. Generally, through various resorting steps and different reassignments of nodes to elements, elements

to nodes, etc., a mixture between static standard-type arrays and DAAs had to be realized. An example is shown in Fig. 6.33, where an assignment between node-to-element is given. It results from the simple model that was depicted in Fig. 6.1 on page 120.

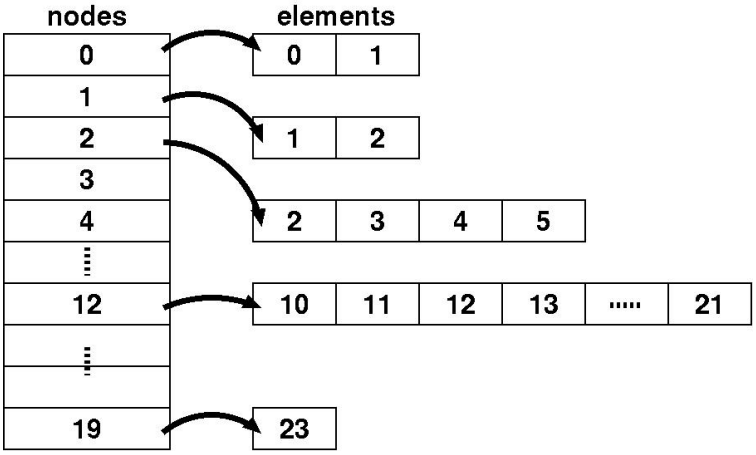


Figure 6.33: A node-to-element assignment using standard arrays and dynamically allocated arrays at the same time

From the example shown in Fig. 6.33 it can be seen that even in such a simple 2D model consisting of only 20 nodes and 24 elements, the total of elements for the nodes varies from 1 (at node no. 23) to 8 (node no. 12), for instance. It makes no sense to directly allocate 8 memory cells for all of the nodes. Rather, as in the wrapper program, each node number is assigned a second array that is realized as a DAA. If necessary, this can be resized accordingly, and if the initial space is sufficient, no actions are necessary at all. The connection between the static standard type of array in which the nodes are stored is made over a pointer. This data type was already introduced as a reference data structure in subsection 6.4.4, starting on page 132, with which the single list elements were linked together.

Here we have the same situation: It is not the node number that is stored in the static array but the pointer to a memory location where the specific DAA for that node number is held. The node number itself in the static array need not to be stored explicitly, as it results as the subcript of the static array. Subscript 0 to the static array of nodes is the pointer for node 0, subscript 1 results in the pointer for node no. 1, and so forth. Thus, the nodes themselves need not to be stored but result indirectly through the organization of the array and its access over the subscripts.

It should be noted that it is not practical to realize such a node-to-element structure or similiar correlations over a true static 2D array. The total number of nodes is known from the very start so that it is no problem to declare directly such a static array to hold the node numbers or at least access them indirectly through the subscripts, as described in the previous paragraph. But as one doesn't know in advance the true oscillation over the values of the number of elements per node, a realization as a DAA in the second dimension of the array makes no sense. Each time a reallocation is necessary, a great deal of information has to be shifted and copied back and forth when using a real 2D array structure. With the realization using a pointer to link the static array for the nodes with the DAA of the specific node, copy actions may still occur occasionally but then only the few values that are stored for one node number have to be shifted and copied rather than not all element information for all node numbers.

When now starting the program in the example above, with the node-to-element correlation, the static array is declared with the maximum node number, as this value is known from the very beginning when reading in the input data set for the model. The single DAAs for each node that will hold the according element information later on is declared and allocated upon program execution with a certain default value so that a first few elements can already be stored.

After various tests with the given input data sets that the author had at hand that varied greatly not only in size but also in the geometrical shape of the elements, it turned out that a basic size of 5 elements was a good starting point for these element DAAs. In case the DAA for a specific node began to run out of space, a new block of 5 storage cells was then "appended" through reallocation when a new datum for which there was no more space in the existing DAA had to be stored.

This construction with a static array linked over a pointer structure to a smaller DAA is a trade-off between memory consumption and runtime. Using arrays on the given hardware is the best possible data structure to assure fast processing not only on the computing nodes of the parallel computer but on most other architectures. On the other hand, by using the construction of a pointer reference to a DAA, memory usage is kept to a minimum so that even larger input data sets can be processed on machines that are poorly equipped with memory. The DAAs are still fast because they are simple; furthermore, direct data structures and the extra cost and time for dereferencing the pointer structure is small in comparison to the benefits using the array structures in general.

One drawback, nevertheless, is that now additional auxiliary variables have to be controlled and kept. But these are also realized as static arrays so that the additional time is kept to a minimum. A full discussion of the different alternatives of data structures that could be realized in the wrapper program with their advantages and disadvantages as well as the necessary trade-offs for one or the other version is given in chapter 7, starting on page 201.

Application Examples of DAAs in the Wrapper Code

As explained previously, allocation and reallocation of memory is a vital procedure throughout the whole preprocessing and thus, many parts in the wrapper code deal with these processes. This brief subsection now shows some code excerpts of the wrapper program as an example where allocation and reallocation routines are used.

One major problem already in the beginning of the whole preprocessing process is the fact that the total number of nodes and elements is not known until the input data files have been opened and the necessary header information obtained. (For the structure of the input data files, the reader can consult appendix D, starting on page 257.)

This means that something like a "virtual container" has to be created where the nodes and elements can be stored later on although the exact size is not known at the stage of the data declaration. (See the description in subsection 6.5.1, page 137ff). Only later in the course of the program flow does that number become known; at this point can the total memory space necessary for the variables/arrays, holding either the nodes and their elements or the elements and their constituing nodes, be allocated.

One important step in the wrapper is to store all the elements that are attached to one vertex. The following code examples all deal with this problem.

First we have to reserve memory not only for the array that shall be used for the assignment between nodes and elements, but also for numerous necessary auxiliary data structures that are

needed to administer the main data structure. The following shows a few examples of memory allocations in this situation:

```
nodes_max = (int *) malloc(total_of_nodes * sizeof(int));

if ( nodes_max == NULL ) {
  /* no memory could be allocated! */

  fprintf(stdout, "\nERROR: Couldn't allocate any memory for\
                     the variable \"nodes_max\"!\n");
  fprintf(stdout, "==> Exiting program ... now!\n\n\n\n");
  fflush(stdout);
  cleanup_before_exit();
  exit(0);

} /* end of nodes_max-if */

nodes_min = (int *) malloc(total_of_nodes * sizeof(int));

if ( nodes_min == NULL ) {
  /* no memory could be allocated! */

  fprintf(stdout, "\nERROR: Couldn't allocate any memory for \
                     the variable \"nodes_min\"!\n");
  fprintf(stdout, "==> Exiting program ... now!\n\n\n\n");
  fflush(stdout);
  cleanup_before_exit();
  exit(0);

} /* end of nodes_min-if */

nodes_amount = (int *) malloc(total_of_nodes * sizeof(int));

if ( nodes_amount == NULL ) {
  /* no memory could be allocated! */

  fprintf(stdout, "\nERROR: Couldn't allocate any memory for \
                    the variable \"nodes_amount\"!\n");
  fprintf(stdout, "==> Exiting program ... now!\n\n\n\n");
  fflush(stdout);
  cleanup_before_exit();
  exit(0);

} /* end of nodes_amount-if */

nodes_left_space = (int *) malloc(total_of_nodes * sizeof(int));

if ( nodes_left_space == NULL ) {
  /* no memory could be allocated! */

  fprintf(stdout, "\nERROR: Couldn't allocate any memory for \
                     the variable \"nodes_left_space\"!\n");
  fprintf(stdout, "==> Exiting program ... now!\n\n\n\n");
  fflush(stdout);
  cleanup_before_exit();
  exit(0);

} /* end of nodes_left_space-if */
```

What should definitely be noted here is the fact that one has to make sure that the program does not crash when there is a problem in reserving the storage space that is needed. All of the 4 example allocations in the excerpt above have the exception handling directly following the allocation. If one neglected such a problem, the program would end abnormally, generally with a *segmentation fault* and the user would have no idea of what had happened. With the exception handling the crashing itself cannot be prevented, as there is basically no memory space that can be asigned to the data structure, but one could regard this form of termination as a "controlled crashing." The user now at least is told that there was a problem during the action of memory assigment and the name of the data structure causing the problem is also given.

The arrays that have been created are not yet the DAAs but simple static arrays at this point. All of the allocations use a variable called `total_of_nodes`, which holds the number of nodes that can be found in the preamble of the specific input data file. With the `malloc` command as given above for each of the variables, a static, normal array is created and memory reserved in the size of `total_of_nodes`, and each of the "containers" of all the nodes has the size of an integer.

When assuming a total number of vertices of 1 mio. on a 32-bit architecture in our input data file, we had already consumed more than 4 x 1 mio. x 4 bytes = 16 mio. bytes (which is about 16 MBs) alone for these 4 different variables; this is just a small percentage of all variables or data structures needed thoughout the course of the wrapper program. Furthermore, we still haven't stored anything in these 4 example variables nor have we set up the DAAs yet. Up to this point, the memory space has been allocated but the content is still "undefined." (See also subsection 6.5.1 of this chapter, page 137ff.)

To prevent another program crash caused by accessing undefined memory contents, we next have to initialize our variables so that we have some arbitrary but defined content in our arrays. One has to make sure that the initialization values are not going to "influence" the later actions and true contents that the arrays are going to hold. The next C command is an example of what such an initialization could look like:

```
memset(nodes_amount, 0, (total_of_nodes * sizeof(int)));
memset(nodes_left_space, 0, (total_of_nodes * sizeof(int)));
```

The C command `memset` takes a defined portion of the memory, we go over the area of all nodes, indicated by `total_of_nodes`, and write a value of 0 into each of the integer "containers" (`sizeof(int)`) of each node. Now, at least we can be sure that when accessing one of the 4 data structures, we have a predefined value of 0 in it when reading it and the danger of a program crash due to undefined contents is banished.

The following longer C code listing now is an excerpt of the processing of the nodes of each element. When reading through it one will not only occasionally recognize the necessary exception handling but also the fact that the author also has kept the additional commands for the *debugging* and the *verbose commandline switch* in it. (For a full discussion of each of the commandline options, refer to section 6.15, starting on page 189.)

```
/* ------------------------------------------------------------
   Process nodes per element
   ------------------------------------------------------------ */

for ( i = 0; i < total_of_elements; i++) {
  /* read element for element and then process the nodes of this
```

```
   element */

if ( debug_mode || (DEBUG && !output) ) {
  fprintf(stdout, "\nNow processing element %d:\n", i);
  fprintf(stdout, "-------------------------\n");
  fflush(stdout);
} /* Ende des Debug-ifs */

for ( j = 0; j < element_node_amount[i]; j++) {

  if ( debug_mode || (DEBUG && !output) ) {
    fprintf(stdout, "\nNow processing node %d of element %d:\n",
            j, i);
    fflush(stdout);
  } /* Ende des Debug-ifs */

  node = element_nodes_listing[i][j];

  if ( nodes_left_space[node] == 0 &&
                    nodes_amount[node] == 0 ) {
    /* first time ever that you store an element for this node
       here. Memory for the first 5 elements has to be allo-
       cated! You need also to set the counters right! */

    nodes_left_space[node] = mem_space;

    /* set an appropriate initial value for the variable that
       holds later on the minimum element of the node! */

    if ( (total_of_elements + 1) <= INT_MAX ) {
          nodes_min[node] = total_of_elements + 1;
    } else {
      /* now we have a problem! There are more elements pre-
         sent in this mesh than our standard C integer can hold!
         If the program doesn't quit on its own take the machine
         maximum for an integer as the initial value for the
         min variable! */
      nodes_min[node] = INT_MAX;
    } /* end of total_of_elements-if */

    nodes_max[node] = -1;
    node_elements = (int *) malloc(sizeof(int) * mem_space);

    if ( node_elements == NULL ) {
      /* no memory could be allocated! */

      fprintf(stdout, "\nERROR: Couldn't allocate any memory \
                       for the elements of node %d!\n", node);
      fprintf(stdout, "==> Exiting program ... now!\n\n\n\n");
      fflush(stdout);
      cleanup_before_exit();
      exit(0);

    } /* end of node_elements-if */

    /* now link in the elements listing array at the correct
       location in the nodes pointer array! */

    elements_of_node[node] = node_elements;
```

```
} /* end of nodes_left_space-if */

/* now store the element in the node listing but make sure
   that there is enough space left in the array! */

if (  nodes_left_space[node] == 0 ) {
   /* no space left, realloc another portion of memory! */

   temp_array = (int *) malloc ( (nodes_amount[node] +
                        mem_space) * sizeof(int) );
   if ( temp_array == NULL ) {
     /* no memory could be allocated! */

     fprintf(stdout, "\nERROR: Couldn't reallocate new memory\
                        for the elements of node %d!\n", node);
     fprintf(stdout, "==> Exiting program ... now!\n\n\n\n");
     fflush(stdout);
     cleanup_before_exit();
     exit(0);

   } /* end of temp_array-if */

   nodes_left_space[node] = mem_space;
   memset(temp_array, 0, ((nodes_amount[node] +
                            mem_space) * sizeof(int)) );

   /* now copy the already stored elements of this node into the
      new array! */

   for ( k = 0; k < nodes_amount[node]; k++ ) {
      temp_array[k] = elements_of_node[node][k];
   } /* end of k-for-loop */

   free(elements_of_node[node]);
   elements_of_node[node] = temp_array;

   #if DEBUG
   if (!output) {
      printf("\nActual elements of node %d:\n", node);
      for (k = 0; k < (nodes_amount[node] + mem_space); k++ ) {
         printf("\t%d", elements_of_node[node][k]);
      } /* end of k-for-loop */
      printf("\n\n");
   } /* end of output-if */
   #endif

} /* end of nodes_left_space-if */

/* okay, now everything should be prepared so that you can store
   the actual element at the corresponding node to which it
   belongs! Find the right position of the element in the
   element array! */

if ( i > nodes_max[node] ) {
   /* the easiest case: the actual element is the biggest one
      found so far for the node! Append it at the very last
      storage position in the array! */

   nodes_max[node] = i;
```

```
    if ( nodes_amount[node] == 0 ) { nodes_min[node] = i; }

    nodes_left_space[node] = nodes_left_space[node] - 1;
    elements_of_node[node][nodes_amount[node]] = i;
    nodes_amount[node]++;

  } else {

    if ( i < nodes_min[node] ) {
       /* the actual element is the smallest one found so far for
          this node. Now move all already stored elements for this
          node up one position and store the new-found element at
          position 0 in the array. */

       for ( k = nodes_amount[node]; k > 0; k--) {
          elements_of_node[node][k] = elements_of_node[node][k-1];
       } /* end of nodes_amount-for-loop */

       /* now as every element has been moved up one position,
          insert the new element at the beginning and adjust all
          counters! */

       nodes_amount[node]++;
       nodes_left_space[node]--;
       nodes_min[node] = i;
       elements_of_node[node][0] = i;

    } else {
       /* okay, element is bigger than smallest one found so far
          and smaller than the biggest one. Find the correct
          position and from there on move all remaining bigger
          elements one position up and insert the new element.
          Adjust all counters! */

       k = 0;
       while ( i > elements_of_node[node][k] ) { k++;}

       /* at position k we have found the correct position for the
          new elment i. Now shift up the remaining elements one
          position. */

       temp_pos = k;

       for ( k = nodes_amount[node]; k > temp_pos; k-- ) {
          elements_of_node[node][k] = elements_of_node[node][k-1];
       } /* end of nodes_amount-for-loop */

       nodes_amount[node]++;
       nodes_left_space[node]--;
       elements_of_node[node][temp_pos] = i;

    } /* end of nodes_min-if */
  } /* end of nodes_max-if */

  if ( debug_mode || (DEBUG && !output) ) {
    fprintf(stdout, "\nActual minimum element of node %d = %d\n",
            node, nodes_min[node]);
    fprintf(stdout, "Actual maximum element of node %d = %d\n",
            node, nodes_max[node]);
```

```
        fflush(stdout);
      } /* Ende des Debug-ifs */

    } /* end of j-for-loop */
  } /* end of i-for-loop */
```

It is not the goal of this thesis to enable the reader to completely understand such an advanced C code listing. But just by scanning over the program excerpt given above, he or she should gain a rough idea of what is being done by reading the comments between the different C statements.

At some point in this code listing, the reader will see a query where a variable `nodes_left_space[node]` is checked if it has a value of 0 or not, after it has been initialized and space has already been allocated. In case it equals 0, no more storage space in the array for this specific node is available and now new space has to be appended. This is exactly where the DAAs that we had talked about in this section are hidden.

But it is impossible to simply just append new memory to the existing array. It is not clear if there is enough memory space available directly behind the already allocated portion of the array. Therefore a temporary array structure is used to reallocate new memory space, as in the following C statement:

```
temp_array = (int *) malloc ( (nodes_amount[node] +
                mem_space) * sizeof(int) );
```

This line of C code now allocates new storage space of the size of the already existing array `nodes_amount[node]` plus a portion of extra space of the size `mem_space`. The fact that integers are going to be saved in this array is indicated by the keyword `sizeof(int)`. The reader may be interested to learn that nowhere is a precise number given; only variables are used. This is good C programming which permits values to be changed throughout the program by just changing the value of a variable once. In case the total numbers had been coded into the C statements directly, the whole program must be controlled, searched and altered in case another value is to be used.

Here in this example the amount of extra memory is coded in the variable given by `mem_space`. This is set once to a value of 5. In case a different value should be used when, for example, it turns out that 5 is not an optimal value, only one single line of code containing `mem_space` has to be changed and not all the locations where now `mem_space` is a place holder for the value of 5.

After the reallocation of the new memory for the array of the specific node, all auxiliary variables have to be adjusted accordingly and – very important – this little extra space that was just appended has to be initialized too. In case it will not be completely filled, this precaution ensures that no values are stored in it in the event of accessing the extra memory locations later on. The following lines give an example of the above, as they can also be found within the previous long listing:

```
nodes_left_space[node] = mem_space;
memset(temp_array, 0, ((nodes_amount[node] +
          mem_space) * sizeof(int)) );
```

Finally, when all the necessary steps have been accomplished, the original array, which was at some point too short, is now copied into this new memory location with more space for more future elements. This is done in the following lines of C code:

```
for ( k = 0; k < nodes_amount[node]; k++ ) {
    temp_array[k] = elements_of_node[node][k];
} /* end of k-for-loop */
```

As memory consumption is one of the major concerns in the preprocessing, an important step is still missing: One must make sure that the memory location where the array was stored and now copied to a new memory location is given up and freed for further use. If this action is neglected, precious memory space is wasted and can no longer be used. This will be explained in further detail in the upcoming section of *Valgrind*.

After the original array, which has grown in size, has been resized, the normal course of the program can continue and one can resume storing additional values in it as if nothing had happened.

Here in this example just one case was shown where such DAAs had to be deployed in the wrapper. As many different assignments of relationships are necessary through the entire preprocessing, these kinds of actions are required at various points throughout the wrapper. The previous example, containing a short C code excerpt representative of many other cases, has hopefully indicated to the reader the complexity of such data structures but at the same time the necessity of thoroughly testing them.

6.8 Valgrind

As memory consumption, organization and management was one of the major issues of the whole preprocessing, it was of utmost importance to assure the best possible outcome of the implementation of the wrapper program. Unfortunately, it often happens that despite great diligence, many memory leaks and an unsloppy memory usage occur.

In this situation it was obligatory to have the code checked thoroughly to prevent such circumstances and errors. The problem often arises that such memory leaks and errors are hardly detected. For this reason a good debugging tool is needed. The commonly used debugger in the Unix and Linux world, called `gcc` does not offer such a functionality, so that an alternative tool was needed.

One of the best known debugging tools is the tool suite called *Valgrind*. The Valgrind distribution includes six production-quality tools, one of which is a memory error detector. It runs on many of the current Linux platforms. (The other profilers, detectors and experimental tools, also offered in the Valgrind framework were not needed for this work.) Valgrind is available as Open Source/Free Software, and can freely be obtained under the GNU General Public License, version 2. Valgrind maintains a very well-organized and up-to-date website. (See [Valgrind] for more information about the Valgrind distribution.)

6.8.1 Memcheck – The Valgrind Memory Profiling Tool

One of the best-known tools of the Valgrind distribution is clearly the memory profiling tool called *Memcheck*. It can detect many of the typical memory errors like

- overrunning heap block boundaries
- reading or writing already freed memory
- using values before they have been initialized
- incorrect freeing of memory
- memory leaks

just to name the most common ones.

To use the Valgrind memory profiler for a detailed examination of the code, one must compile his or her program with an additional `-g` commandline switch. This ensures that the debugging information is set to "on" so that Memcheck's error messages include the exact line numbers of the program code later on when analyzing.

6.8.2 Using Valgrind's Memcheck in Practice

For most of the readers who have presumably never used a debugging tool under Unix or Linux nor have intensive programming experience, a very detailed discussion about the benefits of such a tool like the memory profiler Memcheck would be difficult to follow or understand. For this reason it would be wise to show Memcheck's strength using a few significant examples that demonstrate which main errors can easily be detected with such a profiling tool.

Detection of Memory Leaks

One of the typical memory errors are the so-called *memory leaks*. In subsection 6.4.4, starting on page 132 in this chapter, the memory was described as having an endless row of storage boxes. Using this example, a memory leak could now be illustrated as having taken a couple of boxes out of this long row and unfortunately "misplaced" them somewhere and not being able to remember anymore where they have been put. In reality, memory cannot be "lost;" rather, the references to these memory locations have simply been lost by the underlying system.

To demonstrate how easily such a thing can happen to even the experienced programmer, the following little C program was created and run using Memcheck:

```
#include <stdlib.h>
#include <stdio.h>

#define MAXIMUM_OF_NODES 1000000

struct Node_Coordinates
{
  double x_coordinate;
  double y_coordinate;
};


int
main(int argc, char **argv)
{
  struct Node_Coordinates *new_node;
  int i;
```

```
  /* Create a number of new points */
  for (i = 0; i < MAXIMUM_OF_NODES; i++) {

    new_node = (struct Node_Coordinates *)
                malloc(sizeof(struct Node_Coordinates));

    /* Now do something with the new node... (left out here) */

  }

  /* Free memory associated with the node */
  free(new_node);

  return EXIT_SUCCESS;
}
```

The short C program above allocates memory space for a data structure that holds the x and y coordinate of a node in a two-dimensional space. Using a so-called loop construct, an auxiliary counter variable `i`, which is initialized to 0 at the beginning, is incremented by a value of 1 each time the loop statement is repeated. Within this loop a new node variable is created during each loop iteration, memory space is allocated for its x and y coordinate and something (which is of no importance in this example) is done with this node variable. The loop is repeated as many times as the counter variable `i` is smaller than a macro called `MAXIMUM_OF_NODES` which was set to an arbitrary value of 1,000,000 in the example above, but could hold any other value.

When the loop has terminated, a new node with two coordinates had been created 1,000,000 times and thus, memory space was reserved for their x and their y coordinates the entire time.

An experienced programmer would "sigh with relief" spotting the second-to-last line of the C program where the allocated memory space is finally freed using the C statement `free(new_node);`. So everything seems to be in perfect order. – Really? Now let us see what the profiling tool Memcheck thinks about this little exemplary C program.

We assume that the C code above was stored under the file name `leak.c`. To be able to use the Valgrind tool we have to compile the program with an additional commandline switch `-g`, as explained previously. When the executable had been created by the compiler and linker, we finally can call Memcheck by the following commandline statement:

```
valgrind --leak-check=yes leak
```

The resulting output of Memcheck reads as follows:

```
==22649== Memcheck, a memory error detector.
==22649== Copyright (C) 2002-2008, and GNU GPL'd, by Julian Seward
          et al.
==22649== Using LibVEX rev 1878, a library for dynamic binary
          translation.
==22649== Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
==22649== Using valgrind-3.4.0-Debian, a dynamic binary
          instrumentation framework.
==22649== Copyright (C) 2000-2008, and GNU GPL'd, by Julian Seward
          et al.
==22649== For more details, rerun with: -v
```

```
==22649==
==22649==
==22649== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 13
          from 1)
==22649== malloc/free: in use at exit: 15,999,984 bytes in 999,999
          blocks.
==22649== malloc/free: 1,000,000 allocs, 1 frees, 16,000,000 bytes
          allocated.
==22649== For counts of detected errors, rerun with: -v
==22649== searching for pointers to 999,999 not-freed blocks.
==22649== checked 59,532 bytes.
==22649==
==22649== 96 bytes in 6 blocks are possibly lost in loss record 2 of 3
==22649==    at 0x402401E: malloc (vg_replace_malloc.c:207)
==22649==    by 0x80483F9: main (leak1.c:19)
==22649==
==22649==
==22649== 15,999,872 bytes in 999,992 blocks are definitely lost
          in loss record 3 of 3
==22649==    at 0x402401E: malloc (vg_replace_malloc.c:207)
==22649==    by 0x80483F9: main (leak.c:19)
==22649==
==22649== LEAK SUMMARY:
==22649==    definitely lost: 15,999,872 bytes in 999,992 blocks.
==22649==      possibly lost: 96 bytes in 6 blocks.
==22649==    still reachable: 16 bytes in 1 blocks.
==22649==         suppressed: 0 bytes in 0 blocks.
==22649== Reachable blocks (those to which a pointer was found)
          are not shown.
==22649== To see them, rerun with: --leak-check=full
          --show-reachable=yes
```

When checking the output of Valgrind, suddenly the world is no longer in apple-pie order, as Memcheck unvarnishedly reports that we have lost about 16 MB of memory space. What has happened? Haven't we explicitly called the `free` routine to avoid precisely this error?

The solution can quickly be detected in about the middle of the Memcheck messages. There we read that the C statement `malloc` was used 1,000,000 times but the `free` statement just once. When now examining the program more closely and with the necessary knowledge about hardware and memory organization, all of a sudden, it becomes quite evident what has happened: We allocated 1 mio. times memory space for the 2D node coordinates but when we scrutinize our C program we realize that we just freed one single node. The `free` statement in the second last line of the example frees the very last node, but what happens to the other 999,999 nodes and their coordinates? The memory space for them is still reserved in the memory and as we haven't stored a pointer or any other reference to access them at some point, the information about where to find them is now irretrievably lost.

In the simple example above we lost 16 MB of memory although we ostensibly did everything we could to prevent the error that we nevertheless caused in the end. As one now thinks about the fact that there are many more variables in a program and especially in C many pointer structures are used in general, the memory space that is consumed this way adds up to a multiple of the little bit of memory that we had just used in the example. Without great diligence, care and knowledge, precious storage space is lost in no time for the further program execution.

Now having detected this memory leak, what would be our necessary and obligatory reaction and how can we prevent something like this happening in the future? The solution is as straightforward

as it can be: As we have created 1 mio. nodes and their coordinates, we simply have to free them again. We must store the reference to these nodes at some point in the program and when they are no longer needed, free them one after another, using, for example, a loop statement like the one in the exemplary C program above.

Access of Undefined Memory Space

Our second example illustrates another quite common error which might result in very pesky bugs that often are hard to detect, as they never show the same results with each program run. This type of error was already briefly mentioned in subsection 6.5.1 in this chapter, starting on page 137. We again use a simple, exemplary C program to demonstrate how easily such a mistake can happen even to experienced programmers:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct Node_Coordinates
{
  double x_coordinate;
  double y_coordinate;
};

int
main(int argc, char **argv)
{
  struct Node_Coordinates *new_node;

  /* Allocate memory and initialize it */
  new_node = (struct Node_Coordinates *)
             malloc(sizeof(struct Node_Coordinates));

  memset((void *)new_node, 0, sizeof(struct Node_Coordinates));

  /* Now do something with the new node ...*/

  free(new_node);

  /* now we do many other things in our program here ...

  /* We all of sudden remember our new node and decide to use
     it again */

  new_node->x_coordinate = 2.0;
  new_node->y_coordinate = 3.5;

  return EXIT_SUCCESS;
}
```

We already know from the first exemplary C program the C construct of `Node_Coordinates`. In this program here we also allocate memory space for a new 2D node with its x and y coordinates and do something with it in the further course of the code. As we are experienced and dutiful programmers, we also free the node when we no longer use it. We name our little program `access`, compile it using the necessary switch `-g` and call Valgrind again with the following command:

```
valgrind ./access
```

It should be noted that this time we use Valgrind without the *leak check* option, unlike in the previous example. This leak check option is not necessary here in this case and the second example also shows quite clearly that Valgrind can find all kinds of other errors, not only memory leaks. The resulting output by Valgrind of our `access` program is as follows:

```
==10432== Memcheck, a memory error detector.
==10432== Copyright (C) 2002-2008, and GNU GPL'd, by Julian Seward
          et al.
==10432== Using LibVEX rev 1878, a library for dynamic binary
          translation.
==10432== Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
==10432== Using valgrind-3.4.0-Debian, a dynamic binary
          instrumentation framework.
==10432== Copyright (C) 2000-2008, and GNU GPL'd, by Julian Seward
          et al.
==10432== For more details, rerun with: -v
==10432==
==10432== Invalid write of size 8
==10432==    at 0x8048453: main (access.c:31)
==10432==    Address 0x419c028 is 0 bytes inside a block of size
             16 free'd
==10432==        at 0x4022E3A: free (vg_replace_malloc.c:323)
==10432==        by 0x8048449: main (access.c:24)
==10432==
==10432== Invalid write of size 8
==10432==        at 0x804845E: main (access.c:32)
==10432==    Address 0x419c030 is 8 bytes inside a block of size
             16 free'd
==10432==        at 0x4022E3A: free (vg_replace_malloc.c:323)
==10432==        by 0x8048449: main (access.c:24)
==10432==
==10432== ERROR SUMMARY: 2 errors from 2 contexts (suppressed:
          13 from 1)
==10432== malloc/free: in use at exit: 0 bytes in 0 blocks.
==10432== malloc/free: 1 allocs, 1 frees, 16 bytes allocated.
==10432== For counts of detected errors, rerun with: -v
==10432== All heap blocks were freed -- no leaks are possible.
```

It should be quite obvious what has happened here. Valgrind correctly reports invalid write statements (in about the middle of the Valgrind output) in our program. We had regularly freed our 2D node variable in the middle of our second C example code. Towards the end of the program we seemingly forgot about this fact and started working with the node coordinates again by assigning them new values. We seem to have been very lucky that our program didn't crash, and some values must have been assigned to these two coordinates. But which ones? Officially the two coordinates no longer exist in the memory.

Of course this example was constructed and the error is directly obvious. But normally C programs resulting from complicated model computations are abundant in code lines and consist of various different files. Often many people work simultaneously on the project and unfortunately commenting is cheerfully "neglected" by most programmers. It then can easily happen that one is accessing a variable which had previously been freed by someone else.

6.8.3 Validation of the Wrapper using Valgrind

The C code that resulted for the preprocessing of the input data sets was thoroughly tested with Valgrind and especially with its memory profiling tool Memcheck. Already when checking it just one time at the very beginning it was clear that only "good C knowledge" and some C programming experience was not sufficient, especially for such delicate remits and tasks as a sophisticated and good memory consumption and management.

The first of the two examples above was one of the typical errors that occured relentlessly despite a great deal of diligence during the coding process. Actually, this example was one of the errors in the wrapper that was found using Memcheck. Fortunately, the output and messages issued by Valgrind are very detailed and helpful and if someone uses the commandline switch `-v` when calling Valgrind, the resulting report is even longer and much more detailed than the example output above.

In this way, the more than 11,000 lines of C code that are now finally hidden behind the wrapper in its current version could be checked for all kinds of errors that are not obvious at first sight. In the end, when finishing the debugging and testing phase of the actual version of the wrapper, not a single error could be found in the C code, no matter what switch and level of "nastiness" was chosen when using Valgrind. Reproducing the full output of Valgrind that resulted when having it check the wrapper code is impossible here, as it is simply too long, due to the many different libraries, routines, and other things embedded or linked into the code.

It should be noted, however, that the author was not only quite astonished but also almost amused to see that neither of the two domain decomposition tools, *Jostle* (see section 5.4 in chapter 5, starting on page 107) nor *Metis* (see section 5.5 in chapter 5, starting on page 112) seemingly had been tested with a tool like Valgrind. The only errors that were reported in the end when checking the wrapper code with Valgrind resulted from the program files of these two partitioning tools which were linked into the wrapper program.

6.9 Input Data Processing

Up to this point of the thesis, the reader has learnt a great deal about the implementation itself as well as obligatory and necessary steps in regard to the general task of accomplishing the preprocessing. In the now upcoming three sections, the overall structure of the wrapper code will be discussed in more detail. One can divide the wrapper code into three major blocks: The first one deals with the pure processing of the input data sets of the model. The second takes care of the preprocessing of this input information in such a way that it can be fed into the domain decomposition tools, as described in detail in section 5.4 of chapter 5, starting on page 107. The third part then processes the information returned from the domain decomposition tools and rearranges and preprocesses it in such a way that the output files which are needed later on for the parallel model computation can be created and written. In the section below, the preprocessing of the input data files is now going to be explained in more detail.

As can be seen in appendix D, starting on page 257, the input files consist of a short preamble or "header" with some "hieroglyphical lines" at the beginning of the input files. The next bigger part is then a long listing of the geometrical coordinates of the vertices of the model. The last major block consists of a long listing of all elements in the model and the nodes of which they are built.

The preamble is as short and "ugly" as it is important. In it the vital information of the total number of vertices and nodes in the model is coded. Furthermore, a version number and additional information is stored, the latter being of no further importance for the preprocessing. Instead, the version number gains increasingly in significance, when more and more different types of input files have to be processed in a later stage of the project. (For this, see also the short discussion in section 7.3 of chapter 7, starting on page 7.3).

After processing the header of the input files, the appropriate data structures, which are needed in the further course of the wrapper, as discussed previously in this chapter, can finally be allocated and initialized.

Next follows the block with the geometric coordinates of the vertices in the model. At the current stage of the project, the coordinates were of no further importance and normally one could have easily just skipped the processing of these lines in the input file. Nevertheless, the author decided not to do so but to process the information given there. This way, no major changes are necessary in case this data has to be processed at a future time.

The last part in the input data file is the listing of the elements and their nodes that set them up. Here it is of utmost importance to check the geometric type of the elements. Only with this information can the correct number of nodes building the element be recognized and further processed.

The problem here is to determine the correct column in the line that is given for each element. In the course of the project this structure constantly changed so that with every new "edition" of an input file recognizing the appropriate column in the line began to become a challenging task. This finally led to the decision of expanding the current version of the wrapper from `domdec_v03_2.c` to a future version `domdec_v03_3.c` to take care of this constantly growing problem with the different input file structures. (See also section 7.3 of chapter 7, starting on page 7.3.)

The second difficulty with the processing of the element information is the fact that one doesn't know in advance how many vertices are setting up the element. A triangular-shaped element is built by 3 nodes, whereas a linear-shaped element only requires two vertices, for instance. Only after having read the column with the geometrical shape information of the element does one know how much storage space is needed to save the vertex information of the element. Here the deployment of the DAAs, as discussed in detail in section 6.7 of this chapter, starting on page 158, is obligatory.

In Fig. 6.1, at the beginning of this chapter on page 120, a little example model was given, consisting of 20 vertices and 24 elements. The input file for this example can be found in appendix D on page 258. After having processed this little file, the corresponding data structure holding the element-to-node information will look like the one depicted in Fig. 6.34.

It should be noted that the node information per element is sorted in increasing order in the DAA as shown in Fig. 6.34. When checking the appropriate input file on page 258, one will realize that the node information for each element is not sorted at all. Already the entry for element no. 0 shows a listing of 3 nodes with numbers 0, 6 and 5, whereas the nodes for element 0 listed in Fig. 6.34 correctly have an increasing order. Normally, one could simply take over the information provided in the input data file as is, but this just postpones the problem of sorting to a later stage. It is wise to take care of the sorting already at this point of the process so that the complex steps following the input processing do not get even more complex than they are already.

In Fig. 6.34 each element in the static array holding the total of elements was assigned another small array which has a size of 3 nodes by using a reference. This regular structure results just

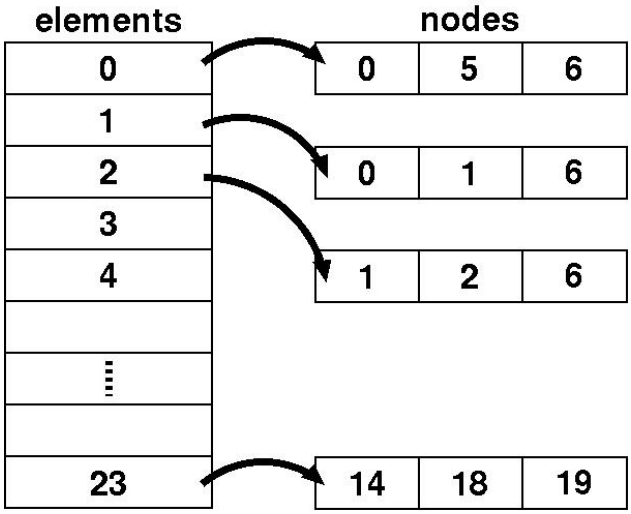


Figure 6.34: Example data structure holding the node information per element of a simple input file consisting of 20 vertices and 24 elements

from the fact that the whole input file consists of homogenously shaped, triangular elements. They require only 3 entries per element. But there are also cases possible where each element might be differently shaped in comparison to its predecessor and then the DAAs assigned to each element will vary greatly in size and will not consist of only 3 nodes as in the given example.

After having processed the information of the element listing, all necessary input data is stored. This is the first of the three major parts of the wrapper and preprocessing process. The next section will deal with the handling of the input data sets to prepare them for the domain decomposition tool calls.

6.10 Preprocessing of Model Data

This section deals with the necessary steps for the preparation of the input data in such a way that not only the adjacency graph can be set up but that the information can also be fed into the different domain decomposition tools. It can clearly be said that this part of the wrapper code is definitely the most complex one. Here numerous reassignments of relationships of the data have to be made. Unfortunately, the complexity in this block of the wrapper code increased in huge dimensions from the very second that a transition from bit arrays to DAAs, as described in sections 6.6 and 6.7 of this chapter, starting on page 145 and 158, respectively, was necessary.

The major problem one has to deal with in this part of the wrapper code is the fact that the input data file has an element-to-node relationship but for setting up the adjacency graph and the input to the domain decomposition tools, an element-to-element relationship is required. In the input data file an unsorted listing of the configuring nodes per element is stored. When reading in this information, as described in section 6.9, the resulting data structure looks like that depicted in Fig. 6.34.

As described in section 6.2 of this chapter, starting on page 120, the goal of the adjacency graph is to determine which element is a direct neighbor to another one. Unfortunately, one cannot get this information from such a data structure, as it results after having processed the input data file. As a

consequence, two more reassignments have to be carried out before the necessary information can be fed into the domain decomposition tools.

As the requirement of being a direct neighbor to an element is based on the fact that they share a mutual vertex, the first reassignment that has to be done as an intermediate step is saving a new relationship which tells which elements share all the same vertex, a node-to-element relationship. This resulting data structure was already introduced in subsection 6.7.3 of section 6.7, starting on page 161. There it served as an example for explaining the DAAs and was depicted in Fig. 6.33 on page 162.

To set up this new assignment, the element-to-node relationship resulting from the input-file processing has to be taken and now each node that is saved for the elements there has to be examined to determine which other elements are also attached to it. This new information has to be stored in the intermediate data structure, as shown in Fig. 6.33 on page 162.

As one can see, the number of elements per node can vary fairly. In the given example, shown in Fig. 6.33, the total per element ranged from 1 for node no. 23 to 8 for node no. 12, for instance. Here, the necessity for the DAAs, as explained in section 6.7, starting on page 158, is quite obvious. At this point it also becomes clear that it is wise to have the nodes sorted directly from the very beginning in the element-to-node data structure. This way the additional auxiliary variables which are needed to administer these DAAs can be kept to a minimum. For this reason the node-to-element assignment is also directly stored in an assorted way.

Having reached this point in the course of the program we can finally set up the adjacency graph using the intermediate data structure of the node-to-element assignment. Recalling the procedures explained in section 6.2 of this chapter, starting on page 120, will make it clear what necessary steps one has to take care of when setting up the element-to-element relationship resulting from the adjacency graph.

As two elements are adjacent to each other when they are attached to the same vertex, one now has to step through the static array that stores the nodes of the node-to-element relationship and examine all elements of this specific node. In case there is just one single element stored there for this node, then no direct neighbor for this element exists. In case the number of elements stored for each vertex is larger than 1, then the whole list of elements has to be processed. Elements are now always taken pairwise, and this vicinity has to be stored for each of them.

Let us assume we are processing the two elements of node no. 2 in Fig. 6.33 on page 162, i.e., elements no. 4 and no. 5. As they are adjacent because they both share node number no. 2, we have to store element no. 5 in the listing of directly adjacent elements for element no. 4 and must do so vice versa for element no. 5.

It should be noted here that in case an adjacency graph has to be set up that uses unidirectional edges and not bidirectional edges as was used here, the processing would be slightly different. The same holds true for the case in which weights have to be processed additionally to the node and element information in the input file. (See also chapter 7, starting on page 201 for more information about this topic.)

The necessity of DAAs at this point should be clear, as before the processing of the node-to-element data structure, the number of elements each element will have as direct neighbors, is unkown. This number can vary greatly, as discussed previously in this chapter.

When setting up the element-to-element data structure the elements are again directly sorted. Unfortunately this requires a resorting in form of a right shift of all entries, should the new element

number to be inserted be smaller than the first element number already stored in the array. This definitely means an extra overhead for the additional shift operations. Still, directly sorting is less time consuming than slowly filling the element listing and putting off the sorting to a later time, at which point it has to be done from the beginning to the end.

After having set up the element-to-element listing, this information can now easily be transformed into the *Chaco format* (see also subsection 5.4.1 of chapter 5, starting on page 107) and fed into the appropriate domain decomposition tool. The exact method of calling the tools and what must be considered, etc., was already explained in detail in subsections 5.4 (see page 107) and 5.5 (see page 112) of chapter 5, respectively.

6.11 Post Processing and Outputting of Model Data

In the last major part of the wrapper code, the data that resulted from the return of the domain decomposition tools has to be further processed and prepared for later actions. Before calling either *Jostle* or *Metis* (see subsections 5.4 (page 107) and 5.5 (page 112) of chapter 5, respectively), the original input data sets have to be reassigned various times until finally an element-to-element relationship is established which represents the adjacency graph.

Unfortunately further reassignments of the resulting data from the domain decomposition tools are again necessary to be able to write the required output files. The information that is returned by Jostle as well as Metis is a static array the size of the total of elements in which for each element the corresponding subdomain number will be stored as an integer value. This is depicted in Fig. 6.35.

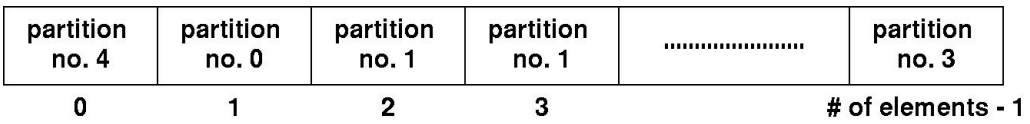


Figure 6.35: Example of the data structure that holds for each of the elements the subdomain number it was assigned by one of the domain decomposition tools

The required output file, called the `ddc` file, needs an assignment of inner and boundary nodes according to the partitions. Therefore, intermediate data structures are again needed to finally succeed in writing the output file.

The first assignement which has to be made, employing the static array returned by the domain decomposition tools, is a partition-to-element relationship. This is accomplished using a data structure with DAAs again, as they had already been applied previously in the preprocessing of the input data, etc. The partitions are stored in a static array and over references DAAs are attached to each of the array members that hold for each of the specific partitions the elements that belong to it. The DAAs are necessary here, as the total number of elements for each of the partitions can not be foreseen initially and results only after the processing of the static array returned by the domain decomposition tools.

In chapter 5 a partitioning of a model consisting of 48 elements was shown in Fig. 5.6 on page 95, as it resulted using Jostle as a domain decompositioning tool. For partition no. 0, which is the pink one, the elements with the numbers 0, 1, 2, 3, 8, 9, 10, 11, 16, 18, 19 and 24 would be stored in the DAA that were attached to partition no. 0 in the static array of this auxiliary data structure.

One should recall at this point that in the static array of the partitions, it is not the partition numbers themselves that are stored but the references to the single DAAs that belong to a certain partition. The partition number results straightforwardly, using the subscript to access the field in the static array, which would be, for example, `domain_elements[0]` for partition no. 0, `domain_elements[1]` for partition no. 1, and so forth. In the example shown in Fig. 5.6 on page 95, the static array would have a size of 4, as 4 subdomains had been created by Jostle. And each of the DAAs per partition consisted of 12 elements stored within it.

This auxiliary data structure can now serve to make a connection between the subdomains and the node numbers. Therefore, a further auxiliary data structure is needed, again combining a static array and DAAs, which assigns to each vertex in the model the corresponding partition number it is located in. The difficulties which arise here come from the fact that some of the nodes in the model might be shared by various partitions when they lie directly on the border of these subdomains. This was discussed in detail in section 5.1 of chapter 5, starting on page 89. (The reader can also have a look at Fig. 5.8 on page 96).

To obtain the information about which node is stored in which partition, one now has to step through each of the lists for the elements that were saved for the partitions and find the vertices that build the specific element. The nodes then will be assigned the same subdomain number as the element holds.

In Fig. 6.36 the data structure is shown that would result when using the model with the 48 elements in Fig. 5.6 on page 95, when setting up the auxiliary data structure described above.

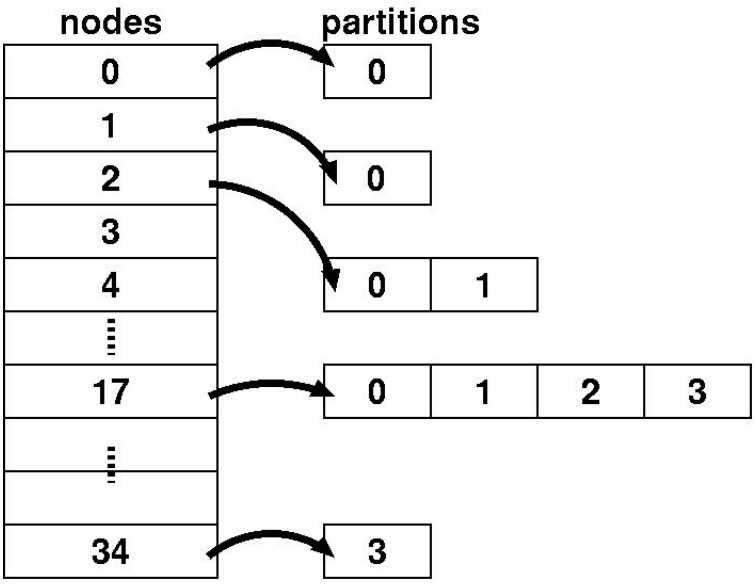


Figure 6.36: Example data structure that holds the assignment of the subdomain for each node in a model using DAAs

One additional transformation is needed so that the output file can finally be written. With the data structure depicted in Fig. 6.36, we just have an assignment of nodes to partitions but we do not yet know if the node is an inner node, i.e. having just one explicit domain number, or if it is a boundary node. In the latter case, the node will be assigned two or more partition numbers.

The final two data structures can now be obtained easily by processing the aforementioned auxiliary data structure with the node-to-partition relationship. Here, one simply has to check the

numbers of subdomains in the DAAs that are stored for each of the nodes held in the static array. In case the node has only one partition element stored in its DAA, it is an inner node; if it has more than one subdomain in the DAA, it is a boundary node.

The last two missing data structures that need to be constructed using the little algorithm described above will both consist of a static array that runs over the number of subdomains. For each of the partitions in this static array, either all inner nodes or all boundary nodes will now be stored in a DAA.

Now all the information needed for finally writing the `ddc` file is available. The interested reader can see the structure of such output files in appendix B, starting on page 241, where various examples of such `ddc` files are given.

6.12 Statistic Functions

During the development and implementation phase of the wrapper software the question quickly arose as to which of the domain composition tools that are deployed for the preprocessing should be used or produces better results (see chapter 5, and especially sections 5.4 and 5.5, respectively, for more information about the domain decomposition tools). And in regard to this important issue the next problem became evident: How should one "measure" success or failure of such a domain decomposition tool? What parameters should be used to compare the tools with each other?

Already in this regard the two tools that are currently implemented into the wrapper code, *Jostle* and *Metis*, differ enormously. *Jostle* allows the programmer and developer to directly retrieve specific parameters that are written during the domain decomposition process. Furthermore, the existing manuals give a short but still detailed enough overview of the functionality and the statistic parameters that can be accessed. In comparison to this, *Metis* does not offer any of this functionality or additional information. Here, the necessary background for interpreting the retrieved data has to be sought tediously in the sparse information material provided with the installation package. Important parameters that are essential for the interpretation of the quality of the domain decompostion have to be implemented and written manually by the programmer or software engineer.

Another difficulty in comparing the two domain decomposition tools is the fact that the two work with completely different underlying algorithms (see also section 5.2, starting on page 99) and have varying and differing imbalance tolerances for the algorithms implemented.

6.12.1 Statistic Information Offered by Jostle

For a comparison of the two tools on a neutral basis, simple parameters that show "hard facts" can be used. Two of such aspects are the size of the smallest and biggest subdomain that result after having carried out the domain decomposition in regard to the original input data set. Jostle allows one to retrieve this information indirectly. It is not possible to store such important data directly into program variables; rather, it has to be saved into a separate file by redirecting the *Jostle* terminal output during the processing of the input data after having called *Jostle* from within the wrapper software.

When saved as a simple text (ASCII) file, for example, the given information can be analyzed further. An example output produced by *Jostle* and then saved in such an extra file (see also Appendix B on page 241 for more on typical output files of the wrapper) is:

```
JOSTLE DOMAIN DECOMPOSITION RESULTS:
====================================

Topology is uniform (P = 3)

Graph:  10 nodes, 9 edges
        node weight = 10
        edge weight = 9

FINAL RESULTS:
Balance: 1.000  Largest: 2 (4)  Smallest: 0 (3)
Cut edges: 2
```

In this example case, 3 subdomains had been created and the largest one was the partition no. 2 which contained 4 elements, whereas the smallest partition was no. 0 with only 3 elements. (It should be noted here that often in computer science, counting starts with 0 instead of 1. Here, partition no. 2 is the 3rd subdomain produced, and partition no. 0 is the first of the three subdomains created.)

In addition to the smallest and biggest subdomain that resulted from the domain decomposition, more information is displayed, like the number of nodes and edges of the original model. As the current version of the wrapper software does not yet work with separate weights for vertices and edges, a node and edge weight of 1 is assumed as default value. Therefore, in the example above, the node weight and the edge weight are computed with 10 and 9, respectively (10 nodes with a node weight of 1 result in a total node weight of 10).

Two other very important parameters are also displayed in the previous example: the number of cut edges, which is 2, and the balance factor that resulted in 1.0.

The number of cut edges is important, as it indicates how many elements are shared by two or more subdomains. This means an additional overhead due to communication and information exchange, as already explained at the beginning of this chapter.

The balance factor is a parameter which results in dependence of the underlying algorithm which is used to detect the different partitions. Its calculated value also depends indirectly on a certain imbalance factor that is tolerated by the algorithm. Typical values for this tolerance range from 1 % up to 5 %.

The *Jostle* manual states that the underlying domain decomposition algorithms of *Jostle* try to create a perfect load balancing while optimizing the partitions. In case a certain imbalance tolerance is allowed, a slightly better optimization is achieved. Unfortunately, there is no further information provided in the documentation or the manuals as to what is meant by the expression “slightly better.”

The balance factor B is defined as

$$B = \frac{S_{max}}{S_{opt}}$$

with S_{max} as the weight of the largest subdomain, and S_{opt} itself is set to

$$S_{opt} = \frac{G}{P}$$

where G is the total weight of nodes in the adjacency graph and P is the number of subdomains. A default imbalance tolerance of 3 % is accepted and allowed.

It should be noted that the results displayed over the terminal output cannot be influenced by the programmer or developer. Any inaccuracies result from internal computations by the *Jostle* domain decomposition algorithm and its implementation.

In addition to the parameters like balance or the maximum or minimum size of the domains, *Jostle* also gives information about the average runtime in seconds (not millisecs nor cpu or clock cycles) as well as the average memory that was used during the computation of the subdomains. The manual indicates that the memory usage is that on the processor from which the call is made and not a global maximum or sum of usage. Unfortunately, no more background information to these two parameters is given.

The following lines of C code are a small extraction of the wrapper code where the *Jostle* function call is performed and additional information is stored in dependence of the commandline switches:

```
int calling_jostle(void) {

  if ( verbose_mode || debug_mode || (DEBUG && output) ) {
       fprintf(stdout, "\nNow calling jostle ...\n");
       fflush(stdout);
  } /* Ende des Debug-ifs */

  if ( statistic ) {
    /* in case user wants the output of jostle being stored you have
       to redirect the output of jostle. */

    if ( redirect_stdout() == 0 ) {

      /* ups, something went wrong when redirecting stdout to file */

      fprintf(stdout, "\nERROR: Redirection of STDOUT to statistic \
                               file not successful!\n");
      fprintf(stdout, "===> Printing to STDOUT as usual ...\n");
      fflush(stdout);

    } else {

      fprintf(stdout, "\n");
      fprintf(stdout, "JOSTLE DOMAIN DECOMPOSITION RESULTS:\n");
      fprintf(stdout, "====================================\n\n\n");
      fflush(stdout);

    } /* end of redirect_stdout-if */
  } /* end of statistic-if */

  jostle_env("check=on");

  jostle(&total_of_elements, &jostle_offset, neighbour_amount,
                 jostle_element_weights, jostle_partition,
```

```
                &total_of_neighbours, jostle_neighbour_listing,
                jostle_neighbour_weights, jostle_machine_characteristics,
                jostle_subdomain_weights, &jostle_output_level,
                &jostle_dimension, jostle_coords);

  if ( verbose_mode || debug_mode || (DEBUG && output) ) {
       fprintf(stdout, "\nStoring jostle statistics ...\n\n");
       fflush(stdout);
  } /* Ende des Debug-ifs */

  /* now check the results of the statistics functions */

  jostle_cut_edges_weight = jostle_cut();
  jostle_balance_ratio = jostle_bal();
            /* The balance expressed as the ratio of the maximum
               subdomain weight over the optimal subdomain weight.
               A return value of 1.00 is perfect balance! */
  jostle_run_time = jostle_tim();
  jostle_memory_usage = jostle_mem(); /* returns the memory used by
                                         jostle in bytes */

  if ( statistic || verbose_mode || debug_mode || (DEBUG && output) )
  {
    fprintf(stdout, "Jostle cut edges weight = %d\n",
              jostle_cut_edges_weight);
    fprintf(stdout, "Jostle balance ratio = %f\n",
              jostle_balance_ratio);
    fprintf(stdout, "Jostle run time = %f\n", jostle_run_time);
    fprintf(stdout, "Jostle memory usage (in bytes) = %d\n",
              jostle_memory_usage);
    fflush(stdout);
  } /* Ende des Debug-ifs */

  if ( statistic ) {

    /* now finish redirection of stdout */
    restore_stdout();

  } /* end of statistic-if */

  return 1;

} /* end of calling_jostle-function */
```

6.12.2 Statistic Information Offered by Metis

When examining *Metis* in regard to functionality of statistics, one can search in vain. There are no special statistical parameters or similar information that one can retrieve with *Metis* or its function call. In case such values are of importance, the programmer or developer has to provide the information him- or herself by implementing auxiliary and temporary structures and variables during the run of the program.

Even when studying the manuals or documentation available to *Metis*, only a short remark on this topic can be discovered in a tiny little footnote.

Metis operates on a *k-way partitioning algorithm*, and the balancing factor is defined here as

$$B = \frac{km}{n}$$

where k is a k-way partition of an adjacency graph with n vertices, and m is the size of the largest part which was produced by this k-way partitioning algorithm. In short, B is essentially the load imbalance which is induced by non-equal partitions. A certain imbalance factor is also tolerated, as it was in *Jostle*. This factor normally ranges between 1 % and 3 %.

To be able to finally compute B for a certain domain decomposition performed by *Metis*, somehow one has to provide the information about the biggest subdomain. As this is not handed over from the *Metis* routines, one has to implement such variables or auxiliary data structures him- or herself. The remaining necessary parameters are given either through the commandline options, like the number of partitions that has to be produced or can be obtained from the input data, like the number of nodes in the adjacency graph of the model to be computed. With this "customized information" the balancing factor for a certain *Metis* partitioning algorithm on a special data set can be calculated.

In case the user has specified the `--statistic` switch (as explained in more detail in subsection 6.15.10 further on in this chapter, on page 195) something similar to the following examplary output will be displayed:

```
RESULTS OF THE TIMING ROUTINES AND STATISTICS:
==============================================
o Filename of input file: testdaten_11knoten_line_01.rfi
o Total of nodes: 11
o Total of elements: 10
o Total of domains: 3
o Usage of weights: OFF

METIS STATISTICS:
-----------------
o Size of biggest partition in elements: 4
o Size of smallest partition in elements: 3
o Number of cut edges by metis: 2
o Balance factor of metis: 1.200000
```

As there are no statistic parameters that can be obtained by the *Metis* procedure automatically, after the domain decomposition and because of the fact that extra data structures therefore have to be implemented and additional C commands executed, setting the `--statistic` switch directly means an increase in runtime due to the additional work that is now necessary for the determination of such statistical values.

The following is an extract of the C code of the wrapper that deals with the statistic parameters when having chosen *Metis* as the domain decomposition tool:

```
int metis_statistics(void) {

  int i;

  if ( debug_mode || (DEBUG && !output) ) {
       fprintf(stdout, "\nPreparing metis statistics ...\n");
       fflush(stdout);
```

```
  } /* Ende des Debug-ifs */

  /* find biggest and smallest partition of metis result */
  for (i = 0; i < number_of_partitions; i++) {

    if ( domain_elements_amount[i] > metis_max_domain ) {
         metis_max_domain = domain_elements_amount[i];
    } /* end of domain_elements_amount-if */

    if ( domain_elements_amount[i] < metis_min_domain ) {
         metis_min_domain = domain_elements_amount[i];
    } /* end of domain_elements_amount-if */

    if ( debug_mode || (DEBUG && !output) ) {
         fprintf(stdout, "\nActual smallest metis domain = %d\n",
                  metis_min_domain);
         fprintf(stdout, "Actual biggest metis domain = %d\n",
                  metis_max_domain);
         fflush(stdout);
    } /* Ende des Debug-ifs */

  } /* end of i-for-loop */

  metis_balance = ((double)(number_of_partitions * metis_max_domain))
                        / ((double) total_of_elements);
  return 1;

} /* end of metis_statistics-function */
```

In chapter 7, starting on page 201, a detailed comparison and discussion of *Metis* and *Jostle* will be provided.

It should be noted, however, that for a normal preprocessing the statistics are not obligatory. Just in case the user is interested in background information or would like to do a more detailed analysis in regard to the success and quality of the domain decomposition provided by the two different tools, such statistic information might be of help.

6.13 Testing and Debugging

Wo sind wichtige Konsistenztests wichtig bzw. ratsam? Was ist beim Debugging zu beachten? Wie ist die Speicherkonsistenz bzw. Memory leaks? Wie sauber wurde programmiert? Oder gibt es quick and dirty hacks im Programm?

6.14 Timer Functions

The implementation of a set of timer functions originated during the coding, debugging and testing phase from the author's curiosity as to where most of the runtime would be consumed during the execution of the wrapper code. The crucial factor for this idea was the domain decomposition tool *Jostle* (see also section 5.4 in chapter 5, on page 107), where a special variable can be retrieved that gives the runtime of *Jostle* in seconds.

It quickly turned out that a unit of seconds is too long a time period to give precise and exact information. Fortunately, on Unix- or Linux-based systems a function called `gettimeofday` can be used in C. When calling it, the time in seconds and in microseconds that has elapsed since January 1st, 1970, is returned and stored by this function in a predefined data structure called `timeval`. To use `gettimeofday` the header file `time.h` that can be found in `sys`, has to be included.

When starting the preprocessing the starting time is measured. During the execution of the code the time is then stopped once in a while at specific interesting locations within the program and finally, upon program termination, the endtime is taken again. The different values are then converted to milliseconds, as microseconds turned out to be too short and the resulting time values did not work well in regard to handling and reading by a human being.

The following C statements show a little excerpt of the code that deals with the time taking:

```
int do_time_check( int time_counter) {

  if ( time_counter < TIME_CHECKS ) {

    gettimeofday(&starttime, NULL);

    starttime_millisecs[time_counter] = (double) starttime.tv_sec *
            1000.0 + (double) starttime.tv_usec/1000.0;
    time_counter++;

  } else {

[...]
```

Depending on the different commandline options that can be specified by the user (see also the upcoming section on this topic), the measured times will be stored into a separate statistic output file or given on the display. A typical result of the timing functions looks as follows:

```
(The following timing results are all given in millisecs!)

o Time between program start and processing commandline parameters: 0.264893
o Time for opening all files and reading in input file: 0.167969
o Time for processing input data and build adjacency graph: 0.0539551
o Time for preparing the data structures/initialization of domain
           decomposition tool: 0.0100098
o Time for calling domain decomposition tool "Metis": 0.675049
o Time for processing data of domain decomposition tool: 0.13208
o Time for printing output file: 23.012

==> Time difference from start of program to the end: 24.3159
```

The time difference that is given at the end of the timing output is more or less the runtime of the complete program. It is also clear that in case lengthy testing routines have to be performed when the corresponding commandline switch was given by the user, the runtime will increase.

In Appendix B, starting on page 241, more examplary output results will be discussed and given.

6.15 The Commandline Interface of the Wrapper Program

Up to this point everything which was explained in more or less detail in this implementation chapter is normally hidden from the user. The only chance for the user to see the implementational work from the "outside" is, e.g., by an improved runtime or uncomplicated handling when using the software, etc. But generally all these important facts are completely invisible and the user can only "trust" the developer and programmer to have spared no effort to succeed with the best possible piece of code.

This second-to-last section of the implementational chapter therefore assumes the point of view of the user, and the wrapper software is now presented from exactly this "outside." The following subsections explain now in more detail the different options that are available when using the wrapper.

At the current stage of the project the wrapper program *domdec* can be used over a terminal or an xterm on a UNIX or Linux platform. Assuming the user is mainly interested in having his or her data be decomposed into a certain number of subdomains and is not concerned about the implementation behind all of this, the program call will be rather simple. The author has tried to reach a state of highest possible transparency as far as this is feasible, while offering maximum flexibility for those users who would like to adjust a few parameters.

The handling is very simple, especially for users accustomed to a UNIX or Linux environment. Most programs on such a platform have in common that there is always a *help switch* or an option indicating how to use a certain software. Such a *help option* is also available for the wrapper; an example of how to begin when never having used *domdec* before follows:

```
==> domdec_v03_2 -h
```

We note that ==> here is a synonym for the system prompt on the terminal where the program name and commandline options have to be typed in (similar to a DOS prompt on a normal desktop machine with something like \>). As can be seen above, the wrapper program is called *domdec*, which derives from *domain decomposition*. The current/actual version number follows; in this case it is version 3.2. The obligatory and optional switches for the wrapper are just appended consecutively in no special order, directly behind the program name. In our example there is just one single commandline option, which is "-h" for *help*.

Submitting the command `domdec_v03_2 -h` results in a text listing on the terminal of all possible and available switches as shown in the excerpt below (the full output of `domdec_v03_2 -h` and `domdec_v03_3 -h` can be seen in Appendix C at the end of this thesis):

```
==> domdec_v03_2 -h
==>

    COMMANDLINE OPTIONS OF DOMDEC_V03_2:
    ====================================

    -b, --debug     Switch to debug-mode and print debug messages.
                    Default is set to "debug = OFF"

    -d, --ddcfile   Expects a filename of a ddc file which will be used
                    as the later output file. If no ddc file is given
                    an output file will be created automatically, using
```

```
                    the input filename plus additional suffixes depen-
                    ding if commandline option "-l" was given. Maximum
                    filename length: 500 characters!

    -h, --help      Prints out this help text and exits, see correct
                    usage below!

    [...]

    USAGE OF DOMDEC_V03_2:
    ======================

    domdec_v03_2 [-v|--verbose] [(-p|--partitions) value] [-b|--debug]
                 [-j|--jostle|-m|--metis|-c|--parmetis] [-s|--statistic]
                 [(-o|--output) filename] [(-d|--ddcfile) filename]
                 [-t|--test] [-w|--weight|-n|--noweight] filename

    domdec_v03_2 [-b|--debug] [(-p|--partitions) value] [-v|--verbose]
                 [-j|--jostle|-m|--metis|-c|--parmetis] [-t|--test]
                 (-i|--input|-r|--rfifile) filename [-s|--statistic]
                 [-w|--weight|-n|--noweight] [(-d|--ddcfile) filename]
                 [(-o|--output) filename]

    domdec_v03_2 (-h|--help)

==>
```

For those users who are not familiar with a UNIX or Linux environment, it should be mentioned that appending a "`| less`" or "`| more`" behind the `domdec_v03_2 -h` results in an output on the terminal where only one terminal page at a time is displayed. The full command using this so-called *pipe* then looks as follows:

```
==> domdec_$v03_2 -h | less
```

It is beyond the scope of this text to provide a full introduction to UNIX or Linux. For those readers who are not used to working on such a platform, good online tutorials can be found in vast numbers on the web. One starting point could be [Linux] or [McCarty] when searching for a good book. Advanced readers who just need a quick start or a reference should have a look into [Welsh], [Gilly], [Siever] (English books) or [Kofler], [Gulbins] and [Zilm] (German books).

At the time of the writing of this thesis `domdec_v03_2` is the actual wrapper program version which has been thoroughly tested and is currently being used. Therefore all switches and options are discussed in regard to this version. `domdec_v03_3` is evolving from `domdec_v03_2` with some improvements and additional options. The core of the wrapper, however, is not going to be altered considerably in the new version. (Future plans in regard to the wrapper program and of `domdec_v03_3` are discussed in section 7.4 and following of chapter 7, starting on page 227.) Table 6.3 gives a brief listing of all implemented commandline options of `domdec_v03_2`:

Most switches and options are self-explanatory, at least for those users who are slightly familiar with a UNIX or Linux environment. As some of the commandline options are obligatory and others are just optional, each of them will be discussed below in more or less detail, depending on their importance. The order in which the parameters will be introduced is intentional. The most important ones are discussed first, as they will either contribute to optimal usage of `domdec_v03_2` or comprise the few obligatory options. The rather insignificant ones which do not really influence

Shortcut of Option/Switch	Long Option/Switch
-b	--debug
-d	--ddcfile
-h	--help
-i	--input
-j	--jostle
-m	--metis
-n	--noweight
-o	--output
-c	--parmetis
-p	--partitions
-r	--rfifile
-s	--statistic
-t	--test
-v	--verbose
-w	--weight

Table 6.3: Listing of all available options and switches of domdec_v03_2

the preprocessing itself but might just "tune" the output or result in additional informational output files will be introduced at the end.

6.15.1 The --help Commandline Option

As can be seen in Appendix C on page 253, the commandline option -help, or its short form -h, is the only one which can be evoked without any other switches:

```
==> domdec_v03_2 (-h|--help)
```

It should be mentioned that the expression with the round brackets in the example command above indicates that the user has to make a choice: Either he or she has to type in

```
==> domdec_v03_2 -h
```

or

```
==> domdec_v03_2 --help
```

If something appears in square brackets instead of round brackets, the usage of the expression within is not obligatory, and the user of `domdec_v03_2` can freely decide whether or not he or she is going to use the expression or parameter when calling `domdec_v03_2`. As can be seen from the examplary help text listing shown previously, most of the parameters that can be handed over when invoking the wrapper software are surrounded by square brackets rather than by round brackets. The reason for this was already indicated at the beginning of this section: The preprocessing should be completely transparent for the user and in case nothing special is needed or wanted by the user, most of the required variables and parameters have already been set to certain reasonable default values.

6.15.2 The `--input` Commandline Option

This switch selects a specific input file that holds the data set of the model to be computed. Actually, this switch is not obligatory. When examining the help excerpt of the example on page 189 and comparing the two long, different calls of `domdec_v03_2` at the end of this help text, one will detect just one small difference. In the first of the two long wrapper calls a filename is obligatory, as there is no square or round bracket surrounding it (at the end of the long call) but the `--input` switch is missing. In the second call there is no obligatory filename at the end of the call but instead the `--input` switch now appears in round brackets, which means it has to be specifically set and is obligatory for the call of the program in this version.

The reason for this difference is quickly explained. Use of the first of the two calls presupposes that an input file has to be given so that the preprocessing can be carried out. No input file means no preprocessing. Therefore a simple wrapper call like

```
==> domdec_v03_2 examplary_inputfile.rfi
```

would be completely sufficient. The second type of call was just introduced for reasons of consistency. Here the user can specify the input file by explicitly setting the `-i` or `--input` switch manually. Using this type of wrapper call, the input file need not necessarily be given at the end of the call, as it was in the first example. Use of the second version in calling the wrapper program could look like this:

```
==> domdec_v03_2 -i examplary_inputfile.rfi
```

or, when using the long option, like this:

```
==> domdec_v03_2 --input examplary_inputfile.rfi
```

It should be noted that for the sake of readability the maximum filename length is reduced to 255 characters. Generally the maximum filename lengths on Unix or Linux systems is longer, in most cases 4096 characters. But normally, filenames this long are extremely rare. As the filename is typed in by hand by the user, such long filenames are impractical. Therefore the system-wide default value for filenames in general is overwritten and reduced to a maximum of 255 characters. (In fact, it is set to 256 characters, but a special character, invisible to the user, is placed at the end of the filename string so that the underlying system can identify the end of the given input filename that the user has typed in.)

6.15.3 The `--partitions` Commandline Option

Other than with the `-i` or `--input` switch the usage of the `-p` or `--partitions` switch is not directly required. If no `--partitions` switch is given, a default value of 12 is automatically set from the wrapper software. There was no explicit reason to take the number 12. It resulted from various preprocessings during testing and productive runs later on. A smaller value is hardly reasonable, especially when using big input sets. A much larger value might also be unreasonable, as the overhead in reassembling the result vector in the course of the main model-computation phase during the modelling process increases with every additional domain that has to be calculated. More domains mean more nodes that are shared by two or more domains in the adjacency graph. And more sharing results directly in more communication overhead. (See also sections 4.4 of chapter 4 on page 79 and section 5.1 of chapter 5 on page 89.)

When explicitly setting the `--partitions` switch, the switch keyword has to be followed by an integer indicating the number of subdomains to be created when calling the wrapper program. An example program call when a user wants 8 partitions instead of the default value of 12 could look as follows:

```
==> domdec_v03_2 -p 8 examplary_inputfile.rfi
```

or, similarily, using an explicit `--input` switch as introduced above,

```
==> domdec_v03_2 -i examplary_inputfile.rfi --partitions 8
```

As indicated by the notation in the previous example help excerpt, the `-p` switch and the `--partitions` switch can be used interchangeably.

6.15.4 The `--jostle` Commandline Option

The usage of the `-j` or `--jostle` switch is a mixture of that described for the `-p` or `--partitions` switch and the `-i` or `--input` switch. If no switch is given at all, Jostle is taken as the default partitioning tool for the preprocessing. Therefore, in case someone chooses Jostle as his or her favorite tool for the subdivision of the model data, the following example call of the wrapper is completely sufficient:

```
==> domdec_v03_2 examplary_inputfile.rfi
```

Setting the partitioning tool explicitly to Jostle by using either `-j` as the short version or `--jostle` with the long parameter name is not necessary at all. The switches have just been implemented for reasons of consistency.

6.15.5 The `--metis` Commandline Option

In contrast to the `--jostle` switch the commandline option `-m-` or `--metis` is important. It must be explicitly set when the user wants to override the default value of the domain decomposition tool that is generally set to Jostle. Only by specifying this switch when calling the wrapper, can Metis be chosen as the favorite partitioning tool. An example call could look like:

```
==> domdec_v03_2 --metis examplary_inputfile.rfi
```

It should be noted that Jostle is hardcoded to be the default tool for the subdivision of the model. Using Metis once instead of Jostle does not switch this default value. The value for the partitioning tool is not a typical "toggle parameter."

6.15.6 The `--parmetis` Commandline Option

The usage of the `-c` or `--parmetis` switch is similar to that of Metis, theoretically. Practically, this commandline parameter has not yet been implemented in the wrapper software. It is one of the future improvements that are planned: See also section 7.5.1 of chapter 7 on page 228.

6.15.7 The `--ddcfile` Commandline Option

The `-d` or `--ddcfile` switch is again one of those commandline parameters that need not necessarily be specified by the user. If no value is given, the output file which results from the preprocessing is created automatically from the filename of the inputfile by taking the input filename without the suffix `rfi` and appending date and time plus the suffix `ddc`.

By way of example: Let us assume that the input data set is stored in a file called `examplefile.rfi`. We further accept all default parameters for the wrapper, which means Jostle is taken as the domain decomposition tool and the program run was invoked on June 5th at 08:15 am. When now starting the preprocessing with a call like

```
==> domdec_v03_2 examplefile.rfi
```

the resulting output file which holds a listing of all the different partitions and their corresponding elements (see also Appendix B on page 241) would be named `examplefile_05June2009_0815.ddc`.

In case the user of the wrapper would like to specify his or her own name for the resulting output file for the partitions, he or she can then hand over this filename by using the `-d` or `--ddcfile` switch to the wrapper software. This way a call of the preprocessing software could also look as follows:

```
==> domdec_v03_2 examplefile.rfi --output my_outputfile
```

As this example clearly shows, it is not necessary to specify the suffix `ddc` on Unix or Linux platforms as is generally required on Windows machines. One should keep in mind, however, that the same restrictions on the length of an output filename apply as those mentioned previously for the input filename when introducing the `--input` switch.

6.15.8 The `--rfifile` Commandline Option

The `-r` or `--rfi` switch is nothing other than an alias for the `-i` or `--input` switch which was just implemented for consistency reasons. When examining the second call in the excerpt of the help text above, one can see that the user has to either specify one of the four switches `-i`, `-r`, `--input` or `--rfi`, as they are set in round brackets, or simply use the first type of call to start the wrapper.

6.15.9 The `--output` Commandline Option

This commandline option allows the user to produce an extra output file in addition to the standard `ddc` file. When the `-o` or `--output` switch is given, a simple ASCII text file is written after having called the domain decomposition tools during the run of the wrapper which holds a simple table that shows each element of the input data set and in which subdomain it was placed.

Normally this special file is not required, and the switch is just a byproduct which resulted from extensive testing and debugging. It might be of interest for the user though, to see a direct mapping between the elements and the domains, as the standard `ddc` output file is sorted according to the resulting partitions rather than to the single elements.

It should be noted that the user should not confuse the `-o` or `--output` switch with the `-d` or `--ddcfile` switch. The latter is the important file which is needed later on in the main processing stage when computing the model. (See also subsection 6.15.7 above.)

6.15.10 The `--statistic` Commandline Option

Both domain decomposition tools, Jostle as well as Metis, offer additional functionality for the computation of certain statistics. This results directly from the subdivision of the input data sets and can be saved or further processed after the return from the call of the domain decomposition within the wrapper software.

In case the user is interested, he or she can use these statistics to analyze the quality of the domain decomposition. Originally, the statistic output file, which is created when explicitly setting this switch, was written during the debugging and testing phase to examine which of the two partitioning tools would be most suitable.

It turned out, though, that in most cases the differences in quality basically did not exist, no matter what input files had been used, large or small, how many partitions had been produced, etc. (For this, the reader can have a look at chapter 7, starting on page 201. In section 7.1, there is a detailed discussion of this topic.)

The `-s` or `--statistic` switch is optional and its default value is set to "OFF." The user has to set it intentionally. When switched to "ON" by giving this commandline option, an additional output file is created during the preprocessing run of the wrapper. This file is assigned the same name as the standard output file (the `ddc` file); just a different suffix is appended. Instead of `ddc`, the statistic output file ends on the file name extension `stt`.

The usage of this switch is simple. By typing, for example,

```
==> domdec_v03_2 -s examplefile.rfi
```

the statistic output is activated and a statistic file with the name `examplefile_date_time.sst` is created. Accordingly, if no `-s` or `--statistic` option is typed in on the commandline, no statistic file is written. (Some example statistic output files can be examined in more detail in Appendix B, part 2, starting on page 247; more on the basics of the available statistic data can be found in the previous section 6.12 of this chapter, on page 182.)

6.15.11 The `--test` Commandline Option

The `-t` or `--test` switch has the same usage as the statistic switch. It is also a typical "ON/OFF" option. If one of the two switches is specified on the commandline by the user, an additional test of correctness of the output that results from the call of the domain decomposition tools is performed.

This switch is a byproduct of the extensive debugging and testing phase, as sometimes the output produced by Metis and Jostle seemed to be faulty. Later on it turned out that the data sets showed inconsistencies which could be, for example, single, non-integrated nodes or elements of the model. At first, the author assumed that the input data sets had already been tested for correctness and that such cases had already been eliminated. Unfortunately the input files had not been checked by test routines. Through thorough debugging and searching for strange results in the output of the preprocessing, the inconsistencies which also resulted in faulty domain decomposition could finally be found. After having implemented this additional testing routine, such errors could be eliminated later on.

It turned out in the further parallelization process that this testing routine, to be called in case the `-t` or `--test` switch is set, was so effective that often the persons responsible for the modelling and creation of the input data sets used the preprocessing just to activate this testing routine to check their files for consistency and correctness.

It should be noted, however, that helpful as this additional function might be, it is also very time consuming, as the complete model is read and processed again within this testing procedure. This is the only way to find the faulty model locations. The total runtime of the wrapper program increases with the complexity of the original input data sets and their pure element or node number.

As the `-t` or `--test` switch is used in the same way as the `--statistical` option, an example call of the wrapper could look as follows when the test function should be activated:

```
==> domdec_v03_2 -t examplefile.rfi
```

After termination of the run of the wrapper, the user will see a short or long commandline notification indicating whether the testing of the subdivided model was succesful or not. The length of the commandline output depends on the activation of further switches that are explained in more detail in the following subsections. An example of such an output, where just the `-t` or `--test` switch is set without any other options, would be:

```
Starting the verification of the output results ...
==> Counter-check of ddc output successful!
```

6.15.12 The `--verbose` Commandline Option

When using the `-v` or `--verbose` switch, the wrapper program becomes a bit more "talkative" during its run. It has the same function as the `-v` or `--verbose` options on Unix or Linux environments. The default value is "OFF." The user has to intentionally set this specific commandline parameter to produce a commandline output indicating the current stage of the preprocessing. The following excerpt is the beginning of a real wrapper run using this command to call it:

```
==> domdec_v03_2 -v testdaten_72knoten_tri_03.rfi
```

This invoking of the wrapper will start the following commandline output on the terminal or console window:

```
START OF PROGRAM EXECUTION ...

Verbose Mode ==> ON
Debug Mode ==> OFF

All input for domain decomposition is treated the same!
==> no usage of different weights!

Validation of domain decomposition tool ...

No domain decomposition tool was chosen!
Take Jostle as default!
"Jostle" was chosen as domain decomposition tool.

These many partitions shall be created: 12

Validation of input file ...
Checking if an input filename was given ...

Validation of output file ...
No valid output file was found! Going to create one ...

==> Now creating a valid output filename ...

Assembly of new output filename complete!
Original input filename that was used:
                 testdaten_72knoten_tri_03.rfi
Resulting new output filename:
                 testdaten_72knoten_tri_03_23Jan2009_1210.ddc

[...]
```

This commandline option resulted from extensive testing and debugging during the development and programming phase of the wrapper program.

6.15.13 The `--debug` Commandline Option

The `-b` or `--debug` switch is merely a byproduct – but a very important one – of the extensive debugging and testing phase during the implementation of the wrapper program. When explicitly

set by the user, the wrapper program becomes even more “communicative” than it would be if the `-v` or `--verbose` commandline parameter were given. The resulting commandline output gives explicit information as to which stage or in which part of the program the process is at the very moment.

It is possible to switch single parts within the wrapper program on and off so that during debugging and testing only within the faulty code fragment which might crash or which needs improvement more commandline output is printed rather than the whole lengthy, time-consuming output for the complete preprocessing run.

Under normal circumstances this special switch is not of any interest to the user and one should avoid using it during a normal production wrapper run. It is only helpful during the coding, testing and debugging phase. Should the reader be interested in such a lengthy output, a wrapper call like

```
==> domdec_v03_2 -b testdaten_72knoten_tri_03.rfi
```

would result in a commandline output like this:

```
START OF PROGRAM EXECUTION ...


End of processing input data and options ...

All input for domain decomposition is treated the same!
==> no usage of different weights!

Validation of domain decomposition tool ...

No domain decomposition tool was chosen!
Take Jostle as default!
"Jostle" was chosen as domain decomposition tool.

These many partitions shall be created: 12

Validation of input file ...

Checking for correct usage of commandline parameters ...
Checking if an input filename was given ...

No input filename found ...

... but additional parameters on commandline found!

Saving given commandline input as input filename ...
        ==> "testdaten_11knoten_line_01.rfi"

Validation of output file ...

[...]
```

6.15.14 The `--weight` and `--noweight` Commandline Option

The the `-w` or `--weight` switch and the `-n` or `--noweight` switch, respectively, can be used interchangeably. Weights are not yet used in the current version of the wrapper. They are one of

the improvements that are planned for a future version. Chapter 7 deals in more detail with such ideas and the interested reader can also have a look at section 7.5 and subsection 7.5.2, respectively, starting on page 228.

6.15.15 General Usage of the Wrapper Program

After the previous detailed discussion about the different commandline options that can be specified or left out by then using a default value, this short subsection gives a few simple examples of how the wrapper could now finally be used in a real production run. With the above information about the switches, the usage should be straightforward and not impose any problems.

As a first example, we assume that the user is interested in a domain decomposition with 8 different subdomains, the default value of Jostle as the domain decomposition tool is correct and the output file is to be created in accordance with the input data set. In such a case the simple call of the wrapper would look like follows:

```
==> domdec_v03_2 examplary_inputfile.rfi
```

Given the same prerequisites as above but now using Metis instead of Jostle and desiring a total of 12 subdomains, the call of the wrapper has to be modified as shown in the this call:

```
==> domdec_v03_2 -m -p 12 examplary_inputfile.rfi
```

In the following example, the user is not sure about the consistency of the input data and would also have an overview of the "quality" of the subdivision of the original model. Therefore he or she specifies the switch for the extra test routine that operates upon the output of the domain decomposition tool and additionally sets the switch for the statistic output that the `stt` file is being written. Still, Metis is chosen as domain decomposition tool to produce 12 subdomains of the original model:

```
==> domdec_v03_2 -m -p 12 -s -t examplary_inputfile.rfi
```

The options can be given in any order. It is also completely unimportant for the result whether the commandline switches are given in their long form using the double hyphen written as `--statistic`, or the short version with just one hyphen and the necessary option character, like `-s`. Therefore, the next three example calls are all identical to the above example:

```
==> domdec_v03_2 -m --partitions 12 -s --test examplary_inputfile.rfi
==> domdec_v03_2 --metis -p 12 --statistic -t examplary_inputfile.rfi
==> domdec_v03_2 -m -p 12 -s -t --input examplary_inputfile.rfi
```

6.16 Independence of the Wrapper

Already in the very first stages of the parallelization process and at the beginning of the thesis it was evident that there would be a transition from 32-bit architectures to 64-bit architectures on the computer market in the future. This shift in architectures will not happen at once; rather, it is a slow process. The current 32-bit machines will be increasingly replaced in time by the more powerful and faster 64-bit architectures.

Not all of the official 64-bit machines indeed offer the full extent of a true 64-bit architecture with 64 separate data or address links. Some of these transitional systems still operate on a regular 32-bit architecture but pretend to be a full-fledged 64-bit architecture by using special hardware or additional operating system software. (A full discussion of these computer systems is beyond the scope of this thesis. The interested reader will find more on this topic in one of the hardware "bibles," like [Patterson]).

As a lot of the very powerful and fast modern parallel computers or vector machines already operate on a true 64-bit hardware, it was clear to the author that the preprocessing software should not only run on the "old" platforms with the 32 bits alone. The wrapper should rather be architecture independent as far as possible so that the code could be compiled and executed on either one of the two architecture types, although this was not required for the parallelization process itself. Thus, during the entire process of development and later implementation and testing, it would not be important on which machine the code was actually run.

The little bit of overhead required for this independence of the underlying platform was not problematic. Rather quickly it became obvious that this extra overhead was worth the effort, as through the process no restrictions due to a specific machine architecture were encountered.

The only thing that had to be taken care of was the installation and implementation of the domain decomposition tools, Jostle and Metis, for each of the architectures, as they are called from within the wrapper software. This was not a major obstacle, as they just had to be newly compiled and installed for the machine on which the wrapper was to be executed.

The author has tried to keep the preprocessing software as machine or architecture independent as possible. The way it is coded should even permit it to compile and run on future platforms with even bigger word sizes or bus widths. The only basic requirement for achieving this independence is the existence of the domain decomposition tools on these platforms as well as a C compiler that works according to the C standard.

As the general computing environment of the big "number crunchers" and "work horses" in HPC is a Unix or Linux environment or derivatives of it, the preprocessing software was developed mainly for these kinds of operating systems. It should also be possible to port the wrapper to a DOS or Windows environment, but as it is rather unusual to have an enormous overhead using "fancy graphical user interfaces" on terminal-based HPC computers, the code would work only on something similar to an Xterm under Windows or DOS. The wrapper does not offer any graphical- or window-based input elements or a similar features.

Unfortunately the programming environments that are commonly available under a Windows or similar operating system prove to not normally conform to the C standard, and often slight variations in regard to the compiler exist. On the Unix- and Linux-based systems the wrapper code could be compiled without remarkable problems using a simple GNU C compiler like the `gcc`, which is built into the normal Unix/Linux environment.

Chapter 7

Discussion of the Results, Outlook and Future Improvements

Apart from the core part of the thesis in the form of the implementational chapter 6, starting on page 119, where the engineering and programming work as well as the challenges experienced have been described, this last chapter is also very important. Here, everything introduced and discussed in the previous chapters now finally comes together and with this thorough foundation, the facts and the outcome can be analyzed and examined. Not only will the domain decomposition tools and their results be compared but also the different possible data structures with their benefits and drawbacks.

It is just natural that trade-offs have to be made that during such a big coding and implementation project. Not only may new and interesting ideas arise, but unfortunate decisions may become evident. These developments will be investigated in this last part of the thesis and potential and probable further improvements or changes and even extensions will also be introduced and discussed.

As the domain decomposition tools are one major part of the preprocessing process, this chapter begins by examining them and comparing the results that could be obtained and analyzed with respect to one another. The other major focus is a detailed discussion and differentiation of the various data structures that one could possibly implement into the wrapper program. They all show strengths as well as weaknesses, as the "perfect, unproblematic" data structure does not exist. Which one is the most suitable and where are its disadvantages outweighed by other very important aspects? Is there a data structure that one could ultimately regard as ideal in the given situation or do all of them show more or less the same applicability? Which one is to be finally selected and which one no longer important?

The chapter will be concluded by a short outlook upon ideas that arose during the implementation phase; possible extensions will be introduced and discussed in regard to future improvements.

7.1 *Metis* or *Jostle*? - A Detailed Comparison between the Two Domain Decomposition Tools

In the very first stage of this thesis the problem of domain decomposition soon became evident. After an analysis of the already existing tools and a consideration of developing and implementing

one's own domain decomposition tool, it quickly became clear that coming up with something original was simply neither feasible nor wise. *Jostle* (see section 5.4 in chapter 5, starting on page 107) and *Metis* (see section 5.5 in chapter 5, starting on page 112) seemed to be two promising candidates for domain decomposition tools during the preprocessing stage.

At first sight they show many similarities, but on second sight, there are minor but also major differences between these two packages. This is already obvious when examining the history and the pure facts behind these two tools. *Jostle* was initiated almost as a "one-man project" in 1995 by Chris Warshaw, who is still today the man behind this package. He signs responsible for everything connected to *Jostle*. This has the advantage that there is continuity within the whole process, but on the other hand, the disadvantage is apparent: Being the only one responsible for maintaining further development of *Jostle* while holding a full-time job slows things down enormously. The author herself experienced this problem after asking for a free but required license to use and work with *Jostle* during this project.

When, after more than half a year no answers had come from Chris Warshaw to the author's numerous emails and faxes expressing an interest in this software package and the need to work with it, it became questionable if *Jostle* would be an appropriate future domain decomposition tool.

It finally turned out that the HLRS had it installed and already registered for research purposes; working there meant having full access to it. The development of *Jostle* continued over the following years, but then, for whatever reasons, seemingly nothing much happened at all. Now, finally, *Jostle* was recently commercialized and is available under the name of *NetWorks*, still maintained mainly by Chris Warshaw. The last official version dates back to 2005. Nevertheless, it is still one of the most popular domain decomposition tools in the world and more than 150 different licensed sites, some of them well-known facilities like NASA, for example, are using it world-wide.

Jostle is reputed to be a stable and solid piece of software and can be installed and deployed on various different platforms in the Unix and Linux world. This can only be underlined by the author, who never encountered any problems in working with *Jostle* whatsover, independent of the underlying computing environment.

The documentation is good enough to familiarize oneself with the package, although some basic background in the topic of domain decomposition itself is required. Still, the manuals and documentation provided show a tendency towards more professionalism than those offered by some other software packages.

Comparing *Metis* to *Jostle*, things look a little bit different. *Metis* started out as a typical university project with a handful of interested and talented researchers. One major contributor was and still is George Karypis. Another major difference from the start is the fact that *Metis* is not one big "moloch" for domain decomposition but rather a collection of smaller functions and routines for domain decomposition. The problem for the user of this software package is to establish which of the many procedures and features is best suited to a given problem.

The existing documentation and manual were not very helpful. It is obvious that the creators of *Metis* began with many ideas initially. At some point, however, as is often the case, documentation falls short. The existing documentary material is sufficient for someone who is already experienced and knows what to look for or has someone at hand for assistance. But for the unexperienced "newbie" to this field, quite some detailed information as well as explanatory text is missing.

As *Metis* was never a "one-man project" like *Jostle*, it now confronted a totally different problem: It almost died at some point and then got revived again, as so often happens in a university envi-

ronment, where the survival or death of certain projects depends on financing. Between version 4.0 and 5.0 of Metis, almost a decade passed during which no visible progress was made.

The *Metis* homepage looks rather professional in comparison to the *Jostle* homepage although *Metis* still is a free software package requiring no licensing at all. The current versions of this package date back to the middle of 2007. *Metis* even offers full 64-bit architecture support in the meantime, unlike the current version of *Jostle*.

Both packages seemingly still rely on the same basic algorithms and basics in regard to the domain decomposition on which they have been founded since their very beginnings.

During the work with *Metis* no direct differences in handling compared to *Jostle* could be experienced. Since there is no painstaking licensing procedure, one could freely work with *Metis* from the very moment it was installed on the machine. The only obvious difference in daily use when considered from the outside was the quality of documentation, which was clearly much better and more helpful when working with *Jostle* than with *Metis*.

When gathering opinions on which of the packages could be recommended, there was neither a consensus on choosing one over the other nor on steering clear of one of the two. Both packages are quite well known and were and are still widely used nowadays. At the time of their development, there were not many other alternatives but in the meantime, HPC is no longer an "exotic field" to work in and many other tools have appeared and vanished again. *Metis* and *Jostle* have survived and are still continously being used. One reason for this definitely is the reputation that they have gained throughout the years. Everybody has at least heard of them and or is familiar with both, and the majority of people who need to care about domain decomposition issues has already worked with one of them, if not even both.

As there was no clear "yes" or "no" as to the choice of tools, the author simply decided at some point to implement both of them. Gaining familiarity and experience with both of the tools and the basics behind them and the domain decomposition was required, no matter which one would be chosen in the end. The additional extra work involved in coding and building in the second domain decomposition tool could be justified by arguing that this was the only way to determine which tool was most suitable.

The software-related handling is very similar with both tools, as already explained in quite some detail in chapter 5. Both rely on the so-called *Chaco Input File Format* discussed in length in subsection 5.4.1, starting on page 107. The notable difference in the amount and also quality of the documentation for the two tools still made it quite difficult for an unexperienced user to decide which of the many functions and procedures contained in the *Metis* package should now be chosen to start the whole process. Here the coding process with *Jostle* was much easier and more straightforward; there was simply not so much choice.

For this reason and invisible to the user, two different calls to *Metis* are now implemented into the wrapper code. According to the documentation there is a certain number of domains on which a given *Metis* function always works better than another. The magical limit is 8 domains, as already explained in section 5.5, starting on page 112.

When testing various smaller, artificial input test files with both tools, the results achieved hardly differed in regard to parameters like biggest or smallest subdomain size or balancing factor (with respect to the underlying algorithm upon which the tool is working, *Metis*: $B = k \cdot m/n$, *Jostle*: $B = S_{max}/S_{opt}$), see also chapter 6 and especially section 6.12 here about the statistic functions, starting on page 182. The detected difference often lay in the numbering of the subdomains and

their location in the model or in the assignment of the various elements of the input model to a certain partition. Some examples of such output files and the results of the two tools can be seen in Appendix B, starting on page 241.

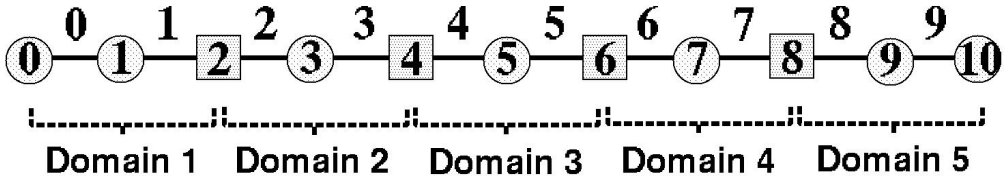


Figure 7.1: Domain decomposition of a 1D model consisting of 11 nodes and 10 linear elements

Fig. 7.1 shows a very simple 1D input test file which consists of 11 nodes (the dots and squares, respectively, which are numbered from 0 to 10) and 10 linear elements (numbered from 0 to 9 above the line). This model should be subdivided into 5 partitions in the given example. The result was completely identical, no matter which of the two domain decomposition tools had been chosen. Both correctly subdivided the input model into 5 equally sized "chunks" consisting of 3 nodes each and 2 elements; only the numbering of the subdomains varied. The boundary nodes, i.e., nodes shared by two different partitions as they lie right on the border between two subdomains, were placed identically. (The boundary nodes are the squares in the figure. For more information about such nodes and the problems behind them, the reader can have a look at chapter 5, section 5.1 and figures 5.8 and 5.9, starting on page 89.)

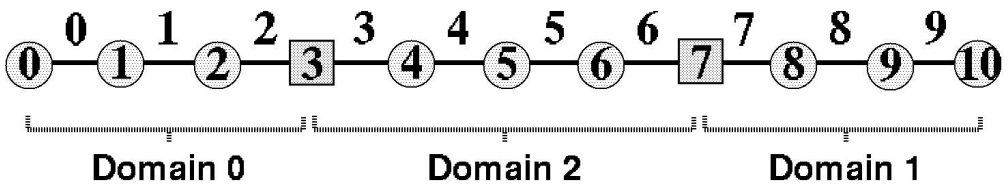


Figure 7.2: Domain decomposition by *Jostle* of a 1D model consisting of 11 nodes and 10 linear elements

Staying with this example but reducing the number of subdomains from 5 to 3, where 10 elements can no longer be shared equally, showed a little bit of variation and a small difference, as can be seen in figures 7.2 and 7.3.

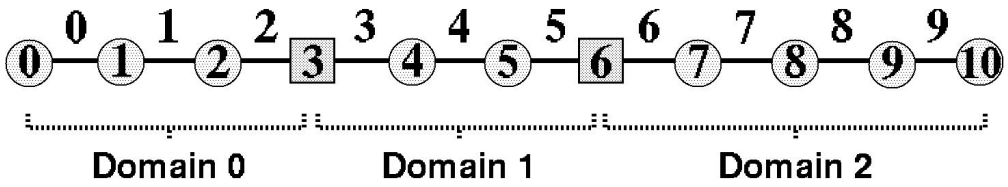


Figure 7.3: Domain decomposition by *Metis* of a 1D model consisting of 11 nodes and 10 linear elements

Both tools resulted in an equal distribution of the elements with two times 3 elements and one time 4 elements. Both of them made the last partition, domain no. 2, the biggest one but *Jostle*

located it at the very right edge of the model, whereas *Metis* placed the biggest domain right in the middle. This leads to a difference in boundary nodes. Both still have just 2 boundary nodes (again indicated by the squares) but in the model that was subdivided by *Jostle*, node 7 became a boundary node whereas when using *Metis*, node no. 6 was shared by two partitions.

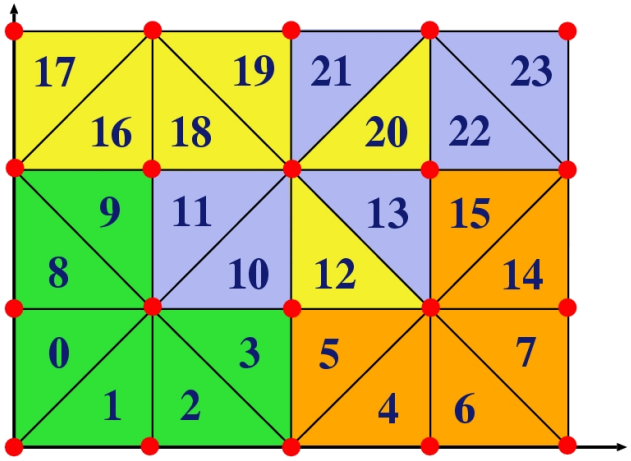


Figure 7.4: Domain decomposition into 4 subdomains by *Jostle* of a model with 20 nodes and 24 elements

The differences between the two tools become already more evident when switching from 1D to 2D, which is shown in figures 7.4 and 7.5. Here, the model consisted of 20 nodes and 24 elements. The goal was a subdivision of the original model into 4 partitions as equal as possible.

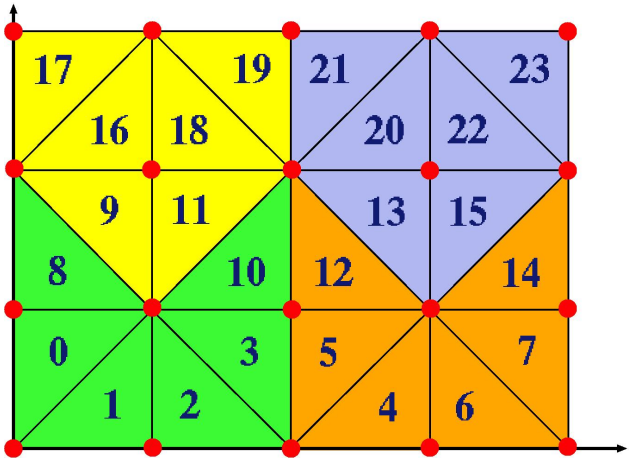


Figure 7.5: Domain decomposition into 4 subdomains by *Metis* of a model with 20 nodes and 24 elements

A humam being, given this simple model, most likely would have simply “cut” it in half and then maybe split the two halves evenly and symmetrically. The solution with which the algorithm hidden behind *Metis* came up with is shown in Fig. 7.5 and is quite close to such an “attractive-looking” solution in the eyes of a person. Comparing this to the result that was computed by the algorithm which underlies *Jostle* depicted in Fig. 7.4, any user would doubt its credibility.

Results similar to those shown in figures 7.4 and 7.5 when using models with 48 or 72 elements were obtained with varying numbers of partitions. The subdivisions often seemed to be completely

# of elements	# of edges	# of partitions	# vertices biggest partition	# vertices smallest partition	# of cut edges	balancing factor
5528	34845	4	1409	1336	928	1.020
		6	947	899	1472	1.027
		8	711	632	1583	1.029
		10	569	509	1943	1.029
		12	474	420	1930	1.028
17,141	102,541	4	4339	4245	1262	1.012
		6	2939	2613	1864	1.029
		8	2204	2056	2155	1.028
		10	1765	1509	2523	1.029
		12	1471	1303	3150	1.029

Table 7.1: Exemplary table with results obtained after subdividing differently sized input files using the domain decomposition tool *Jostle*

muddled, with no ordering evident. Sometimes not even a trace of symmetry could be recognized. But such examples show clearly that one must not forget that hard mathematical facts rule behind this domain decomposition of a model. A model that is split in the most perfect way with respect to minimizing communication and at the same time maximizing the workload for a certain processing unit may not necessarily look "good" or "symmetrical." Sometimes human eyes are easily deceived by such "attractive" symmetrical subdivisions and we, as the users of such sliced models, often forget about how and why we are working with domain decomposition. (The author herself experienced more than once great doubt about exemplary partitioned models, and much explaining and many demonstrations were necessary to convince others of the importance of domain decomposition, especially when such "weirdly" shaped subdomains were computed by the algorithms.)

What really counts in the end are the results obtained in regard to: How many edges have been cut? What is the balancing factor? How much do the biggest and smallest partitions differ with respect to their total number of elements? And finally: How fast could the domain decomposition be computed for a given input model?

At this point the statistic tool (see also chapter 6, section 6.12, starting on page 182 for a detailed description and a comparison between the two tools) came in handy. Originally just implemented out of curiosity by the author, it turned out to be very useful at this point. Many domain decomposition runs using numerous input data sets of different sizes were carried out. For each of the two domain decomposition tools, long tables with results obtained were adjusted and kept. An example of what such a table looked like can be seen in Table 7.1.

When examining Tab. 7.1, something becomes evident very quickly, and is a logical consequence of the whole domain decomposition process: With an increase in the number of partitions in which such a model has to be sliced, the number of cut edges also increases. It doesn't matter which tool is used, *Metis* or *Jostle*. On the other hand, it is hardly possible to keep a balancing factor of 1.0 as is often obtained with small input models that consisted only of a handful of elements or nodes.

What is also interesting to observe is the astonishing fact that the algorithms hidden behind the two domain decomposition tools are so sophisticated, efficient and accurate that, compared to the total of edges existing in the models in the two exemplary files in Tab. 7.1, the number of cut

edges is relatively small. In the second case, e.g., where the model consists of 102,541 edges, only 1262 have been cut when 4 subdomains have been created. Or when tripling the number of partitions, less than 3 times more, namely 3150 edges, have been cut. When now recalling what this means (points of obligatory communication by the processing nodes), the result appears to be very promising.

Very interesting finally was the outcome of the domain decomposition produced by the two tools when processing either files with different geometrically shaped elements or with a larger number of elements in the model. In Fig. 7.6 an example of the first case is shown.

Inputfile:	**Jordan Valley, 3D**
Number of Nodes:	**58092**
Number of Elements:	**116395**
Type of Elements:	**mixed (Pris, Tri, Quad)**
Number of Partitions:	**12**

Jostle		Metis
1.027732	**Balance Factor**	1.030044
45859	**Number of Cut Edges**	49494
9969	**Size of Biggest Partition**	9991
9259	**Size of Smallest Partition**	9426
4505.03	**Processing Time of Tool (ms)**	1209.24

Figure 7.6: Comparison between the domain decomposition results of *Jostle* and *Metis* for a real model that consists of different geometrically shaped elements

Fig. 7.6 demonstrates the differences experienced between *Jostle* and *Metis* quite well. In this example three types of elements occured in the input model: prisms, triangles and quadrangles. The data set was a real-life model that was processed from a geometrical model of the Jordan Valley. It consisted of about 58,000 vertices and 116,000 elements. A reasonable number for practical application of 12 subdomains had to be processed.

When comparing the results determined by *Jostle* and *Metis*, one can see two tendencies in opposite directions. *Jostle* has the slightly better balancing factor with 1.027 vs. 1.030 with *Metis*, and the number of cut edges, 45859, is significantly smaller than 49494, as calculated by *Metis*. The processing time of *Jostle* on the other hand is about 4 times higher at 4505 ms than that of *Metis*, which needs only 1209 ms.

When now comparing the results of the second example of a real-life input model shown in Fig. 7.7, the differences are even more conspicuous. This example shows a real-life model simulating a heat flow. The elements are all now uniformly shaped as triangles but the number of nodes exceeds 1 mio, which means 2 mio elements in this case. Fifty partitions had to be created – a number that is not likely to occur often; a smaller number of subdomains is more reasonable in practical use.

In this example, *Metis* has the slightly better balancing factor of 1.028, whereas *Jostle* only reaches a factor of 1.030. The number of cut edges is again significantly smaller for *Jostle*, with 88527 compared to 98585 for *Metis*. Comparing the two runtimes reveals the most striking discrepancy: *Jostle* is about 5 times slower than *Metis*, with 82576 vs. 16800 ms.

Input File:	Heat Flow, 2D
Number of Nodes:	1002001
Number of Elements:	2000000
Type of Elements:	triangle
Number of Partitions:	50

Jostle		Metis
1.0300	Balance Factor	1.0284
88527	Number of Cut Edges	98585
41196	Size of Biggest Partition	41136
29776	Size of Smallest Partition	38834
82575.5	Processing Time of Tool (ms)	16800.1

Figure 7.7: Comparison between the domain decomposition results of *Jostle* and *Metis* on a real model that consists of 2 mio. elements

These are just two examples from numerous input files that the author processed. It turned out that a balancing factor of about 1.025 to 1.030 was the average. It varied in dependance of the complexity and size of the input file. Here, both tools showed good results and it was impossible to predict a clear "winner" as to which would have the better balancing factor. Generally *Jostle* always did much better in minimizing the number of cut edges than *Metis* did. The differences in subdomain sizes for the biggest or smallest subdomain varied a great deal, again depending on the type of the input model. In regard to the runtime, *Metis* was often much faster than *Jostle*. Basically, *Jostle* never reached the fast runtimes of *Metis*, except for when processing very small input models with a handful of nodes and elements.

It should be noted, however, that the results presented here were all obtained in a computing environment which was technically inadequate. One mio. nodes in the input model was the maximum size that the input data sets were allowed to have. Files that exceeded this number could not be processed anymore; the available memory on the test machine – at 256 MB – was very small at the time of the test runs. Unfortunately the project and thus the financing ended before the final installation of a better equipped machine. As computation time is very expensive, additional test runs could not be performed anymore after the loss of financial support.

Nevertheless the results ultimately showed that it was clearly a good idea to have implemented both domain decomposition tools. If load balancing and especially the minimization of communication is the most important factor for the further parallelization process, *Jostle* should be the tool of choice – if runtime of the preprocessing is of no importance. At the time of this writing the preprocessing was decoupled from the pure parallelization process. This made it possible to have models already "chopped" long before the proper parallel processing, which would take place subsequently.

In case runtime is an issue, *Metis* might be the better choice in the end. It is much quicker than *Jostle* although its outcome is generally slightly worse with respect to communication aspects.

As both tools are equally implemented into the preprocessing software and their usage is kept as simple as possible when calling the wrapper (see also chapter 6 with (sub)sections 6.15 and 6.15.15, starting on page 189), the author would recommend taking the time to subdivide the given input model with both tools, if time is immaterial. By varying the number of desired partitions,

the user can quickly determine which tool is best suited to the specific conditions. A useful help in this evaluation phase are the statistic functions, described, discussed and compared in detail in chapter 6 in section 6.12, which can be found on page 182.

7.2 Implemented Data Structures in the Wrapper - Comparison and Discussion

The various data structures that seemed to be suitable candidates to be implemented in regard to speed but also memory consumption have already been introduced and discussed in full detail in sections 6.4 (pages 128ff), 6.6 (pages 145ff) and 6.7 (pages 158ff) of chapter 6, respectively.

This section now takes a closer look at all three of these data structures and discusses not only their proper benefits and disadvantages but also compares them among each other. As bit arrays and dynamically located arrays are just special versions of a standard array, the linked lists are compared to the more general version of an array whereas for the bit arrays and the dynamically allocated arrays, the differences are shown in comparison to each other.

As often the strengths or weaknesses of certain data structures and programs are described by using the so-called *Big 0 Notation*, the theory of this calculus will be introduced briefly at the beginning of this section. It is one of the fundamentals in computer sciences and helps to promptly describe basic features and behaviors of functions and their application on special data structures. The Big O Notation (BON) is an excellent tool to quickly compare such characteristics with each other and helps to give a first estimate of their benefits and disadvantages.

The section starts out with a general overview of parameters such as runtime estimation, memory consumption and performance analysis and of why the examination of such values is important.

7.2.1 General Aspects of Performance Analysis

When analyzing software one is interested in the performance or runtime of the program. Unfortunately, often this depends strongly on the input data. It would be most convenient if such a performance analysis could be done without considering this input data, as it generally changes and we don't know what the input data might look like with every call of the program.

For this reason it is necessary to have a closer look at the behavior of the program for the worst case and for an average case in regard to the input data. Such an analysis allows us to at least give an upper bound of the performance of the program whereas it is often impossible to give a precise boundary of the behavior, as one cannot always show that an exact data set exists with which a certain maximum of runtime, for example, can be reached. The only fact that we can prove by this performance analysis is that the behavior stays below such an upper bound.

Unfortunately, many details during this analysis depend strongly on the underlying algorithms, data structures, constructs that are used in the program and even the programming language itself, just to name a few. Sometimes even the hardware or architecture might be of importance. Accessing arrays on a normal out-of-the-box computer generally takes a certain time whereas doing the same on a NUMA architecture (see subsection 3.6.3 in chapter 3, starting on page 44) will often result in varying times for such accesses. Furthermore, not the "elegance" of an algorithm is important,

but its portability or stability. Sometimes first-rate algorithms might be so complex that it is not reasonable to implement them if they necessitate increased maintainance or constant work on such a program. Occasionally such complex but fast algorithms show their efficiency only for a typical type of input data whereas they perform poorly for the average case.

Another difficulty with such a performance analysis is the fact that often artificially designed input data sets are used but the real data sets which are going to be processed later on might vary extensively from the virtual ones. How should the resulting parameters be interpreted in such cases? Which criterion regarding the input data is the most important one? Is it the size? Is it its randomness? Or even its variance?

Especially in the field of searching for and sorting algorithms many detailed tests and studies have been undertaken. Here, a great deal depends on the underlying implementational facts like language and data structure, etc. As most programs, no matter what field they are applied, generally contain some searching and sorting procedures, such investigations and analysis are quite important.

It is impossible to come up with a universal solution to this problem. Everything depends on the specific situation. At least the field of complexity theory in computer science provides the interested and experienced programmer with a small set of important tools to thoroughly analyze either just important parts or even the whole program in such a way that its performance can be roughly predicted under average conditions.

It is not the goal of this thesis to provide a full introduction to the field of complexity theory. Various good books on this field exist, either for the interested novice or for the already advanced programmer. With the more practical aspect of programming in mind the reader should consult [Sedgewick02], [Cormen] or, in case he or she is undaunted by considerable amounts of mathematics and theory, the "bible" of complexity theory, [Knuth01] and [Knuth02].

7.2.2 Introduction to the *Big O Notation*

The origins of the BON as it is introduced and used nowadays already date back to a number theorist called Paul Bachmann. In 1894 he published his 2nd book, "Analytische Zahlentheorie," in which the basics of the BON were described. The Stanford professor *Donald E. Knuth*, one of the most important and well-known scientists in computer science and still active in research and science today, reintroduced these basics of Bachmann and also added the related *Omega* and *Theta notations*. Often, the terms BON, Bachmann notation or asymptotic notation are used interchangeably. As Knuth is called the "father" of the BON, most of the following in this theoretical subsection is either taken from [Knuth01] or [Cormen], when the former became too detailed or complex.

The notations that are used to describe the asymptotic running time of an algorithm or a program are defined in terms of functions whose domains are the set of natural numbers $\mathbb{N}$ = {0, 1, 2, ...}. Such notations are convenient for describing the worst-case runtime function $T(n)$, which is usually defined only on integer input sizes.

To describe the behavior of a program or algorithm, generally 3 different types of notations are used: the *O Notation*, Θ *Notation* and the Ω *Notation*. The most common one is the O notation whereas the other two are rather rarely applied.

The O Notation

The most important fact when analyzing a program or algorithm is what runtime it shows under the worst conditions. To describe such behavior the O notation is used, as it gives an asymptotic upper bound on a function, to within a constant factor. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions with

$$O(g(n)) = \{f(n) : \text{ there exist positive constants c and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

To indicate that a function $f(n)$ is a member of $O(g(n))$, we write $f(n) = O(g(n))$. Fig. 7.8 shows the intuition behind the O notation.

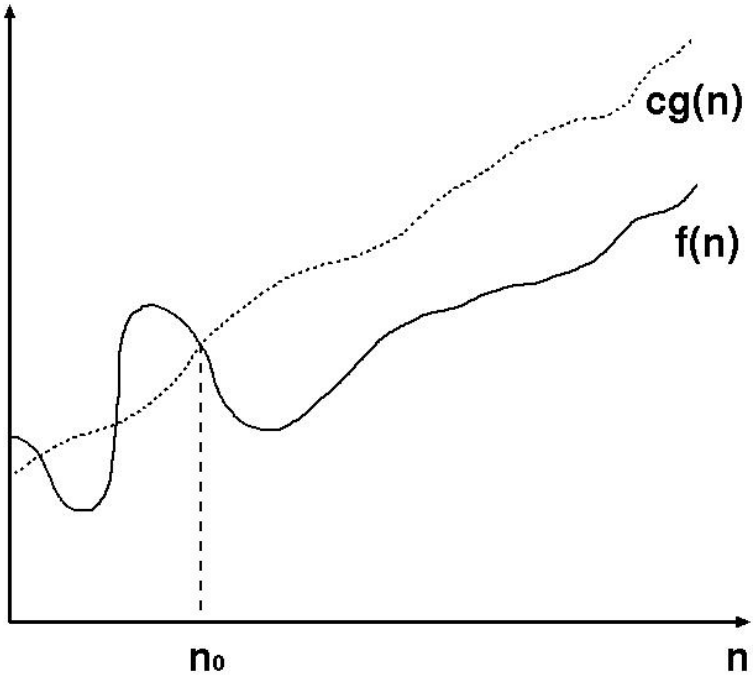


Figure 7.8: The O notation gives an upper bound for a function to within a constant factor: $f(n) = O(g(n))$

As can be seen in Fig. 7.8, the value of the function $f(n)$ is on or below $g(n)$ for all values n to the right of n_0.

The Ω Notation

Just as the O notation provides an asymptotic upper bound on a function, the Ω notation provides an asumptotic *lower bound*. Therefore, for a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions with

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants c and } n_0 \text{ such that}$$

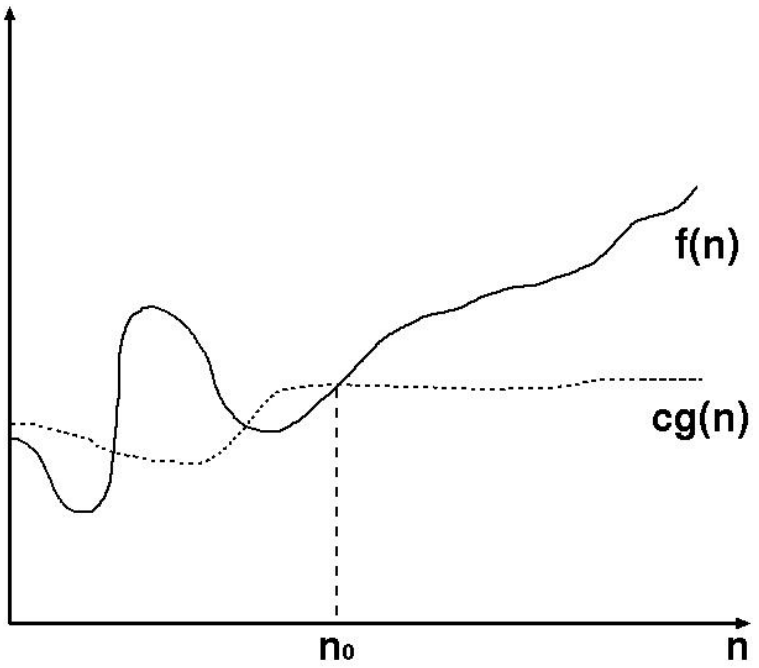


Figure 7.9: The Ω notation gives a lower bound for a function to within a constant factor: $f(n) = \Omega(g(n))$

$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Fig. 7.9 shows the intuition behind the Ω notation.

As can be seen in Fig. 7.9, the value of the function $f(n)$ is on or above $g(n)$ for all values n to the right of n_0.

The Θ Notation

The third and last of the notations introduced here is a combination of the aforementioned O and Ω notation. It is called the Θ notation and is defined as follows: For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions* with

$$\Theta(g(n)) = \{f(n) : \text{ there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

This means a function $f(n)$ belongs to the set of $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. Fig. 7.10 gives an intuitive picture of the functions $f(n)$ and $g(n)$, where $f(n) = \Theta(g(n))$.

As can be seen in Fig. 7.10, for all values of n to the right of n_0, the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an *asymptotically tight bound* for $f(n)$.

Although $\Theta(g(n))$ is a set, we write "$f(n) = \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$, or "$f(n) \in \Theta(g(n))$."

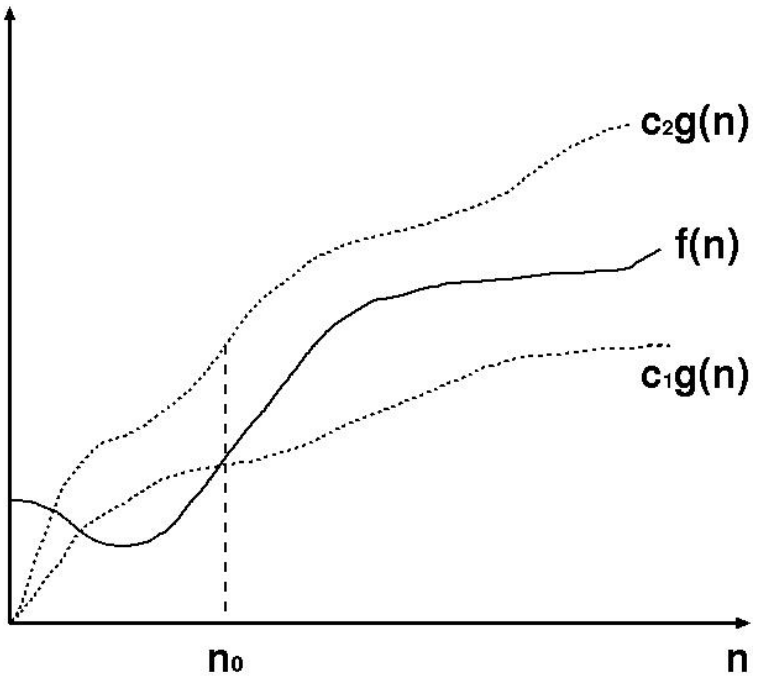


Figure 7.10: The Θ notation bounds a function to within constant factors:
$f(n) = \Theta(g(n))$

The definition of $\Theta(g(n))$ requires that every member of $\Theta(g(n))$ be asymptotically nonnegative, i.e., that $f(n)$ be nonnegative whenever n is sufficiently large. Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty. We shall therefore assume that every function used within the Θ notation is asymptotically nonnegative. This assumption holds also true for the other asymptotic notations, Ω notation and O notation.

Now, what is this all good for? To demonstrate the strength of these notations we examine a little example. One should not forget that when making rough estimates, it is permissable to discard low-order terms and also to ignore leading coefficients of the highest-order term.

We wish to show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. To do so, we must determine positive constants c_1, c_2 and n_0 such that

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for all $n \geq n_0$. Dividing the above by n^2 yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

The right-hand inequality can be made to hold for any value of $n \geq 1$ by choosing $c_2 \geq \frac{1}{2}$. Likewise, the left-hand inequality can be made to hold for any value of $n \geq 7$ by choosing $c_1 \leq \frac{1}{14}$. Thus, by setting $c_1 = \frac{1}{14}$ and $c_2 = \frac{1}{2}$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Certainly, other choices for the constants exist, but the important thing is that *some* choice exist. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$. A different function belonging to $\Theta(n^2)$ would usually require different constants.

Let us consider any quadratic function with $f(n) = an^2 + bn + c$, where a, b and c are constants and $a > 0$. As we said earlier, we may discard the low-order terms, which results in $f(n) = an^2$. Ignoring the constant a yields $f(n) = \Theta(n^2)$. Formally, one could do the same thing by taking $c_1 = \frac{a}{4}$, $c_2 = \frac{7a}{4}$, and $n_0 = 2 \cdot max((\frac{|b|}{a}), \sqrt{(\frac{|c|}{a})})$. The reader may verify that $0 \geq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ for all $n \geq n_0$.

One should also note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since the Θ notation is a stronger notation than the O notation. Written set-theoretically, we have $\Theta(g(n)) \subseteq O(g(n))$. This means that our proof that any quadratic function $an^2 + bn + c$, where $a > 0$, is in $\Theta(n^2)$ also shows that any quadratic function is in $O(n^2)$. What may be more surprising is that any *linear function* $an + b$ is in $O(n^2)$, which is easily verified by taking $c = a + |b|$ and $n_0 = 1$.

Sometimes one sees something like $n = O(n^2)$. In the literature, the O notation is occasionally used informally to describe asymptotically tight bounds, i.e., what was defined using the Θ notation above. Unfortunately, this misuse of the O notation instead of the correct Θ notation is increasingly found in publications nowadays.

The O notation is often used to describe the worst-case runtime of an algorithm or a program by merely inspecting its overall structure. Let us assume that a small C program, consisting of some simple C command lines, 10 lines, for example, and a nested loop statement with the indices i and j which both run from 1 to n, which is n times n iterations. Each of the simple C statements have a cost of $O(1)$, which is constant. The nested loop structure yields $O(n^2)$. This results in $10 \cdot O(1) + O(n^2)$, which is $O(n^2)$. The actual runtime depends on the particular input of size n.

There are more notations, like the *o notation* or the *ω notation*, as well as numerous mathematical laws on how to apply all these different notations. But to dive into more detail here does not serve an understanding of this thesis and might just confuse the reader unused to the very theoretical field of complexity theory and runtime performance. The above introduction will be more than sufficient for examining the different data structures and their behavior when applied in a C program later on.

For the upcoming subsections we are going to assume for the sake of simplicity that accessing the memory always takes a constant time T. It doesn't matter if we are going to do a write or a read access of the memory. We have to make this simplification as otherwise, a comparison of the data structures would not be possible. One should recall that even basic operations required in a computer must deal with such diverse aspects as cache hierarchies, latency times or even bank conflicts, just to name a few. Only by using a certain level of abstraction is it possible to design a universal model that can be underlain in such a way that a reasonable comparison of certain data structures becomes principally possible. Furthermore, the size of input data is some integer number n of the arbitrary size and number of stored data already existing in a certain data structure of size N.

7.2.3 Advantages and Drawbacks of Linked Lists

The data structure of a *linked list* was already introduced and discussed in section 6.4 of chapter 6, starting on page 128. The lists could be either singly linked or doubly linked by being of a linear or circular structure. Each of the list elements has one part to hold the values that are stored and another part where a reference is stored that points to the successive list element, and in case a doubly linked list is used, a third part that gives information about the predecessor. Linked lists are a so-called *self-referential datatype*.

Whenever we work with linked lists we have to make sure that we have allocated enough memory for inserting new list elements. In this specific case we assume that we have already reserved this necessary memory space in such an insertion situation.

Runtime Approximation of Linked Lists

No matter if we are going to insert, read or delete a certain list element, we always have to search for the correct position within a linked list. In the optimal case, the position within the linked list where one of the actions, like reading, inserting or deleting takes place later on, is at the very beginning of the list, which results in $O(1)$. In the worst case, however, the position is at the very end of the list; this is $O(N)$. Especially in the latter case we have to always read the stored value in the actual list element that we are examining. If it is not the correct position for the upcoming action, in order to find the successive list element we have to read the pointer value and dereference it until we have finally found the correct position in the list.

We assume that reading a list element is accomplished in a constant time, which results in the worst case in $N \cdot O(1) = O(N)$. Similarily, dereferencing the pointer in the list element for finding the next list element is also done in a constant time, which again results in $N \cdot O(1) = O(N)$, for the worst case. This leads to $2 \cdot O(N)$ which is still $O(N)$.

In case we want to delete a certain list element or even insert one, we always assume the action for deleting or inserting to be accomplished in a constant time. Therefore, also for these two additional cases, deletion or insertion, we still end up with a cost or time of $O(N)$ in the worst case. It should be noted though, that the action itself, either reading, inserting or deleting without the search process, is done in $O(1)$ with the assumption of a constant time for the proper action.

As we are going to process a total of n as the size of the input data, our algorithm will have a total upper bound of $O(n)$ in the worst case. But all of this just holds true only for an unsorted list!

In sections 6.9 (see pages 176ff), 6.10 (see pages 178ff) and 6.11 (see pages 180ff) of chapter 6, the various transformations of the data that are necessary throughout the course of the wrapper program were described in detail. Whatever was accomplished with the former input data was always done in an assorted manner. The only time the data was completely in increasing order and could just be read without the necessity of checking the order was at the very beginning of the program, when the input data sets are processed.

Now coming back to the performance analysis in regard to the wrapper, we suddenly realize that we have costs of $O(n^2)$ although we had explained previously that linked lists only show a behavior of $O(n)$. What has happened?

The tiny but consequential difference that leads to the increased costs and time results from the sorting. Especially when having a look at the very first step in the wrapper, the input data processing, this quickly becomes evident: The elements stored in the input data files are given in increasing order, which means when they are inserted in a linked list, which itself is sorted with an increasing order, one has to always start at the very beginning when searching for the correct position of the datum to be inserted. As the data is already sorted, the right location in the list where the element has to be finally stored is always at the end of the list. For the first element the correct position is at the beginning of the list. The second input datum has to be stored behind the first element, because of the increasing order, and so forth. This means, when inserting element no. N, already N-1 stored elements in the list have be examined for their element number until element N can be appended again at the end of the list. For the last element with no. n we have n-1 other elements already stored in the linked list.

When realizing this in C, two nested loops are necessary to accomplish such a search. As we have to do the searching and inserting for all n elements in the input data set n times in the worst case, the real cost of the usage of the linked list in the wrapper already just processing the input data sets results in $O(n^2)$.

As described further in sections 6.10 (see pages 178ff) and 6.11 (see pages 180ff) of chapter 6, we still continue to maintain our temporary and auxiliary data in a sorted, increasing order. Although the total lengths of the linked lists might no longer be of a length going over a full n, the worst-case performance still stays at $O(n^2)$.

It should be noted briefly that one could improve the list processing by simply saving the position of the last list member. In this way it would not be necessary to step through the complete linked list when the elements are already sorted in order to find the correct position at the end of this linked list. Then setting up the initial linked list with the sorted input data sets would only result in O(n). Unfortunately, not only one but many aspects have to be considered when choosing the most suitable data structure, and it turned out that on the underlying platform of the HPC environment, the usage of pointers is not advisable.

General Advantages and Drawbacks of Linked Lists

When just considering linked lists in general, something becomes evident right away: For every datum that is going to be stored within a linked list, at least one pointer has to be stored in addition; for doubly linked lists, even two pointer values. Although we have seen in the previous subsection that in theory in the worst case only $O(n)$ is the cost for using a linked list, in practical life it will clearly make a difference if not 10 mio. but 20 mio. or even 30 mio. data elements have to be stored when using a doubly linked list.

Already previously in this thesis, various examples demonstrated what this means in regard to memory consumption. When assuming a 32-bit architecture, which means a word size of 4 bytes, only the data itself will consume about 10,000,000 x 4 bytes = 40,000,000 bytes, which is about 40 MBs when storing, for example, simple integer values. When now adding another 40 MBs just for the pointer values, we already need a total of 80 MBs simply to store this information in a singly linked list. When using a doubly linked list this even adds up to a total of 120 MBs, as each pointer needs 4 bytes of storage space in the memory.

When the memory of the computer has a size of only 512 MB (as was standard not too long ago for an average computer), a doubly linked list for storing the 10,000,000 input data consumes a little less than a quarter of the overall, available memory. Not only the storage of these elements reqires a great deal of time but also their processing, like reading, maybe searching, assorting, etc. Now again, assuming 10,000,000 single values to be processed in one way or another, and assuming a constant time for storing, writing and/or reading them, the increase in time for each action quickly becomes evident. When now coming back to the O notation, 10,000,000, 20,000,000 or 30,000,000 single items are still all covered by $O(n)$ in the worst case but now a clear difference definitely exists when analyzing performance issues on a theoretical basis or for a real existing, underlying architecture. All of a sudden there is a notable difference between something that equals $1 \cdot O(n)$ or $3 \cdot O(n)$, for example.

One should not forget that not only the lists themselves demand extensive storage space but generally various additional and auxilary variables and parameters have to be kept in the memory in order to process the data. Sometimes the temporary data structures or variables have the same size

as the original input data, which means in our example above with the 10 mio elements that the memory might be already 50 % or more occupied, just in case a second or even third linked list has to be kept in the memory for processing the original data.

Therefore, the decision of using a singly or doubly linked list in the program should be considered thoroughly before the coding process. Sometimes, depending on the given situation, despite the overhead for an additional pointer value for each of the data values to be stored, a doubly linked list might be of advantage over a singly linked list. It could eventually save more overhead than is necessary to administer and work with a singly linked list in a given case. The extra memory consumption for a second pointer and the increase in complexity still might be less problematic than the additional temporary variables or parameters, etc., in case a singly linked list is used, although its complexity is less than that of a doubly linked list. One never can say in general what is best; an analysis of the given situation is definitely essential.

For the rest of this section we now assume that we are considering only singly linked lists. The same that is valid for this type of list also holds true for the doubly linked lists but in the case of the preprocessing given for the implementation of the wrapper, only singly linked lists were of advantage and seemed suitable, not doubly linked ones. For this reason we no longer consider doubly linked lists here.

Clearly, one of the major advantages when working with linked lists is their simple handling. The insertion but also deletion of a list element is very easy when the respective location in the list is finally found. Also setting up such linked lists is accomplished in hardly no time. But still, there is always the extra overhead for the additional pointer information. Thus, space saving is not really one of the major strengths of the linked lists. But it could be of importance when one thinks about the fact that generally there is no “empty space” within such a linked list. Normally, only “real, existing” information is stored in linked lists, an “undefined value” would never be stored in such a data structure, as it could happen under certain circumstances in arrays, as discussed previously in section 6.5 of chapter 6, starting on page 137.

Whenever we have a list element present in a linked list we normally also assume that it has some kind of value. This means, we have not wasted any memory space by “storing gaps,” as could happen with arrays. Regarded from this point of view such linked lists can be memory-space saving.

One should not confuse this “undefined value” with zero values as they can be found in sparse matrices. In such matrices we has just a few non-zero entries, whereas most entries are explicitly set to zero. But thus, they have a “true value” which is zero. The “undefined value” as it is meant in this regard here a location in memory which was not explicitly initialized in one way or another, simply as it was depicted in Fig. 6.18 on page 139 or in Fig. 6.19 on page 143 or like it can also happen in case someone forgets to initialize an array structure, assignes values to certain but not all of the array elements within the memory. The array elements without having been assigned a true value will hold any value but this generally will differ with any execution of the program (see also the basics of arrays as discussed in section 6.5 of chapter 6, starting on page 137).

As we do not have a directly accessible data structure, we have in any case an evident overhead for the pointer evaluation. Even nowadays, on modern computer systems, such actions still require various micro-instructions until a pointer is finally evaluated and the memory location it points to is found. For this reason, one would always try to avoid data structures with a majority of pointer operations and the need of dereferencing them. Sometimes this is impossible but, depending on the underlying hardware, it can be of advantage to rather implement a different, less sophisticated data structure than very elegant linked lists, simply because of this pointer problem.

Since linked lists are not directly accessed like, for example, an array, one cannot use an index or something like the element number to directly find and access the list element. Whatever one wants to do with the list and its elements, painstaking search operations are always necessary before one can start with the actual operations, like inserting or deleting. And even the search itself is not as fast as with other data structures because of the problem of pointer evaluation.

One big advantage of linked lists is, however, the fact that small "gaps" within the memory can easily be used. In comparison to a data structure that always has to be stored "in one piece," like, for example, an array, the linked list can be distributed all over the memory in the smallest storage spaces whenever the available memory space is big enough to host a list element. In regard to overall memory usage, such a linked list might even be of advantage if the operating on the linked list structure and the additional pointer dereferencing time is not of any importance in a given situation.

One major drawback of linked lists is clearly the danger that by using the pointer structure in the list element one has direct access to the memory and the data that is stored there. If someone fails to take utmost care when working with pointers and their dereferencing, hazards might result from uncontrolled or misplaced pointer operations and unprotected memory accesses. Memory manipulation based on invalid and/or erroneous pointer evaluation often leads to false results not only in a certain memory location but also in regard to the overall outcome of a program. Such errors are generally extremely hard to find and to trace as they normally never exhibit the same apperance when debugging and testing the program. The extra overhead for debugging actions in such cases is usually enormous.

7.2.4 Advantages and Drawbacks of Arrays in General

In comparison to a linked list the standard array is a structure with a direct access; the basics of such static, normal arrays have already been introduced and discussed in section 6.5 in chapter 6, starting on page 137.

One typical characteristic of such an array is the fact that it has to be declared at least once at some time that memory space can be reserved and later used. As the name already indicates, the access to an array element is not so complicated as it is when using pointer structures as in linked lists, described in the previous subsection, but by a direct access over a subscript with which the correct position within an array can immediately be determined, no matter what dimension the array has.

Here, still the same assumptions that we made earlier for the linked lists hold true: N is the number of currently stored elements in the array, n is the total of the input data, T is a constant time for accessing the memory and that read and write operations as well as "resolving" a subscript take a fixed but also constant time. We further assume that the storage space for the array is already declared, reserved and allocated and that we can immediately start working with such a static array.

Runtime Approximation of Standard Arrays

No matter if we are working with an unsorted or a sorted array, the time for reading, accessing, writing or even deleting an element in the array always takes a constant time and is accomplished in $O(1)$. Why is that so?

When working with an array what happens internally is that the beginning of the array is determined. This is done in a constant but short time which results in $O(1)$. When determining the specific position within the array for a certain element the subscript simply is added to the position at the beginning of the array. This again can be accomplished in a short, constant time, so that we have here $O(1)$. Now reading, writing and deleting can be done directly at this exact position, which we have also assumed as being done in a constant time; therefore, this again results in $O(1)$. In total we have $3 \cdot O(1)$, i.e. $O(1)$.

It doesn't matter how many elements are already stored in the existing array – none, only one, N elements or even all n elements. For each element, each of the above actions always takes a time of $O(1)$, if we assume that the elements are not just inserted unordered into the array from beginning to the end but in dependence of their subscript. As we have a total of n elements and for all of them a time of $O(1)$ is consumed, we end up with a complete cost of $O(n)$ when working with such an array.

Unfortunately, matters are less auspicious when we now assume a completely unordered array. The elements will be stored one after another from the beginning of the array to the end without consideration of the ordering. If at some point in the further course of a program we now have to order the elements, things look rather "nasty." Although we can directly access each array member, in the worst case we have to move around each element n-1 times when ordering n elements.

In a program, such ordering operations are normally done using nested loops with two counter variables. As we have to do the sorting over n elements, a maximum of $n - 1$ reordering actions is necessary until everything is at its correct location. Just producing an ordered array (it is of no importance here if the ordering is increasing or decreasing), a total cost of $O(n^2)$ results in the worst case. Fortunately, there are many sophisticated sorting algorithms available with which the cost can be reduced, but the optimal case still requires a total time of $O(n\ log(n))$ operations.

With respect of the wrapper, assuming that the input elements are all already sorted in increasing order, we can now directly insert them immediately at their correct position within the array. No extra sorting steps are necessary, so we finally have $n \cdot O(1)$ operations in total that result in $O(n)$. In the course of the program the order is maintained and therefore we always remain with the costs of $O(n)$.

7.2.5 Comparison of Linked Lists versus Standard Arrays

When comparing the data structure of a linked list with that of a standard array the advantages and drawbacks most likely already have become evident through the previous subsections and the discussion of the costs and the necessary actions involved when working with one of the two data structures.

Although working with arrays automatically requires having the complete memory space already declared and allocated in the beginning, the benefit of having direct access when working with the single elements later on that are stored in the array is most striking. When computation time is a major issue to be considered during program development, normally arrays are definitely the obligatory means of choice.

The problem with arrays lies in their nature. When the precise number of elements to be stored in an array is not clear from the beginning, the programmer always has to assume the possible maximum number when declaring and allocating the storage space for such an array. If it turns out

later on that maybe in the worst case just one single element out of n elements is finally stored in the array, the memory space for the n-1 unrequired element storage locations is simply wasted and consumed in vain. That can be a major problem in such situations in which memory consumption is an important issue. Here, in most cases but not all, the linked lists are much more suitable.

The linked list structure consumes precisely that amount of memory space that will ultimately be needed. If n elements is the maximum that might be stored but in the end only one element was then placed in the linked list, only that much storage space is "eaten up." The space in memory remaining for $n - 1$ elements that have not been stored is not even touched and can be used for other things.

Now let us consider an average case in which we are going to store about half of the n elements. When we use a standard array we still have to reserve n times storage space for these n elements in advance, but we only have a total cost in this specific case of $O(n/2)$ because of the direct access, i.e., $O(n)$. (Note the tiny but important difference between the cost for the storage space and the cost for working with the data structure!)

When we now store the same number of elements in a linked list and assume that the data is already sorted when slowly setting up the list, we have consumed in the end $n/2$ in storage space for the elements themselves and another $n/2$ for the necessary pointers that have to be stored for each of the list members. This results in a total of $2 \cdot O(n/2)$, i.e., $O(n)$. That was the same cost we had when using the standard array. But storage space is just one side of the coin. And recalling what we already indicated previously when discussing the cost for the linked lists in case the elements are sorted, we suddenly have costs of $O(n^2)$ when storing the $n/2$ elements.

This little but impressive example shows clearly that it is of utmost importance to thoroughly examine all facets of a given situation before the coding process. In case memory consumption is of importance and the true amount of data that has to be processed is rather small compared with the possible maximum of what could appear in the worst-case scenario, it is possible that a linked list would be more suitable than a standard but still directly accessible array.

On the other hand, in case the program has to be extremely fast and needs to be adapted in the optimal way to the underlying hardware whilst memory consumption is of minor importance, the standard arrays most likely will be the better choice.

Another aspect that should not be forgotten is the complexity and handling of the data structures. Dealing with linked lists involves assignment and working with pointers; someone who does not know precisely what he or she is doing may get into trouble. Deploying a standard array in comparison is something that even a beginner can do. Here of course some important things have to be considered, like a thorough initialization of the array as well as staying within its limits, but the proper handling itself is very easy. Thus, the likelihood of errors is probably much smaller when working with arrays than during the application of pointers when dealing with linked lists.

At this point a little nitpicking will be necessary: When working with linked lists, elements can be deleted from the list, which means they are really "thrown away" and the values of such elements are lost. Deleting in this sense is not possible when working with arrays. Here one can overwrite an array element or set it to an "undefined value" but it is not possible to physically dispose of such an element. There is still something stored in the array at the position of the element which is no longer needed.

7.2.6 Advantages and Drawbacks of Bit Arrays

When analyzing bit arrays, which were introduced and discussed in detail in section 6.6 of chapter 6, starting on page 145, it very quickly becomes apparent that such arrays are nothing but very specialized standard arrays. This way everything that was said previously in the above subsections of this chapter holds also all true for bit arrays.

The only difference on closer look arises from the fact that when using bit arrays the general minimum size for storing information is not based on the word size as the basic internal operating size but on working with the smallest possible unit within a computer. This way, storage space is reduced by the size of n divided by 8, as one byte holds 8 bits. Unfortunately, this gain in reduced memory consumption is offset by a strong increase in complexity, as now additional operations are necessary to find the correct location of an "array entry." Furthermore the administration of such bit arrays requires an extra overhead for additional auxiliary variables and parameters.

Still, depending on the given situation, bit arrays might be an interesting alternative, as they are very fast and the overhead of storing the proper information in the array itself is minimized. They operate at bit level, which is the smallest possible size in which information can be coded. If there is a way in storing values just by using 0 or 1, i.e., either a bit is set to 0 indicating "no value present" or set to 1 for "yes, the requested information exists," one has found the optimal way of storing certain data.

It should be noted at this point that the application of bit arrays is only reasonable with regard to sorted information. This results from the fact that the bit that has to be toggled is determined over the value of the datum that has to be stored. The data to be processed can exist in an unsorted manner but automatically when the bit array is introduced, the sorting is done as a "byproduct" when setting the bits.

When analyzing the costs of bit arrays we again assume a total input data of the size of n. The other important factor when using this specialized type of array is the dimension. Using 1-dimensional bit arrays results in costs of $O(n/8)$, which is still $O(n)$. The also holds for 2-dimensional bit arrays as used in an early version of the wrapper. Here, one of the indices iterates over all n elements, whereas the second one operates on the bit size, resulting in total costs of $O(n) \cdot O(n/8)$, this is $O(n^2)$.

The strength of bit arrays clearly lies in their "compactness" but the price for this is increased complexity. Although bit arrays still have costs of $O(n^2)$, they perform much better on average in real life. Here it is clear that theory with all its simplifications and the same facts, when finally applied to a real case, might differ by some extent.

A little example will quickly prove this: Let us again assume that we are dealing with 10,000,000 elements to be processed, and that element-to-element relationships have to be stored (their background is of no interest here). When using a normal, standard 2-dimensional array 10 mio. x 10 mio. elements have to be stored, which results in an enormous matrix. Given a word size of 4 bytes, the number of total bytes simply consumed for storing this giant matrix grows by 4, which is then 400,000,000,000,000 bytes or about 400 terabytes, a quite stunning size even for well-equipped computing environments.

Now let us take the same example and use a simple bit array. We still have to iterate over 10 mio. elements in one dimension, but to indicate that an element-to-element relationship exists we simply flip a bit in the row and column in this "virtual matrix" that represents the relationship of the two elements. On a normal computer 10 mio. elements represented by a simple bit can be stored in

only 10 mio. mod 8 = 1,250,000 bytes, which is clearly an eighth of the original size. This gives us a total size for the whole matrix of 1,250,000 x 10. mio = 12,500,000,000,000 or about 12,5 terabytes, which is again only an eighth of the size required when not using a bit array. Even if we have an additional overhead of a "couple of bytes" for the temporary and auxiliary data structures necessary to work with these bit arrays, the enormous difference in memory space consumed is evident.

When regarding the performance of a bit array it should be noted that the additional overhead of determining the position of the bit and whether it should be set or read is independent of the size of the input data. The operations necessary for this can be accomplished in a constant time, which is $O(1)$. In case we regard data to be stored in a bit array of the size of n in total, we still result with total costs of $O(n)$,which is exactly the same amount which we determined for a standard array.

7.2.7 Advantages and Drawbacks of Dynamically Allocated Arrays

As the dynamically allocated arrays (DAA) which were introduced in section 6.7 of chapter 6, starting on page 158, are just a specialized type of the standard arrays, the same that was valid for the latter also holds true for the DAAs.

The reason that one implements DAAs is that the final size of an array has to be known at the time of its declaration in order to allocate memory space for its later use. Often, such a size is not yet known during the time of the declaration and only results at a later stage during the course of the processing. The only way to overcome this problem is to declare the maximum possible size needed when using a static array. In an extreme case it could happen that this maximum that might occur and be a size of 10,000,000 integers which have to be stored, but in real life the effective and actual size only results in one. Thus, when having now declared a static array for 10 mio. integers, almost the entire space was "wasted" and could not be allocated for other maybe more important things.

DAAs try to surmount this problem by the method of reallocation. At the beginning a certain "minimum amount" is allocated so that one has something to work with. The initial amount of memory space should be chosen appropriately. It would be beneficial if at least the average cases worked with this much storage space. If chosen too large, memory space may be wasted, as it was in the previous example. But if the initial space is too small, additional storage space has to be provided by reallocation. This automatically involves a certain overhead for the allocation of a larger, new array and an obligatory copy process to move the information stored in the too-small array into the slightly bigger, new one.

If possible a thorough analysis of the requirements can result in a reasonable initial array size which in many instances works without problems. Only in those cases in which this initial size proved unsuitable is a reallocation process necessary.

With this technique a trade-off between memory consumption and additional overhead for extra operations is made. As the time that is needed to reallocate the new, larger array space is always constant, this results in costs of $O(1)$. Unfortunately, the copy operations are more problematic. When assuming that whenever we reallocate a slightly larger array, this involves setting up a complete new memory area, the copy operations start to become expensive. When, in the worst case, copying for n different variables n times a smaller array into a new, larger one, we end up with costs just for the copy operations of $O(n^2)$. This is still the same as for a typical static, standard array but then the usage of DAAs would not be reasonable.

The strength of the DAAs is the fact that the underlying operating system tries to minimize the overhead of reallocation by appending the additional memory size that the old array lacks to the end of the existing array if there is enough free memory space available to do so. Thus, no real data copying is necessary. But one cannot always rely on this fact. In the worst scenario, a complete true copying process of information is obligatory and a reallocation might be necessary everytime. Normally, this is not the case; here we see again that theory and real life might not always produce the same result.

In regard to memory consumption, in the worst case $O(n)$ results when indeed information of size n has to be stored. But should, for example, half of this size have to be stored and the rest discarded, only the true amount finally has to be kept in the array. This is still again $O(n)$ but there is definitely a difference between having to store 10 mio. elements and perhaps only 5 mio. in reality.

When using 2-dimensional DAAs, for example, storing a special relationship between two elements, as during the preprocessing when using the wrapper, one of the indices runs over all n elements, whereas the total number of related elements for one specific element just results during the course of the program. In the worst case this could result in a n:n relationship but it also could just happen to be a 1:n relationship. One never knows in advance. When using DAAs one relies on the fact that the average case is definitely way below the worst-case scenario (otherwise one would have used a normal, static array directly and would have saved him- or herself the hassle with the additional overhead for administering such DAAs and the likelihood of more errors.)

The "connection" between the static array that holds all n elements and the smaller, individually sized arrays that will be resized is established using pointer structures. (See Fig. 6.33 on page 162.) This means an additional overhead, but setting up these connections and administering this construction is again done in a constant time, thus resulting in $O(1)$, and for all n elements, in $O(n)$. As each element just has a number m as the maximum number of relationships to other elements, the total cost for a 2-dimensional DAA is not $O(n^2)$ as in the worst case, but indeed in average of only $m \cdot O(n)$, i.e., $O(n)$.

Again a short example: Let us assume 10,000,000 elements to be stored in such a DAA and their associated element-to-element relationship. We definitely have to set up one static, normal array in which we store all 10 mio. elements. When the elements are stored as simple integer values we need a total storage space of 4 x 10 mio. = 40 mio. bytes, which is about 40 MB. When assuming that each element has only a total number of m = 20 directly adjacent elements, the amount of memory space that is needed to finally store the whole element-to-element relationship over all input data results in 40 MB + 15 x 10 mio. x 4 bytes + 40 MB = 680 MB all together. This is obviously just a percentage of the 400 TB of storage space for a simple static 2-dimensional array.

One might wonder when doing the above calculations where the additional 40 MB come from. They represent the memory space that is consumed by linking the static array over the 10 mio. elements to the smaller arrays by a pointer structure. We need 10 mio. pointers which consume a space of 4 bytes each, which results in 40 MB.

7.2.8 Comparison of Bit Arrays versus Dynamically Allocated Arrays

Just by doing the simple math shown above, the benefits of using DAAs in certain cases will be obvious. But why is bit-array performance so poor although they are just operating on a bit level instead of a byte level, as the DAAs are doing?

The answer to this question is not directly evident but can easily be explained by the example of the element-to-element relationship which was used in the subsection above. When we assume a maximum size m of directly adjacent elements, we still have to allocate 10 mio. / 8 bytes for each of the single elements to store the relationship just for this one element. And even worse: Not only are we wasting this much memory space (although a short array of size m would be sufficient), but we are also just flipping m bits within the whole array. All the bits for the elements that are not in the vicinity of a certain element are set to 0. Thus, throughout the 10 mio. *bits* (i.e., 1,250,000 bytes) that are allocated for each of the single elements to store its relationship with the other elements, only a few bits of a total number m are finally set to zero.

And now it turns out that a bit array is indeed nothing but a static, normal array that has been "shortened" in size by a certain number, namely 8, as 8 bits comprise one byte. But, like in the example with the element relationship, we actually still work with a true 2-dimensional array structure although it already saves quite some memory space when dealing with high element numbers. Unfortunately a lot of storage space is simply wasted as the value zero means "not important." The overhead for operating on the bit arrays is theoretically done in constant time and does not matter, but anybody who has tried to do bit operations in C knows how complex such programming quickly becomes and that one has to pay great attention when detecting the correct bit position or when performing normally easy programming steps, for example. Great effort is needed by the programmer to not lose the overview, especially in a bigger programming project.

7.2.9 Summary and Concluding Discussion

In the given situation it turned out that the DAAs seemed to be the only useful data structure for implementing the various relationships and transformations that had been necessary during the preprocessing process. The bit arrays which were implemented in the first version of the wrapper worked extremely fast and seemed to be very convenient at first; by coincidence it turned out, however, that they have the major drawback of simply wasting too much memory space without really storing important information.

As memory consumption was of no special interest at the beginning of the project, and other aspects were concentrated upon, bit arrays seemed to be a good compromise between fast processing speed and saving at least some storage space. Only due to the fact that the underlying hardware was not sufficiently equipped during the coding process when testing more demanding data sets, was this "hidden bug" suddenly discovered.

The author can count herself lucky that this unfortunate finding occurred already quite early in the coding process; all of a sudden the issue of storage space and memory consumption became very important and one of the key aspects for the implementation of the wrapper. Consequently, at every stage of the coding process she constantly checked and tested to confirm that the best possible solution was achieved. Also for this reason, Valgrind (as introduced in section 6.8 of chapter 6, starting on page 170) became a very useful and also powerful tool for revealing the – sometimes awkward – "whole truth."

But one should not forget that each project is unique; especially in the field of HPC no two programs look alike and what might seem an optimal solution in one case could be totally inapplicable in another. Generally every programmer and engineer should sit down in the beginning and thoroughly examine the given requirements in regard to speed, memory consumption, time frame for coding, program development, testing and debugging, monetary issues, etc. This holds true not only for projects in the field of HPC but for all programming projects. Unfortunately, often this

has to be done in a hurry or even completely skipped, as the supervisors do not see the necessity for such an analysis. And even a solution that works properly and might be "perfect" can suddenly fail when underlying conditions have changed, as the author experienced herself quite painfully with the seemingly "brilliantly" working bit arrays which completely crashed unexpectedly although the program itself was faultless.

The DAAs turned out to work well, no matter which input data set was finally chosen for testing, and hardly any difference in processing speed could be observed for these data sets that also ran without problems with the bit array wrapper version. It was impossible to compare the two versions in regard to time, as the powerful machine in the HPC center was not only used by the author but also by many other users. Therefore the processing time assigned varied greatly. Testing both versions on a small average computer did not result in useful findings, as the data sets that could be processed there were simply too small and the processing time of the computer too fast to really see a remarkable difference.

The most striking contrast could be observed in the consumption of memory space used by either having the wrapper version crash when processing a certain number of input elements or having it smoothly finish its program run to produce a domain decomposition with the given parameters for the original input file.

The linked lists never really had been coded as a possible alternative for the wrapper, as their drawbacks are simply too apparent under the given circumstances.

7.3 The Work Conditions during the Project

Normally the work conditions in such a big computing project are of no importance for the outcome of such a thesis nor are they interesting to the reader. In this case, it had actually been more than helpful not to have the optimal conditions; due to this fact, major improvements could be made.

Until it was possible to work at the HPC center in Stuttgart, the coding, debugging and testing environment was limited by the given restrictions of a very poorly equipped, simple laptop with neither enough disk space nor sufficient memory. In the beginning the existing hardware seemed to cause no problems, and the work environment under Unix and Linux is quite the same everywhere if a good editor and an effective and powerful debugger are available.

On most Unix and Linux installations there is generally either a `vi` or an `emacs` as one of the most popular editors existent. `gcc` as the standard compiler and `gdb` as the best-known debugger are also at hand so that it makes no big difference during the coding process if one works on an out-of-the-box computer, a big number cruncher or a powerful parallel computer, all running some derivates of Unix or Linux.

At some point in the course of the project the program development and the coding was done in parallel on the little laptop mentioned above as well as on the parallel machine or an almost-as-powerful test environment at the computing center in Stuttgart. When the bit arrays had been readily implemented on both machines it suddenly turned out that the code ran without problems on the bigger and better equipped machines but once in a while crashed for seemingly no reason on the laptop, although the program itself as well as the memory organization, etc. had been completely tested by tools like gdb and Valgrind (see also section 6.8 in chapter 6, starting on page 170). No errors could be detected and nobody had an explanation for the "weird" behavior of the wrapper.

Shortly before giving up in frustration, the author discovered the reason behind the laptop crash. What was happening became obvious when one day all kinds of input data files had been checked on both platforms in parallel. Up to a certain size no problems occured on the HPC environment or on the laptop. But when the input files exceeded a certain number of nodes (and later on, of course, the resulting number of elements in the model), the version running on the laptop ended abnormally.

Analyzing these special problem cases with the specialists present at the HPC center in Stuttgart who had accompanied the project most of the time, it turned out that although the memory-space-saving bit arrays had been implemented, thoroughly tested and found to be well adapted, it was the bit arrays that ultimately caused the problem.

Although they save a great deal of memory space, are extremely fast to work with and are optimally customized to the given computing environment, the bit arrays still consume too much memory space by being "empty" most of the time. Further memory space is depleted due to the great number of auxiliary variables and other data structures that are needed to administer and use these bit arrays accordingly.

Had the coding and development process been started and completed exclusively on the better equipped HPC environment, this "bug" would not have been discovered or, if so, at least at a much later time. This way the author can count herself quite lucky that this unanticipated problem turned up so early in the implementation and testing process. Now data structures better suited and adapted to the given situation and environment could be searched for and implemented by using dynamically allocated arrays, as described in section 6.7 of chapter 6, starting on page 158.

Having an inadequate working environment led not only to an improvement in the data structures. Also some "administrative restrictions" imposed on the project by its leadership ultimately resulted in the implementation of certain features or at least showed the necessity of expansions or improvements for future versions of the wrapper.

Problem-solving during the project clearly profited from the usage of both domain decomposition tools, *Jostle* (see section 5.4 in chapter 5, starting on page 107) and *Metis* (see section 5.5 in chapter 5, starting on page 112), finally implemented in the preprocessing code.

The upcoming version of the wrapper `domdec_v03_3` is also a product of difficulties with the persons in charge. When implementing such a big parallelization project which is based on the transition of an already existing sequential code, a so-called *feature freeze* is obligatory and mandatory. Typical interfaces are defined and established. The structure of the input as well as the output files are described and determined, and other necessities and prerequisites which are important under these circumstances are investigated and nailed down.

Unfortunately none of this took place during the entire course of the project. The sequential version was continuously being changed, new features added, file formats constantly altered, etc. It was impossible to get a clear statement as to what features should now be in the wrapper and which ones not. When certain things had been coded, others had already been changed again. Furthermore, teamwork was impossible in the project team, as most members of the group hardly spoke English; some of them not even a work German. Cooperation was barely existent; collaboration was "enforced." Even a life-saving version control hardly existed in the beginning, as no version control system was available. The people coding hardly ever spoke to each other and the person responsible for merging the sequentially existing versions was definitely more than relieved to be able to pass on the resulting version to the next person for the subsequent merging process.

Learning from these work conditions, extensive changes from wrapper version domdec_v03_2 to version domdec_v03_3 had been planned. It was unfortunately not possible to finish the later version before the given time and financial frames ran out. The upcoming sections give a short overview of the transformation of domdec_v03_2 to domdec_v03_3; they also indicate future improvements and expansions that might be implemented at some later stage of the project.

7.4 Changes and Improvements between Wrapper Version domdec_v03_2 to domdec_v03_3

In this section the evolution from wrapper version domdec_v03_2 to domdec_v03_3 is briefly discussed. Various suggestions for improvements and future expansions for the wrapper which will also be discussed, not here but in the upcoming sections.

7.4.1 Handling of Input Files

The biggest change from version domdec_v03_2 to domdec_v03_3 can clearly be found in the handling of the input data sets. As the given test and real-life data files that had to be read in for preprocessing the stored data changed a number of times in their structure and the routines affected had to be changed various times, it soon became evident that some more flexible method of input data treatment was necessary.

On first sight the changes weren't too obvious for the independent observer but they had strong effects on the data handling. The files always consisted of something called a header or preamble in the first view lines of the file. Next followed a listing of all node numbers of the model with their geometrical coordinates in 1D, 2D or 3D space. The last and biggest part consisted of a long listing of each element with certain important paramenters, like the geometric shape of the element and its configuring vertices, besides other things.

The general structure did not change; what changed was the composition of the small details which might not seem to be of importance to a human being but can show extreme effects when they have to be processed by a computer program. And that was exactly the problem: At some point during the project 3 or 4 different versions of input files containing the variables and parameters given for the elements changed not only in their number but also in their ordering. The only way to recognize this in time was by checking the preamble of the given files, which unfortunately never looked exactly the same for the input data set files concerned.

The final solution was to implement a "universal input file parser" that would check the preamble and, just for safety reasons, also the different types of lines in the files with their structure and their entries. This way the user would no longer have to make sure that the input files had the correct structure when calling the wrapper in its latest version but would just give any input file that was known up to that point and simply have it processed.

A second problem was that until now the file recognition had to be done over the file name extension or over the structure of the header in the file. At some stage the same file name extension happened to be used for an input file with a different structure, leading to total confusion.

With a universal input parser, neither the user nor the programmer would have to be concerned with such details and problems. The file would simply be opened and analyzed by the reading-in

process. At the same side everything would stay as transparent for the user in the handling of the wrapper as it was designed to be. The two voluntary requirements turned out to be more complex in their realization than expected.

7.4.2 Dynamical Adjustment and Adaption of Array Size during Reallocation

In section 6.7 of chapter 6, starting on page 158, the dynamical allocated arrays were described. With the given test and input files it finally turned out that a size of 5 integers was best suited for the new array to be appended. Unfortunately, many of the existing files which the author had for testing and evaluation had triangular-shaped elements. Only a few of them had elements with another structure, nor could the author fall back on such interesting input files as those consisting of a mixture of many differently shaped elements.

Two ideas arose out of this situation. One of them was to take the currently implemented size of 5 integers when reallocating and increasing an array as a default value but to allow an experienced user of the wrapper to individually alter this value to his or her personal taste. The other idea was to dynamically adjust the size of the array to be appended in accordance with the given element size or circumstances.

The author assumes that depending on the geometrical shape of the elements, 5 might not always be the best number to choose. It turned out that this final size, which is currently implemented, resulted from the majority of triangular-shaped elements in the given input files, but it could also be that a different value would be better suited for linear or tetrahedral-shaped elements, for example. To finally prove this assumption, many more input files are needed which could not be obtained until the end of the project.

In the current state of implementation the first idea is pursued. This allows the user to choose between a certain range of input values for the array to be newly allocated. The former size of 5 will still be set as the default, especially if the user is not experienced with the wrapper or lacks enough background information or knowledge in this field. In case someone wisehs to influence the array size, he or she will have to explicitly give an optional commandline switch and an integer value when calling the wrapper.

7.5 Future Improvements and Expansions of the Wrapper

7.5.1 Parmetis Expansion for Parallel Domain Decomposition

In chapter 5, starting on page 89, the various aspects of domain decomposition were discussed in quite some detail. Based on this knowledge, a comparison and future outlook followed towards the end of the chapter in section 5.6, on page 116ff.

The following two main trends were discussed there: either keeping the domain decomposition as a stand-alone application outside any further parallelization processes or, contrarily, to completely intregrate it directly into the further model computation as one major initial stage of the overall parallelization process.

The reasons favoring one or the other trend were discussed generally; which of the two solutions, stand-alone or integration, will be realized, depends on the given circumstances.

In addition to the already existing and integrated domain decomposition tools, *Metis* (see section 5.4 in chapter 5, starting on page 107) and *Metis* (see section 5.5 in chapter 5, starting on page 112), it might be interesting to also implement a parallel domain decomposition tool named *ParMetis*, which is one of the many program routines that come with the Metis program family.

The Basics of ParMetis

ParMetis is an MPI-based, parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. It extends the functionality provided by Metis, the sequentially working domain decomposition tool, and includes routines that are especially suited for so-called parallel *adaptive mesh refinement* (AMR) computations as well as large-scale numerical simulations [ParMetis].

The algorithms implemented in ParMetis are based on the parallel multilevel k-way graph partitioning (the same algorithm on which also `kmetis` is based; see also section 5.2 and subsection 5.5.1 of chapter 5, starting on page 99 and 113, respectively), adaptive repartitioning and parallel multi-constrained partitioning schemes.

ParMetis provides the following five major functions:

- graph partitioning
- mesh partitioning
- graph repartitioning
- partitioning refinement
- matrix reordering

Future Implementation of ParMetis into the Wrapper

The goal is to implement ParMetis as a new partitioning tool in addition to the already coded tools *Jostle* and *Metis*. The inclusion would again be done completely transparently to the user. Over a special commandline option ParMetis can then be picked in the same way that Metis and Jostle are chosen in the current version. (In section 6.15 of chapter 6, starting on page 189, the usage of the different commandline options of the wrapper program is explained in full detail).

As the reader can see, when checking Appendix C, starting on page 253, the author had already planned to implement ParMetis for a new version of the wrapper in the near future, as the switch `-c` or its long equivalent `--parmetis` is already listed not only for version `domdec_v03_2` but also `domdec_v03_3` in their current man pages.

Having ParMetis included in the wrapper does not solve the problem of whether a stand-alone version is better suited than an all-comprehensive, completely parallel model computation. But in case a user has access to a full parallel computing environment, he or she can already switch to a full parallel domain decomposition and use the strength and the benefits provided by such a well-equipped hardware. If the preprocessing is still done stand-alone and decoupled from the main model computation, the user at least profits from the fast environment and obtains his or her partitioning results most likely much faster than on a sequentially working architecture.

In case the wrapper is included in the whole model computing suite and has ParMetis implemented in it, the user can now generate a partitioning with the full parallel support before he or she starts the parallel model calculation. In case just a sequential partition tool is implemented, as it is in current state of the wrapper, it is still possible to include the preprocessing process in the full parallel model computation; but the HPC environment cannot be utilized to its full extent, as it just runs on one single computing node.

7.5.2 Support of Weights for Vertices and Edges

In the current version of the wrapper only the nodes and the edges that result from the input data sets when setting up the adjacency graph are supported. The models that had to be preprocessed so far abstained from the usage of weights. But as both, *Jostle* and *Metis*, offer the deployment of weights for nodes and edges, and their usage might be of some interest for future models and their input data sets, upcoming versions of the wrapper program are already prepared in a currently inchoate version for such a use of weights.

When examining the man pages for the wrapper of version `domdec_v03_2` but also `domdec_v03_3`, the reader will already "spot" a commandline switch called `-w` or its equivalent long counterpart `--weights`, with which the usage of the weights can be turned on. In case no weights are to be processed, the option `-n` or `--noweights` can be set.

In the actual version of the wrapper the usage of weights is set to "NO" per default. The necessary algorithms are not yet fully implemented. Right now, in case someone is interested in the handling of weights, all weights are equally set to 1 for either the nodes or the edges. Different values for the weights are not possible in the current wrapper version.

The author had already begun to implement a full deployment and handling of the weights for curiosity's sake but was unfortunately not able to finish this additional feature in the given time and financial frame. The implementation of the weights was treated more as a "hobby project" than as a serious part of the preprocessing process, since it was not required, nor did anyone show interest in the handling of the weights.

7.5.3 A Graphical User Interface for the Wrapper

Already at the beginning of the whole preprocessing process it quickly became evident that the average user of the wrapper is more familiar with a Windows-based operating system than with the typical commandline interface as it is standardly provided with most Unix and Linux applications. This fact led the author to providing a so-called *graphical user interface* (GUI). This way the wrapper can be operated quite comfortably using the mouse with, for example, radio buttons, check boxes or simple text-input fields. Such a GUI normally operates on a window manager or a graphic-based program loader like Microsoft Windows and is started in its own window.

The GUI itself is nothing more than an "attractive-looking" and handy interface which is operated on one side by the user and which sets up and controls the wrapper by calling it, using the information provided by the user.

As a simple example one can consider the domain decomposition tools that are included in the wrapper, currently *Jostle* and *Metis* (see also chapter 5, starting on page 89). As explained in detail in chapter 5, Jostle is currently set as the default partitioning tool. When now providing a GUI,

the domain decomposition tools that are implemented in the wrapper would be listed, including a radio button for selecting them, independently of how many tools are currently available and implemented. The user now makes his or her choice by clicking on the proper radio button of the appropriate tool and in case he or she agrees with the pre-selection, no further steps are required, as in this case Jostle would already be checked.

For those readers not familiar with these kinds of elements, a radio button can easily be envisioned as the kind of radio buttons on old radios that needed to be pushed. Once one of these buttons had been pressed it could just be "reset" by pushing another one; thus, at least one of the buttons was always pushed down.

As another suitable example, we could take the classical "ON/OFF" switches as discussed in detail in section 6.15, starting on page 189 of chapter 6. These commandline options, like `--statistic`, `--verbose` or `--test` (or their short counterparts `-s`, `-v` and `-t`), would be given in a list within the GUI and provided with check boxes. In comparison to the radio buttons above, the check boxes do not come with a pre-selection. With these elements of a GUI the user has to explicitly "check" which option he or she would like to have activated and which not. This way the user could select whether he or she is interested in one or several of the provided ON/OFF switches, and if not, simply not check the appropriate box of the option.

Handling text strings is also not problematic when using a GUI. Here, an input field where the user can type in his or her strings is provided. Such a field can be used, for example, when the user wants to give his or her own output file name, as described in subsection 6.15.7 on page 194ff. If no file name is typed in, the wrapper is called and creates and names the resulting `dcc` file in the same fashion a commandline call would. Only in case the user has provided his or her own output file name does the GUI pass this to the wrapper when calling it, as if the `--ddc` switch or its shorter counterpart `-d` had been set on the commandline.

It should be clear at this point how such a GUI works. In what language it will be realized depends on various different factors and criteria. A full discussion of what would be the most suitable language for such a GUI is far beyond the scope of this thesis. Typical languages nowadays for developing GUIs are either a programming language called *Java*, developed by Sun Microsystems (see also [Java]), or *Qt*, originally invented by a company called Troll Tech (see also [Qt]). Both languages provide support for the most widely used platforms like Windows, Macintosh and Unix/Linux systems, and the code is normally interchangeable without too much difficulty. The look and feel of the GUI is almost the same with all systems.

At the current stage, the wrapper is just implemented so that it runs on Unix- and Linux-based architectures. Therefore the portability is a major concern. Most of the HPC environments are based on Unix and Linux systems. It is most likely that in the future, portability to Windows or Macintosh systems will not be very important. Still, using, for example, Qt as the coding language for the GUI, it could simply be transferred one day to other architectures and operating systems that are supported by the Qt toolkit.

7.5.4 Web Interface for the Wrapper

Generally well-equipped HPC centers are not found "on every corner" nor is it normal to have one's own number cruncher sitting in the basement. Rather, they are either installed at big universal or global companies or are maintained by the state or serve at a federal facility.

In Germany, for example, there are 4 official federal HPC centers, which are located at Jülich, Munich, Stuttgart and Berlin. There are also some larger, similar computing centers often owned, for example, by enterprises working in the automotive engineering sector, the oil- and gas industries and in the meteorological services (see also the motivation chapter at the very beginning of this thesis, starting at page 1.)

Often such institutions are kept and led like a fortification, especially when highly delicate data is processed there. Sometimes users have a difficult time entering such areas and are confronted with a high level of security systems and measures of precaution; even reaching them can present problems, since they may be quite some distance away. Depending on the position one holds, visiting such a center may conflict with compulsory presence at the company or institution where he or she is employed.

For such cases the existence of a so-called *web interface* would be advisable as a future expansion and improvement. Actually, it resembles a GUI, as described previously in subsection 7.5.3, on page 230. The only difference between a real GUI and a web interface is that the latter need not necessarily run on the same computer, platform or computing environment as the GUI normally does.

Let us assume that such a web interface exists already and that the user is not able, for what reasons soever, to leave his or her current workplace, but has to take care of the preprocessing process that is installed and running on a remote HPC architecture.

To establish something like a "connection" to the computing environment somewhere further away, the user would type in something similar to a web address (a URL) that is also used when working with the Internet when surfing the web using a web browser. Depending on how sophisticated the whole visionary web interface is, everything from a simple starting page in plain HTML up to animated pages, etc. would be possible.

At some point it is necessary for the user to provide the important information for the wrapper call. Generally, something like a form familiar to most readers from standard websites that provide an online shop, for instance, will be displayed. Here, exactly as with the GUI, the user will select the appropriate radio buttons, check boxes or fill in additional text fields with his or her information as necessary. The elements in such a form do not differ in their handling from those that were already described for the GUI and its usage.

When the user has provided all the information that is needed to start the wrapper on the remote computer, he or she will have to press a "send" or "submit" button, and the information that was displayed in a graphical way on the web page is now translated in a form that can be submitted over the Internet to the remote system. There, another application which is listening for such requests receives and takes over the data that was submitted by the user. This auxiliary application now initiates the call of the wrapper, exactly as the GUI did, and provides all the switches and information desired by the user, just as if the wrapper had been called using the commandline interface.

Upon termination of the preprocessing, the final results are gathered by the auxiliary program and again, depending on how sophisticated the web interface is laid out, sent back to the user in a certain way. The latter is informed that the wrapper execution was successful and will receive his or her results, which can either be submitted over the Internet or stored temporarily on the remote system until they are downloaded.

The realization of such a web interface can range from simple HTML forms available since the very beginnings of the Internet, to the more advanced XHTML that conforms to the Web 2.0 standard,

up to a version that already employs a popular script language called *Javascript*. Depending on the further usage of the results and how sophisticatedly the whole system is set up, a quite modern but very elaborate and complex system using AJAX might be employed. AJAX is quite a new approach in which one tries to avoid always reloading whole web pages. Instead just the "new parts" are redisplayed while the "old data" stays untouched if possible. It uses Javascript or HTML, XML and is based on HTTP, which has been well established for a long time in the Internet world.

On the remote site the auxiliary application generally is implemented in one of the three popular programming languages: Python, Perl or PHP. All of them have been widely used and tested. Which of these different languages – or perhaps even a totally different one – would be best suited depends on many different factors and cannot be determined on the fly. Sometimes such a decision might be based on the personal gusto of the programmer in charge or the underlying company policy or even such aspects as security reasons.

It is impossible to offer a comprehensive discussion here of the benefits and disadvantages of each of the forms such a web interface might take. The development in this sector of computer sciences is proceeding at an extremely rapid pace these days and many decisions depend strongly on the situation and the prerequisites and circumstances of such a project. Even the literature in this field is being updated faster than in many other areas so that is almost impossible to select a specific book to read as a good starting point. For this reason, the author recommends using the Internet itself in this special case for doing an extensive search for possible introductory or more advanced texts on the various topics connected to the concept of a web interface.

7.5.5 Comfortable Installation on Various Unix/Linux-Based Platforms

Already in the early stages of the preprocessing process it quickly became evident that the average user who is going to use the wrapper software later on is used to a Windows or similar operating system rather than to a Unix/Linux derivate. Most of them are familiar neither with the concept of a commandline nor with operating software using switches. For this reason the development not only of a GUI (see subsection 7.5.3 previously in this chapter, starting on page 230) but also of something similar to an installation package that can be maintained and used by one of the popular so-called *package mangement systems* would be a helpful improvement.

Such package management systems (PMS) are quite common nowadays in the Unix and Linux world. The usage of the term PMS varies slightly, but generally a PMS is a tool with which programs can be installed, maintained and deleted easily without too much knowledge about the underlying operating system. The two most common ones are the so-called *Red Hat Package Manager*, which works with `rpm` packages, and *Debian's* `dpgk` and `apt`, respectively. Both operate on Linux derivates. In the meantime, there are also PMSs for other platforms, like `cygwin` on Windows-based machines or `fink`, which is partly based on `apt`, for Macintosh computers. In the meantime even some GUIs exist like, *Synaptic* or *Aptitude*, which both operate on Debian-based Linux derivates.

When using such PMSs, generally spoken, the software that is to be installed, updated or maintained is "wrapped" in such a package and the user just has to find and take the appropriate package, feed it into the PMS. Thus he or she can handle the software and the necessary actions that have to be performed with it without much knowledge about the underlying operating system. The PMS takes care of all the obligatory steps for the installation or update process, etc. The individual PMS and its handling varies slightly but the basic principles are all the same: simplifications of

the software handling and maintainance process on the rather complex underlying Unix or Linux operating systems.

As the current and future user of the wrapper generally proves not to be too familiar with Unix or Linux systems on which the wrapper software most likely will be operated in the future, an appealing idea would be to build such an installation package out of all the necessary wrapper files. The user could then simply take such a package in which the wrapper code is hidden, and use the appropriate PMS to easily overcome the potential difficulties during the installation process, for example. Software updates could be performed in the same way.

7.5.6 Construction of Classical Unix/Linux Man Pages

In the current version of the wrapper software the man pages are called by using the commandline option `--help` or its short counterpart `-s`, as described in detail in sections 6.15 (see page 189ff) and 6.15.1 (see page 191ff) of chapter 6, respectively. This help function is integrated in the wrapper code itself.

The standard, classical man pages on Unix and Linux platforms are generated using a program named `troff`. The origins of `troff` date back to the middle of the 60s of the last century and have undergone various changes and enhancements. Nowadays, `troff` is a typesetting program in which the text to be displayed or printed is fed in and then produces a "nice-looking" output. It can be controlled by special commands, and various extensions are available. One of them is a macro with which the Unix and Linux man pages are created; this results in identical help texts, no matter which derivate of operating system one is using.

In a future version the help could be decoupled from the main program and set up in a separate text file which is then piped through `troff`. This way a call of the wrapper itself is unnecessary; the unexperienced user could simply use the standard command under Unix and Linux to get initial help or further instructions. Such a command would then look as follows:

```
==> man domdec
```

The output would then display – as is usual with such Unix or Linux man pages – first the general usage of the wrapper command, as it is listed currently at the bottom of the help text that is printed on the screen when using the `--help` or `-s` switch (see also bottom of the example on page 189). This would be followed by a short description of the function of the wrapper. The options would be described in further detail after this general information and at the end of the man page, either examples or special remarks would be given.

7.5.7 Automatic Coupling of the Wrapper

At the current stage the wrapper just runs as a stand-alone version that is decoupled from the proper parallelization process itself. The advantages and drawbacks of this situation have already be discussed in section 5.6 in chapter 5, starting on page 116.

Both types of wrapper, the decoupled version as well as the version integrated into the parallelization process, do have their benefits but also severe disadvantages. Which would be the best choice, stand-alone or coupled, strongly depends on the situation one is working in.

The goal of a future version of the wrapper program is to change the C code in such a way that the wrapper could still be used as an independent program as it is used today but also, if necessary, could run as a bigger initial routine of the parallelization program. In the latter case, the fact that the wrapper would be actively working as part of the parallel program would need to be controlled by using a certain commandline option or by the addition of a program parameter with which the wrapper could be used just like a function call. This is similar to how Jostle and Metis are currently called and deployed within the wrapper. Both also exist in stand-alone versions as well.

With the option of choosing between the two types of wrapper, the user could easily decide, in dependence of the given situation and framework, what version would be best suited to the case in question. Thus, the naturally existing drawbacks of each of the two types of wrapper software could be reduced to a minimum whilst profiting from their advantages.

Appendix A – Acronyms

A

- **AJAX** – **A**synchronous **J**avascript **a**nd **X**ML
- **ALU** – **A**rithmetic **L**ogical **U**nit
- **AMP** – **A**symmetrical **M**ulti**p**rocessor
- **AMR** – **A**daptive **M**esh**R**efinement
- **API** – **A**pplication **P**rogramming **I**nterface
- **ASCII** – **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange

B

- **Bit** – **Bi**ndary Digi**t** or **B**ndary Dig**it**
- **BON** – **B**ig **O**-**N**otation

C

- **CPU** – **C**entral **P**rocessing **U**nit
- **CM-1** – **C**onnection **M**achine-1
- **CM-2** – **C**onnection **M**achine-2

D

- **DAA** – **D**ynamically **A**llocated **A**rray
- **DM** – **D**istributed **M**emory
- **DNAPL** – **D**dense **N**on-**A**queous **P**hase **L**iquid
- **DSM** – **D**istributed **S**hared **M**emory
- **DVD** – **D**igital **V**ersatile **D**isc (*not* digital video disc as often claimed)

E

- **ES** – **E**arth **S**imulator

F

- **FE** – **F**inite **E**lements
- **Flops** – **Fl**oating Point **Op**eration**s**
- **FPU** – **F**loating **P**oint **U**nit
- **FTP** – **F**ile **T**ransfer **P**rotocol

G

- **GWh** – **G**iga **W**att **h**ours
- **GS/RF** – **G**eo**S**ys/**R**ock**F**low
- **GUI** – **G**raphical **U**ser **I**nterface

H

- **HLRS** – **H**öchst-**L**eistungs **R**echenzentrum **S**tuttgart
- **HPC** – **H**igh-**P**erformance **C**omputing
- **HTML** – **H**yper **T**ext **M**arkup **L**anguage
- **HTTP** – **H**yper **T**ext **T**ransfer **P**rotocol

I

- **IEC** – **I**nternational **E**lectrotechnical **C**ommission
- **I/O** – **I**nput **O**utput

L

- **LCD** – **L**iquid **C**rystal **D**isplay
- **LNAPL** – **L**ight **N**on-**A**queous **P**hase **L**iquid

M

- **MIMD** – **M**ultiple **I**nstruction **M**ultiple **D**ata
- **MISD** – **M**ultiple **I**nstruction **S**ingle **D**ata
- **MMU** – **M**emory **M**anaging **U**nit
- **MP-1** – **M**assive **P**arallel-1
- **MP-2** – **M**assive **P**arallel-2
- **MPI** – **M**essage **P**assing **I**nterface
- **MPS** – **M**ulti**p**rocessor **S**ystem
- **MSB** – **M**ulti **S**pectral **B**isection

N

- **NAPL – N**on-**A**queous **P**hase **L**iquid
- **NASA – N**ational **A**eronautics and **S**pace **A**dministration
- **NP – N**on-**P**olynomial
- **NPC – N**on-**P**olynomial-**C**omplete
- **NUMA – N**on-**u**niform **Memory Access**

O

- **OpenMP – Open M**ulti **P**rocessing

P

- **PC – P**ersonal **C**omputer
- **PDE – P**artial **D**ifferential **E**quations
- **PMS – P**ackage **M**anagement **S**ystem

R

- **RAM – R**andom **A**ccess **M**emory
- **RF – R**ock**F**low

S

- **SETI – S**earch for **E**xtra**t**errestial **I**ntelligence
- **SIMD – S**ingle **I**nstruction **M**ultiple **D**ata
- **SISD – S**ingle **I**nstruction **S**ingle **D**ata
- **SM – S**hared **M**emory
- **SMP – S**ymmetrical **M**ulti**p**rocessor

T

- **THMC – T**hermal-**H**ydrological-**M**echanical-**C**hemical

U

- **UMA – U**niform **M**emory **A**ccess
- **URL – U**niform **R**esource **L**ocator

V

- **VLIW – V**ery **L**arge **I**nstruction **W**ord

W

- **WWW – W**orld **W**ide **W**eb

X

- **XHTML –** e**X**tensible **H**yper **T**ext **M**arkup **L**anguage

Appendix B
Different Output Examples of the Wrapper Program after Preprocessing

This appendix consists of three parts. In the first, small exemplary output files will be shown to demonstrate the structure of the produced preprocessing results which are necessary for the model processing at a later stage of the parallelization process. In the second, typical output scenarios of the statistic tool will be shown with an automatically created overview of major parameters and the time consumption of different tasks during the preprocessing of the input data. The last one contains an optional output file which simply shows the direct mapping of the original input elements to the subdomains after the domain decomposition tools have been called.

For a more detailed description of the various types of output files produced by the wrapper program, the reader is referred to the corresponding sections in the main part of the thesis.

Appendix B.1 – Exemplary Output of the Wrapper Program

The first example output file is a very short one which had originally just one-dimensional elements, i.e., lines, in its input data file. The model consisted of 11 nodes, which in such a case results directly in 10 linear elements. (The input file can be found on page 257 in Appendix D, where some illustrative test data files are introduced.) The first file listing below is produced by using *Jostle* (see also section 5.4 of chapter 5, starting on page 107), dividing the input data into 5 domains.

```
#DOMAIN 0
$ELEMENTS 2
 8
 9
$NODES_INNER 2
 9
 10
$NODES_BORDER 1
 8
#DOMAIN 1
$ELEMENTS 2
 2
 3
$NODES_INNER 1
 3
$NODES_BORDER 2
```

```
 2
 4
#DOMAIN 2
$ELEMENTS 2
 0
 1
$NODES_INNER 2
 0
 1
$NODES_BORDER 1
 2
#DOMAIN 3
$ELEMENTS 2
 4
 5
$NODES_INNER 1
 5
$NODES_BORDER 2
 4
 6
#DOMAIN 4
$ELEMENTS 2
 6
 7
$NODES_INNER 1
 7
$NODES_BORDER 2
 6
 8
#STOP
```

The following output results from the same input file, again using Jostle, but this time a subdivision of just 3 domains has been performed.

```
#DOMAIN 0
$ELEMENTS 3
 0
 1
 2
$NODES_INNER 3
 0
 1
 2
$NODES_BORDER 1
 3
#DOMAIN 1
$ELEMENTS 3
 7
 8
 9
$NODES_INNER 3
 8
 9
 10
$NODES_BORDER 1
 7
#DOMAIN 2
$ELEMENTS 4
```

```
 3
 4
 5
 6
$NODES_INNER 3
 4
 5
 6
$NODES_BORDER 2
 3
 7
#STOP
```

The third example with the same input data set was produced by using *Metis* (see also section 5.5 of chapter 5, starting on page 112), but again with 3 subdomains. Small changes can be recognized in comparison to the result produced by *Jostle* (above) which result from the different algorithms implemented in these two domain decomposition tools.

```
#DOMAIN 0
$ELEMENTS 3
 0
 1
 2
$NODES_INNER 3
 0
 1
 2
$NODES_BORDER 1
 3
#DOMAIN 1
$ELEMENTS 3
 3
 4
 5
$NODES_INNER 2
 4
 5
$NODES_BORDER 2
 3
 6
#DOMAIN 2
$ELEMENTS 4
 6
 7
 8
 9
$NODES_INNER 4
 7
 8
 9
 10
$NODES_BORDER 1
 6
#STOP
```

The next example output file results from the input data set that can be seen on top of page 257 in Appendix D. This is a purely artificially manufactured input data set which was just created for

testing purposes. It consists of a mixture of geometric shapes for the elements but it cannot be used for any reasonable output. Nevertheless the file clearly demonstrates that one can use input data sets that do not contain uniformly geometric elements to process them with the wrapper program. The tool applied for processing this input data set was *Jostle* with a domain number of 4.

```
#DOMAIN 0
$ELEMENTS 2
 0
 1
$NODES_INNER 2
 10
 11
$NODES_BORDER 3
 8
 9
 12
#DOMAIN 1
$ELEMENTS 2
 2
 4
$NODES_INNER 0
$NODES_BORDER 5
 5
 7
 8
 9
 12
#DOMAIN 2
$ELEMENTS 2
 3
 6
$NODES_INNER 6
 0
 1
 2
 6
 13
 14
$NODES_BORDER 4
 3
 5
 7
 8
#DOMAIN 3
$ELEMENTS 1
 5
$NODES_INNER 1
 4
$NODES_BORDER 3
 3
 5
 9
#STOP
```

The last example input data set consists of 20 nodes which form 24 triangular elements. The original data can be seen on page 258 in Appendix D. The first output displayed below was produced using *Jostle* and partitioning the data set into 4 subdomains.

```
#DOMAIN 0
$ELEMENTS 6
 0
 1
 2
 3
 8
 9
$NODES_INNER 3
 0
 1
 5
$NODES_BORDER 5
 2
 6
 7
 10
 11
#DOMAIN 1
$ELEMENTS 6
 10
 11
 13
 21
 22
 23
$NODES_INNER 1
 19
$NODES_BORDER 9
 6
 7
 8
 11
 12
 13
 14
 17
 18
#DOMAIN 2
$ELEMENTS 6
 12
 16
 17
 18
 19
 20
$NODES_INNER 2
 15
 16
$NODES_BORDER 8
 7
 8
 10
 11
 12
 13
 17
 18
#DOMAIN 3
```

```
$ELEMENTS 6
 4
 5
 6
 7
 14
 15
$NODES_INNER 3
 3
 4
 9
$NODES_BORDER 5
 2
 7
 8
 13
 14
#STOP
```

The result of partitioning the same input data set into again 4 subdomains, but now using *Metis* instead of *Jostle*, is shown below. The output files vary slightly.

```
#DOMAIN 0
$ELEMENTS 6
 13
 15
 20
 21
 22
 23
$NODES_INNER 3
 13
 18
 19
$NODES_BORDER 4
 8
 12
 14
 17
#DOMAIN 1
$ELEMENTS 6
 4
 5
 6
 7
 12
 14
$NODES_INNER 3
 3
 4
 9
$NODES_BORDER 5
 2
 7
 8
 12
 14
#DOMAIN 2
```

```
$ELEMENTS 6
 0
 1
 2
 3
 8
 10
$NODES_INNER 3
 0
 1
 5
$NODES_BORDER 5
 2
 6
 7
 10
 12
#DOMAIN 3
$ELEMENTS 6
 9
 11
 16
 17
 18
 19
$NODES_INNER 3
 11
 15
 16
$NODES_BORDER 4
 6
 10
 12
 17
#STOP
```

Appendix B.2 – Examples of Statistic Output of the Wrapper

The first example of a statistic output file was produced by processing the input test data set shown at the beginning of this appendix, which simply consisted of linear elements. For the output file shown there, the following corresponding statistic file resulted when calculating a subdivision of 3 partitions using *Jostle*.

```
JOSTLE DOMAIN DECOMPOSITION RESULTS:
====================================

Topology is uniform (P = 3)

Graph:  10 nodes, 9 edges
        node weight = 10
        edge weight = 9

FINAL RESULTS:
Balance: 1.000  Largest: 2 (4)  Smallest: 0 (3)
```

```
Cut edges: 2

Storing jostle statistics ...

Jostle cut edges weight = 2
Jostle balance ratio = 1.000000
Jostle run time = 0.000000
Jostle memory usage (in bytes) = 3936


RESULTS OF THE TIMING ROUTINES AND STATISTICS:
==============================================
o Filename of input file: testdaten_11knoten_line_01.rfi
o Total of nodes: 11
o Total of elements: 10
o Total of domains: 3
o Usage of weights: OFF

(The following timing results are all given in millisecs!)

o Time between program start and processing commandline parameters: 0.248047
o Time for opening all files and reading in input file: 0.135986
o Time for processing input data and build adjacency graph: 0.0151367
o Time for preparing the data structures/initialization of domain
                                          decomposition tool: 0.00488281
o Time for calling domain decomposition tool "Jostle": 0.389893
o Time for processing data of domain decomposition tool: 0.0942383
o Time for checking result of correctness: 0.0119629
o Time for printing output file: 0.119873

==> Time difference from start of program to the end: 1.02002
```

For the same input data set the statistic output file looks as follows when *Metis* is used instead of *Jostle*:

```
RESULTS OF THE TIMING ROUTINES AND STATISTICS:
==============================================
o Filename of input file: testdaten_11knoten_line_01.rfi
o Total of nodes: 11
o Total of elements: 10
o Total of domains: 3
o Usage of weights: OFF

METIS STATISTICS:
-----------------
o Size of biggest partition in elements: 4
o Size of smallest partition in elements: 3
o Number of cut edges by metis: 2
o Balance factor of metis: 1.200000


(The following timing results are all given in millisecs!)

o Time between program start and processing commandline parameters: 0.245117
o Time for opening all files and reading in input file: 0.124023
o Time for processing input data and build adjacency graph: 0.0148926
o Time for preparing the data structures/initialization of domain
                                          decomposition tool: 0.00488281
```

```
o Time for calling domain decomposition tool "Metis": 0.0910645
o Time for processing data of domain decomposition tool: 0.100098
o Time for checking result of correctness: 0.0119629
o Time for printing output file: 0.105957

==> Time difference from start of program to the end: 0.697998
```

The statistic output for the purely artificial test data input set which was given as the next exemplary output file and uses a mixture of geometrically shaped elements looks as follows when *Jostle* is used and 4 subdomains are to be created:

```
JOSTLE DOMAIN DECOMPOSITION RESULTS:
====================================

Topology is uniform (P = 4)

Graph:  7 nodes, 16 edges
        node weight = 7
        edge weight = 16

FINAL RESULTS:
Balance: 1.000  Largest: 0 (2)  Smallest: 3 (1)
Cut edges: 13

Storing jostle statistics ...

Jostle cut edges weight = 13
Jostle balance ratio = 1.000000
Jostle run time = 0.000000
Jostle memory usage (in bytes) = 4240


RESULTS OF THE TIMING ROUTINES AND STATISTICS:
==============================================
o Filename of input file: testdaten_15knoten_mixed_fake_01.rfi
o Total of nodes: 15
o Total of elements: 7
o Total of domains: 4
o Usage of weights: OFF

(The following timing results are all given in millisecs!)

o Time between program start and processing commandline parameters: 19.9919
o Time for opening all files and reading in input file: 41.292
o Time for processing input data and build adjacency graph: 0.026123
o Time for preparing the data structures/initialization of domain
                                        decomposition tool: 0.00488281
o Time for calling domain decomposition tool "Jostle": 29.7839
o Time for processing data of domain decomposition tool: 0.123047
o Time for printing output file: 47.73

==> Time difference from start of program to the end: 138.952
```

When using the same input test data set but just switching to *Metis* instead of Jostle, the resulting statistic output file is as follows:

```
RESULTS OF THE TIMING ROUTINES AND STATISTICS:
==============================================
o Filename of input file: testdaten_15knoten_mixed_fake_01.rfi
o Total of nodes: 15
o Total of elements: 7
o Total of domains: 4
o Usage of weights: OFF


METIS STATISTICS:
-----------------
o Size of biggest partition in elements: 2
o Size of smallest partition in elements: 1
o Number of cut edges by metis: 13
o Balance factor of metis: 1.142857


(The following timing results are all given in millisecs!)

o Time between program start and processing commandline parameters: 0.25293
o Time for opening all files and reading in input file: 0.123047
o Time for processing input data and build adjacency graph: 0.0429688
o Time for preparing the data structures/initialization of domain
                                          decomposition tool: 0.00512695
o Time for calling domain decomposition tool "Metis": 0.124023
o Time for processing data of domain decomposition tool: 0.11499
o Time for printing output file: 21.9978

==> Time difference from start of program to the end: 22.6609
```

The following example of a statistic output file results from the processing of the test data set which consists uniformally of 24 triangular elements. In this example it was subdivided by *Jostle* into 4 domains.

```
JOSTLE DOMAIN DECOMPOSITION RESULTS:
====================================

Topology is uniform (P = 4)

Graph:  24 nodes, 110 edges
        node weight = 24
        edge weight = 110

FINAL RESULTS:
Balance: 1.000  Largest: 0 (6)  Smallest: 0 (6)
Cut edges: 59

Storing jostle statistics ...

Jostle cut edges weight = 59
Jostle balance ratio = 1.000000
Jostle run time = 0.000000
Jostle memory usage (in bytes) = 8512


RESULTS OF THE TIMING ROUTINES AND STATISTICS:
==============================================
o Filename of input file: testdaten_20knoten_tri_01.rfi
```

```
o Total of nodes: 20
o Total of elements: 24
o Total of domains: 4
o Usage of weights: OFF

(The following timing results are all given in millisecs!)

o Time between program start and processing commandline parameters: 0.253906
o Time for opening all files and reading in input file: 0.463135
o Time for processing input data and build adjacency graph: 0.0539551
o Time for preparing the data structures/initialization of domain
                                        decomposition tool: 0.0090332
o Time for calling domain decomposition tool "Jostle": 29.937
o Time for processing data of domain decomposition tool: 0.135986
o Time for printing output file: 0.216064

==> Time difference from start of program to the end: 31.0691
```

In the last example of a statistic output file, the same input data set as above was used, but this time it was subdivided by *Metis* instead of *Jostle*, again in 4 subdomains.

```
RESULTS OF THE TIMING ROUTINES AND STATISTICS:
==============================================
o Filename of input file: testdaten_20knoten_tri_01.rfi
o Total of nodes: 20
o Total of elements: 24
o Total of domains: 4
o Usage of weights: OFF


METIS STATISTICS:
-----------------
o Size of biggest partition in elements: 6
o Size of smallest partition in elements: 6
o Number of cut edges by metis: 54
o Balance factor of metis: 1.000000

(The following timing results are all given in millisecs!)

o Time between program start and processing commandline parameters: 0.264893
o Time for opening all files and reading in input file: 0.167969
o Time for processing input data and build adjacency graph: 0.0539551
o Time for preparing the data structures/initialization of domain
                                        decomposition tool: 0.0100098
o Time for calling domain decomposition tool "Metis": 0.675049
o Time for processing data of domain decomposition tool: 0.13208
o Time for printing output file: 23.012

==> Time difference from start of program to the end: 24.3159
```

Appendix B.3 – Examples of Mapping Files of the Wrapper

The last part of this appendix shows an example of an optional output file in ASCII format which displays the direct mapping between the elements of the original input data set and its subdomain

it was placed in after the domain decomposition tools have been called. This output file is not produced automatically. It rather has to be created manually by the user by explicitly setting the `--output`-switch when calling the wrapper, as explained in subsection 6.15.9 of chapter 6 on page 195.

```
Amount of Elements: 7   Amount of Partitions: 4

   Element number         Jostle partition
-----------------------------------------
        0                          0
        1                          0
        2                          1
        3                          2
        4                          1
        5                          3
        6                          2
```

The example above demonstrates the structure of this mapping file. The number of elements in the input file is given as well as the desired number of subdomains specified either by the user or given over a default value when no number of partitions has been set by the user (see also subsection 6.15.3 of chapter 6 on page 193). It is also obvious which partitioning tool was used to slice the original model data set. The following file resulted from the same input file and similar parameters; just a different partitioning tool was chosen.

```
Amount of Elements: 7   Amount of Partitions: 4

   Element number         Metis partition
-----------------------------------------
        0                          1
        1                          0
        2                          1
        3                          2
        4                          3
        5                          3
        6                          2
```

Appendix C –
Help Text of Domdec V3.2 and V3.3

```
COMMAND LINE OPTIONS OF DOMDEC_V03_2:
=====================================

-b, --debug     Switch to debug mode and print debug messages.
                Default is set to "debug = OFF".

-d, --ddcfile   Expects a filename of a ddc file as later output
                file; if no ddc file is given, an output file is
                created, using the input filename plus additional
                suffixes. Maximum filename length: 255 characters!

-h, --help      Prints out this help text and exits; see correct
                usage below!

-i, --input     Expects an input filename of a rfi file with the
                mesh data, same as the command line switch -r or
                --rfifile! Maximum filename length: 255 characters!

-j, --jostle    Jostle will be used as domain decomposition tool.
                This is the default if no domain decomposition
                tool is chosen by the user.

-m, --metis     Metis will be used as domain decomposition tool.
                If no domain decomposition tool was chosen,
                Jostle is taken by default!

-n, --noweight  When using this option all input values for the
                domain decomposition tool are treated with the
                same weight. This is also the default if no user
                input is specified.

-o, --output    An additional output file can be given as an
                intermediate output file which holds the
                element-to-partition relation. Maximum filename
                length: 255 characters!
```

```
-c, --parmetis  ParMetis will be used as domain decomposition
                tool. If no domain decomposition tool was chosen,
                Jostle is taken by default! (ParMetis is not yet
                implemented.)

-p, --partitions
                Takes an integer value for the amount of parti-
                tions which should be created with the domain
                decomposition tool. If no integer value was
                specified by the user, a number of 12 is taken
                as a default value!

-r, --rfifile   Expects a filename of a rfi file as input.
                This is the same as the command line switch -i or
                --input. Maximum filename length: 255 characters!

-s, --statistic An additional statistic output file is created.
                It is given the same name as the ddc file, but
                with the extension "stt" instead of "ddc", plus
                the tool name of the domain decomposition tool.
                Default value for this switch is "statistic = off".

-t, --test      Additional tests of correctness of the output of
                the domain decomposition tools are performed.
                (Very time consuming!) Default value is set to
                "test = off".

-v, --verbose   Switches to verbose mode; default is "verbose = off".

-w, --weight    With this option, the input for the domain
                decomposition tool uses different weights
                for the input values. Default value is option "-n";
                input with no weights!


USAGE OF DOMDEC_V03_2:
======================

domdec_v03_2 [-v|--verbose] [(-p|--partitions) value] [-b|--debug]
             [-j|--jostle|-m|--metis|-c|--parmetis] [-s|--statistic]
             [(-o|--output) filename] [(-d|--ddcfile) filename]
             [-t|--test] [-w|--weight|-n|--noweight] filename

domdec_v03_2 [-b|--debug] [(-p|--partitions) value] [-v|--verbose]
             [-j|--jostle|-m|--metis|-c|--parmetis] [-t|--test]
             (-i|--input|-r|--rfifile) filename [-s|--statistic]
             [-w|--weight|-n|--noweight] [(-d|--ddcfile) filename]
             [(-o|--output) filename]

domdec_v03_2 (-h|--help)
```

```
COMMAND LINE OPTIONS OF DOMDEC_V03_3:
=====================================

-b, --debug      Switch to debug mode and print debug messages.
                 Default is set to "debug = OFF".

-d, --ddcfile    Expects a filename of a ddc file which will be used
                 as the later output file. If no ddc file is given,
                 an output file will be created automatically, using
                 the input filename plus additional suffixes, depen-
                 ding on whether or not the command line option "-l"
                 was given. Maximum filename length: 500 characters!

-h, --help       Prints out this help text and exits; see correct
                 usage below!

-i, --input      Expects an input filename with the mesh data.
                 Either the old format with 'file.rfi' or the
                 new format with 'file.msh' can be read. Maximum
                 filename length: 500 characters!

-j, --jostle     Jostle will be used as domain decomposition tool.
                 If no domain decomposition tool was chosen, Metis
                 is taken by default!

-l, --longname   Using this switch will result in a longer filename
                 for the ddc and stt files. Information like time
                 of creation, tool used, number of partitions, etc.
                 will then be added.

-m, --metis      Metis will be used as domain decomposition tool.
                 This is the default if no domain decomposition
                 tool is chosen by the user.

-n, --noweight   When using this option, all input values for the
                 domain decomposition tool are treated with the
                 same weight. This is also the default if no
                 weights are specified by the user.

-o, --output     An additional output file can be given as an
                 intermediate output file which holds the element
                 -to-partition relation. Maximum filename length:
                 500 characters!

-c, --parmetis   ParMetis will be used as domain decomposition
                 tool. If no domain decomposition tool was chosen,
                 Jostle is taken by default! (ParMetis is not yet
                 implemented.)
```

```
-p, --partitions       Takes an integer value for the number of
                partitions which should be created with the
                domain decomposition tool. If no integer value
                was specified by the user, the number 12 is
                taken as default value!

-s, --statistic        An additional statistic output file is
                created. It is given the same name as the
                ddc file, but with the extension "stt" instead
                of "ddc". If the command line option "-l" is
                used, additional information is added to the name.
                Default value for this switch is "statistic = OFF".

-t, --test      Additional tests of correctness of the output
                of the domain decomposition tools are performed.
                (Depending on the size of the input mesh this can
                be very time consuming!) Default value is set to
                "test = OFF".

-v, --verbose   Switches output to verbose mode; default is
                "verbose = OFF". (Recommendation is to use
                it only for debugging purposes!)

-w, --weight    With this option, the input for the domain
                decomposition tool uses different weights
                for the input values. Default value is option
                "-n", which is "weights = OFF"!


USAGE OF DOMDEC_V03_3:
======================

domdec_v03_3 [-v|--verbose] [(-p|--partitions) value] [-b|--debug]
             [-j|--jostle|-m|--metis|-c|--parmetis] [-s|--statistic]
             [(-o|--output) filename] [(-d|--ddcfile) filename]
             [-t|--test] [-l|--longname] [-w|--weight|-n|--noweight]
             filename

domdec_v03_3 [-b|--debug] [(-p|--partitions) value] [-v|--verbose]
             [-j|--jostle|-m|--metis|-c|--parmetis][-s|--statistic]
             [-w|--weight|-n|--noweight] [(-d|--ddcfile) filename]
             [(-o|--output) filename] [-t|--test] [-l|--longname]
             (-i|--input) filename

domdec_v03_3 (-h|--help)
```

Appendix D – Test Data Sets

The following small files are some small example files of data sets, which have been used throughout the development and thoroughly testing phase of the wrapper program. Most of them are real but small test models; just a few have been artificially created to check and improve the process of file recognition, element shape detection and other small but important aspects.

```
#0#0#13#1#901200000#9062#3805OK34##################################
0 15 7
0 54 90 34.000
1 494.0 55.0 353453.0
2 4 5 7
3 44.0004 34.333 -4554.34
4 54 88.004 -0.333
5 -1 -88 -444.44
6 99.2 -0.55 3.3
7 333.3 88 101
8 445 34 -84.3
9 -1 0 0
10 0 44 4.4
11 1 2 3
12 5 6 7
13 -3 -4 -5
14 88 -88 100
0 0 quad 8 9 10 11
1 0 tri  8 11 12
2 0 tri  7 8 12
3 0 quad 5 6 7 8
4 0 tri  5 8 9
5 0 quad 3 4 5 9
6 0 hex  0 1 2 3 5 6 13 14
```

```
#0#0#0#1#0.0#0#########################################################
0  11  10
0  0.00 0.0  0.0
1  0.1 0.0  0.0
2  0.2 0.0  0.0
3  0.3  0.0  0.0
4  0.4  0.0  0.0
5  0.5  0.0  0.0
```

```
6  0.6  0.0  0.0
7  0.7  0.0  0.0
8  0.8  0.0  0.0
9  0.9  0.0  0.0
10 1.0 0.0  0.0
0  0  line   0  1
1  0  line   1  2
2  0  line   2  3
3  0  line   3  4
4  0  line   4  5
5  0  line   5  6
6  0  line   6  7
7  0  line   7  8
8  0  line   8  9
9  0  line   9 10

 #0#0#0#1#0.0#0#3909OK#######
0   16   4
0     0.    0.    0.
1   100.    0.    0.
2   100.  100.    0.
3     0.  100.    0.
4     0.    0.  -10.
5   100.    0.  -80.
6   100.  100.  -80.
7     0.  100.  -10.
8     0.    0.  -90.
9   100.    0.  -90.
10  100.  100.  -90.
11    0.  100.  -90.
12    0.    0. -100.
13  100.    0. -100.
14  100.  100. -100.
15    0.  100. -100.
0 0 2 quad 4 5 6 7
1 1 3 hex  0 1 2 3 4 5 6 7
2 2 3 hex  4 5 6 7 8 9 10 11
3 3 3 hex  8 9 10 11 12 13 14 15


#0#0#0#1#0.0#0#####################################################
0  20  24
0  0.000000  0.000000  0.000000
1  1.000000  0.000000  0.000000
2  2.000000  0.000000  0.000000
3  3.000000  0.000000  0.000000
4  4.000000  0.000000  0.000000
5  0.000000  1.000000  0.000000
6  1.000000  1.000000  0.000000
```

```
7  2.000000  1.000000  0.000000
8  3.000000  1.000000  0.000000
9  4.000000  1.000000  0.000000
10  0.000000  2.000000  0.000000
11  1.000000  2.000000  0.000000
12  2.000000  2.000000  0.000000
13  3.000000  2.000000  0.000000
14  4.000000  2.000000  0.000000
15  0.000000  3.000000  0.000000
16  1.000000  3.000000  0.000000
17  2.000000  3.000000  0.000000
18  3.000000  3.000000  0.000000
19  4.000000  3.000000  0.000000
0   0  tri   0  6  5
1   0  tri   0  1  6
2   0  tri   1  2  6
3   0  tri   2  7  6
4   0  tri   2  3  8
5   0  tri   2  8  7
6   0  tri   3  4  8
7   0  tri   4  9  8
8   0  tri   5  6 10
9   0  tri   6 11 10
10  0  tri   6  7 12
11  0  tri   6 12 11
12  0  tri   7  8 12
13  0  tri   8 13 12
14  0  tri   8  9 14
15  0  tri   8 14 13
16  0  tri  10 11 16
17  0  tri  10 16 15
18  0  tri  11 12 16
19  0  tri  12 17 16
20  0  tri  12 13 18
21  0  tri  12 18 17
22  0  tri  13 14 18
23  0  tri  14 19 18


#0#0#0#1#0.0#0##################################################
0 22 10 quad
0 0 0 0
1 0.01 0 0
2 0.01 0.01 0
3 0 0.01 0
4 0.02 0 0
5 0.02 0.01 0
6 0.03 0 0
7 0.03 0.01 0
8 0.04 0 0
```

```
9 0.04 0.01 0
10 0.05 0 0
11 0.05 0.01 0
12 0.06 0 0
13 0.06 0.01 0
14 0.07 0 0
15 0.07 0.01 0
16 0.08 0 0
17 0.08 0.01 0
18 0.09 0 0
19 0.09 0.01 0
20 0.1 0 0
21 0.1 0.01 0
0 0 quad 0 1 2 3
1 0 quad 1 4 5 2
2 0 quad 4 6 7 5
3 0 quad 6 8 9 7
4 0 quad 8 10 11 9
5 0 quad 10 12 13 11
6 0 quad 12 14 15 13
7 0 quad 14 16 17 15
8 0 quad 16 18 19 17
9 0 quad 18 20 21 19


#0#0#0#1#0.0#0################################################
0     24      6
  0   0.0000   0.0000   0.0000
  1   0.3333   0.0000   0.0000
  2   0.6667   0.0000   0.0000
  3   1.0000   0.0000   0.0000
  4   0.0000   0.5000   0.0000
  5   0.3333   0.5000   0.0000
  6   0.6667   0.5000   0.0000
  7   1.0000   0.5000   0.0000
  8   0.0000   1.0000   0.0000
  9   0.3333   1.0000   0.0000
 10   0.6667   1.0000   0.0000
 11   1.0000   1.0000   0.0000
 12   0.0000   0.0000   1.0000
 13   0.3333   0.0000   1.0000
 14   0.6667   0.0000   1.0000
 15   1.0000   0.0000   1.0000
 16   0.0000   0.5000   1.0000
 17   0.3333   0.5000   1.0000
 18   0.6667   0.5000   1.0000
 19   1.0000   0.5000   1.0000
 20   0.0000   1.0000   1.0000
 21   0.3333   1.0000   1.0000
 22   0.6667   1.0000   1.0000
```

```
 23    1.0000    1.0000    1.0000
0 0 hex   12   13   17   16    0    1    5    4
1 0 hex   13   14   18   17    1    2    6    5
2 0 hex   14   15   19   18    2    3    7    6
3 0 hex   16   17   21   20    4    5    9    8
4 0 hex   17   18   22   21    5    6   10    9
5 0 hex   18   19   23   22    6    7   11   10
```

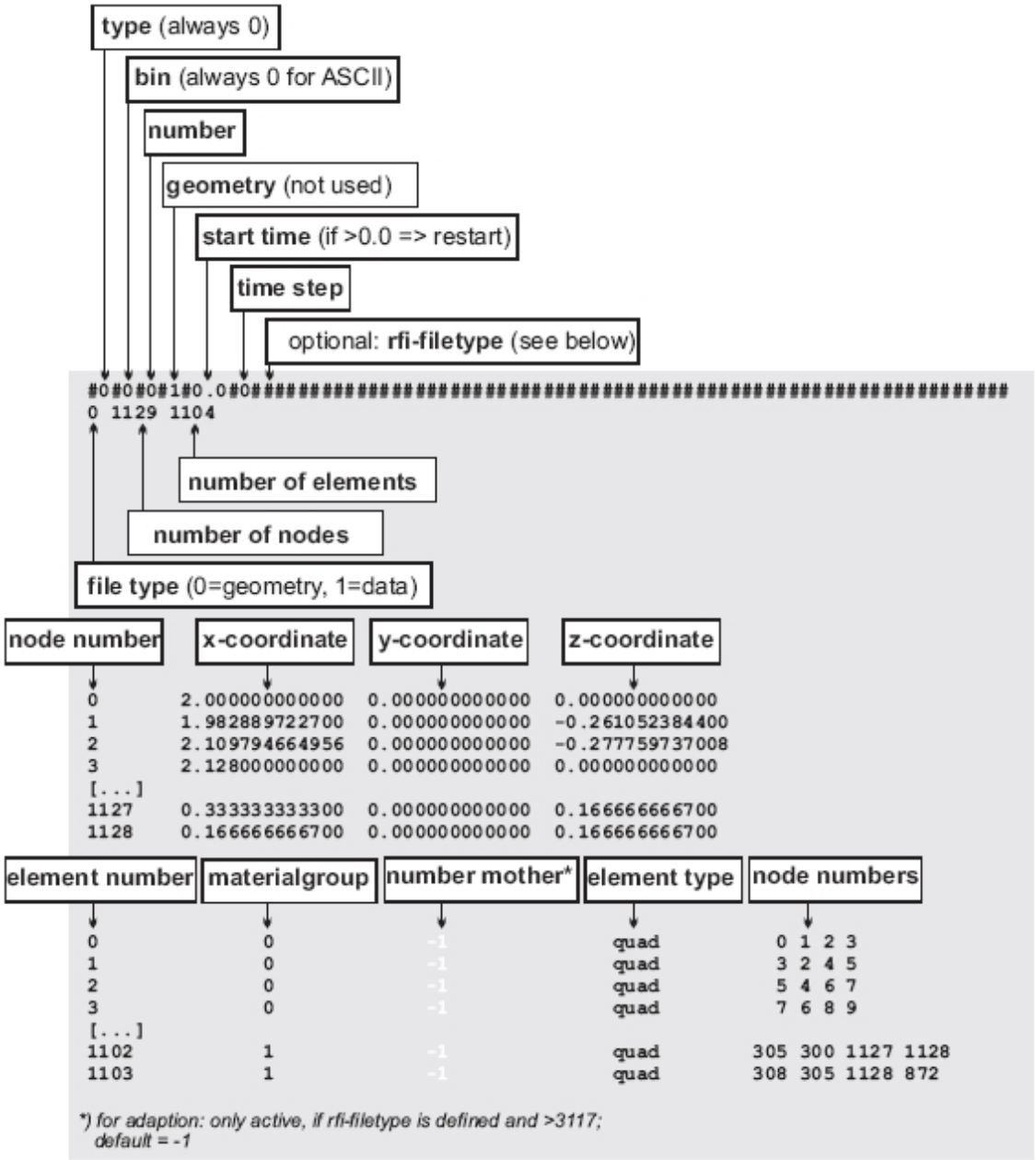


Figure 7.11: An example of the structure of one of the input data files that have been used

Bibliography

[Agarwala] T. Agarwala, J. Cocke. *High Performance of Reduced Instruction Set Processors*, IBM Tech. Rep. March 1987

[Allen] Randy Allen, Ken Kennedy. *Optimizing Compilers for Modern Architectures*, Academic Press, London. 2002

[Ben-Ari] M. Ben-Ari. *Grundlagen der Parallel-Programmierung*, Carl Hanser Verlag. 1985

[Bentley] Jon Bentley. *Programming Pearls*, Addison-Wesley, 2nd revised Edition. 2000

[Berkeley] University of California at Berkeley.
CPU Info Center, `http://bwrc.eecs.berkeley.edu/CIC/`,
Accessed May 26, 2006

[Beth] Thomas Beth. *Signale, Codes und Chiffren II*, Vorlesungsskript, University of Karlsruhe. SS 1987

[Bons] Paul D. Bons. *Process Modelling, Part One*, Lecture Notes, University of Tuebingen. 2007

[Bovet] Daniel Pierre Bovet, Plerlulgi Crescenzl. *Introduction of the Theory of Complexity*, Prentice-Hall. Nov. 1993

[Chan] Tony F. Chan, John R. Gilbert, Shang-Tua Teng. *Geometric spectral partitioning*, Xerox PARC Technical Report. 1994

[Chandra] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. *Parallel Programming in OpenMP*, Academic Press. 2001

[Chen] Cui Chen, Ali Sawarieh, Thomas Kalbacher, Martin Beinhorn, Wenqing Wang,Olaf Kolditz. *A GIS based 3D Hydrosystem Model Application to the Zarqa Ma'in - Jiza areas in Central Jordan*, Journal of Environmental Hydrology, Vol. 13, pages 1-13. 2005

[Cormen] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*, MIT Press, 2nd edition. 2001

[Darnell] Peter A. Darnell, Philip E. Margolis. *C – A Software Engineering Approach*, Springer-Verlag, 3rd revised Edition. 1996

[Debian] Official Debian Linux Homepage.
`http://http://www.debian.org/`, Accessed February 12, 2009

[Diefendorff] K. Diefendorff, M. Allen. *Organization of the Motorola 88110 Superscalar RISC Microprocessor*, IEEE Micro. April 1992, pages 40-63

[Drakos] Nikos Drakos. *Computer Architecture*, Computer Based Learning Unit, University of Leeds. 1994

[Du] Xiao-Xin Du, Hu Guoding. *Combinatorics, Computing and Complexity*, Springer-Verlag Gmbh, 2nd Edition. March 1990

[Duden] *Duden "Informatik" - Ein Sachlexikon für Studium und Praxis*, Dudenverlag, Mannheim. 1988

[Earth] Earth Simulator Homepage. `http://www.es.jamstec.go.jp/esc/eng/index.html`, Accessed July 06, 2006

[Edwards] C. Henry Edwards, David E. Penney. *Elementary differential equations with boundary value problems*, 5th Edition, Pearson Prentice Hall. 2004

[Fiduccia] M. Fiduccia, R. M. Mattheyses. *A linear time heuristic for improving network partitions*, Proceedings on the 19th IEEE Design Automation Conference, pp. 175 - 181. 1982

[Fiedler1] Miroslav Fiedler. *Algebraic connectivity of graphs*, Czechoslovak Math. Journal, Vol. 23, pp. 298 - 305. 1973

[Fielder2] Miroslav Fiedler. *A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory*, Czechoslovak Math. Journal, Vol. 25, pp. 619 - 633. 1975

[Flynn1] Michael J. Flynn. *Very High Speed Computing Systems*, Proceedings IEEE, 54, No. 12. 1966, pages 1901-1909

[Flynn2] Michael J. Flynn, Kevin W. Rudd. *Parallel Architectures*, ACM Computing Surveys 28(1). March 1996, pages 67-70

[Foster01] Ian Foster. *Designing and Building Parallel Programs*, Addison-Wesley. 1st Edition, 1995

[Foster02] Ian Foster, Carl Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, S. F., CA. 1999

[Foster03] Ian Foster, Jonathan Geisler, Bill Nickless, Warren Smith, Steven Tuecke. *Software infrastructure for the I-WAY metacomputing experiment*, Concurrency: Practice and Experience Volume 10, Issue 7. 1998, pages 567-581

[Fox] G. G. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, D. W. Walker. *Solving Problems on Concurrent Processors: Volume 1*, Prentice Hall. 1988

[Gilly] Daniel Gilly. *UNIX in a Nutshell - A Desktop Quick Reference for System V and Solaris 2.0*, O'Reilly. 2nd Edition, June 1998

[Giloi] Wolfgang K. Giloi. *Rechnerarchitektur*, Springer-Verlag GmbH. 2nd revised edition, Juni 1997

[Globus] Homepage of the Globus Alliance. `http://www.globus.org/`, Accessed July 19, 2006

[Gulbins] Jürgen Gulbins. *UNIX Version 7, bis System V.3* (the "worm book"), Springer Verlag. 3rd Edition, 1985

[Gupta] Anshul Gupta, George Karypis, Vipin Kumar. *Highly scalable parallel algorithms for sparse matrix factorization*, IEEE Transactions on Parallel and Distributed Computing. 1994

[Hendrickson] Bruce Hendrickson, Robert Leland. *An improved spectral graph partitioning algorithm for mapping parallel computations*, Technical Report SAND92-1460, Sandia National Laboratories. 1992

[Hu] Y. F. Hu, R. J. Blake, D. R. Emerson. *An optimal migration algorithm for dynamic load balancing*, Concurrency: Practice and Experience, Vol. 10(6), pp. 467 - 483. 1998

[Huckle] Thomas Huckle, Stefan Schneider. *Numerische Methoden: Eine Einführung für Informatiker, Naturwissenschaftler, Ingenieure und Mathematiker*, Springer-Verlag, 2nd Edition. 2006

[Huseynov] Javid Huseynov. *Distributed Shared Memory Home Pages*, `http://www.ics.uci.edu/ javid/dsm.html`, Accessed August 14th, 2006

[ISO 9899] American National Standards Institute. *ANSI Standard ISO/IEC 9899:1999 – Programming Language C (C99).* `http://www.ics.uci.edu/ javid/dsm.html`, 1999

[Istok] Jonathan Istok. *Groundwater modeling by the Finite Element Method*, American Geophysical Union. 2000

[Jájá] Joseph Jáá. *An Introduction to Parallel Algorithms*, Addison-Wesley. 1992

[Java] Official Java Homepage of Sun Microsystems. `http://java.sun.com/`, Accessed February 9, 2009

[Johnson] M. Johnson. *Superscalar Microprocessor Design*, Prentice Hall. 1991

[Jostle] Official Jostle Homepage. `http://staffweb.cms.gre.ac.uk/ c.walshaw/jostle/`, Accessed February 09, 2009

[Karypis1] George Karypis, Vipin Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, SIAM Journal on Scientific Computing. 1998

[Karypis2] George Karypis, Vipin Kumar. *Multilevel k-way partitioning scheme for irregular graphs*, Journal of Parallel and Distributed Computing, Vol 48(1), pp. 96-129. 1998

[Karypis3] George Karypis, Vipin Kumar. *Multilevel algorithms for multi-constraint graph partitioning*, Technical Report, Department of Computer Science, University of Minnesota, TR 98-019. 1998

[Kemmler] D. Kemmler, P. Adamidis, R. Rabenseifner, W. Wang, S. Bauer, O. Kolditz. *Solving Coupled Geoscience Problems on High Performance Computing Platforms*, Springer-Verlag Berlin, pages 1064-1071. 2005

[Kernighan] B. Kernighand, S. Lin. *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, Vol. 29, pp. 291 - 307. 1970

[Kinzel] Wolfgang Kinzel. *Programmierkurs für Naturwissenschaftler und Ingenieure - Schnelleinstieg in Linux, C, Java und Mathematica/Maple*, Addison-Wesley. 2001

[Knuth01] Donald E. Knuth. *The Art of Computer Programming - Volume 1: Fundamental Algorithms*, 2nd Printing, Addison-Wesley. 1969

[Knuth02] Donald E. Knuth. *The Art of Computer Programming - Volume 3: Sorting and Searching*, 3rd Edition, Addison-Wesley. 1998

[Kofler] Michael Kofler. *Linux - Installation, Konfiguration, Anwendung*, Addison-Wesley. 7th Edition, 2006

[Kohlmeier] M. Kohlmeier. *Coupling of thermal, hydraulic and mechanical processes for geotechnical simulations of partially saturated porous media*, Dissertation, Institut für Strömungsmechanik, Universität Hannover. 2006

[Kolditz] Olaf Kolditz. *Computational Methods in Environmental Fluid Mechanics*, Springer Verlag. 2002

[Kröhn] K.-P. Kröhn, J. Wollrath, W. Zielke. *Modelling Transport in Fractured Rock by the Finite Element Method*, Proceedings on IAHR International Symposium on Contaminant Transport in Groundwater, pages 291-297. April 4-6, 1989

[Lee] R. B. Lee. *Empirical Results on the Speed, Efficiency, Redundancy and Quality of Parallel Computations*, Proceedings on International Conference on Parallel Processing. 1980, pages 91-96

[Leland1] B. Hendrickson, R. Leland. *The Chaco User's Guide, Version 2.0*, Technical Report SAND95-2344 of Sandia National Laboratories, Albuquerque, NM. July 1995

[Leland2] B. Hendrickson, R. Leland. *An improved spectral load balancing method*, Proceedings on 6th SIAM Conference on Parallel Processing for Scientific Computing, pp. 953 - 961. 1993

[Leland3] B. Hendrickson, R. Leland. *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM Journal on Scientific Computing, Vol. 16. 1995

[Leland4] B. Hendrickson, R. Leland. *A multilevel algorithm for partitioning graphs*, Proceedings on Supercomputing'95, ACM. Dec. 1995

[Linden] Peter van der Linden. *Expert C Programming - Deep C Secrets*, SunSoft Press. 1994

[Linux] German homepage of Learning Linux. *Learning Linux – Linux lernen leicht gemacht*, `http://www.learninglinux.de/index.html`, Accessed September 19, 2006

[Maertin] Christian Märtin. *Rechnerarchitektur*, Hanser-Verlag Munich. 1994

[Markowski] Hartmut Markowski. *Einführung in die Informatik, Band 1B - Algebra für Informatiker: Gruppen*, Verlag Moderne Industrie. 1974

[McCarty] Bill McCarty. *Learning Debian GNU/Linux*, O'Reilly. 1999

[McGraw] James R. McGraw, Timothy S. Axelrod. *Exploiting multiprocessors: Issues and Options*, In Robert G. Babb II, (ed.) *Programming Parallel Processors*, pp. 7-25, Addison-Wesley. 1988

[Meissner] Udo F. Meissner, Andreas Maurial. *Die Methode der finiten Elemente: Eine Einführung in die Grundlagen*, Springer-Verlag, 2nd Edition. 2008

[Meister] Andreas Meister. *Numerik linearer Gleichungssysteme - Eine Einführung in moderne Verfahren*, Friedrich Vieweg Verlagsgesellschaft. 1999

[Metis] Metis Homepage. `http://www.cs.umn.edu/~metis`

[Miller] Gary L. Miller, Shang-Hua Teng, Stephen A. Vavasis. *A unified geometric approach to graph separators*, Proceedings of 31st Anual Symposium on Foundations of Computer Science, pp. 538-547. 1991

[MPI-1] M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, J. Dongarra. *MPI – The Complete Reference, Volume I, The MPI Core*, MIT Press, 2nd Edition. 1998

[MPI-2] W. Gropp, S. Huss-Ledermann, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir. *MPI – The Complete Reference, Volume I, The MPI Extensions*, MIT Press. 1998

[MPI-3] MPI Homepage. `http://www-unix.mcs.anl.gov/mpi/`, Accessed June 24, 2006

[Neale] Vicky Neale. *Modular Arithmetic*, `http://nrich.maths.org`, Accessed Dec 04, 2008

[OpenMP] Official OpenMP Forum Webpage. `http://www.openmp.org/`, Accessed June 24, 2006

[Ortmann] J"urgen Ortmann. *Einführung in die PC-Grundlagen*, Addison-Wesley. 2001

[Pacheco] Peter S. Pacheco. *Parallel Programming with MPI*, Morgan Kaufmann. 1st Edition, 1997

[ParMetis] Official Homepage of ParMetis. `http://glaros.dtc.umn.edu/gkhome/views/metis/`, Accessed February 19, 2009

[Patterson] David A. Patterson, John L. Hennessy. *Rechnerorganisation und -entwurf*, Elsevier GmbH. 3rd Edition, 2005

[PHREEQC] Official Homepage of PHREEQC. `http://wwwbrr.cr.usgs.gov/projects/GWC_coupled/phreeqc/`, Accessed February 20, 2009

[Ponnusamy] R. Ponnusamy, N. Mansour, A. Choudhary, G. C. Fox. *Graph contraction and physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers*, International Conference of Supercomputing. 1993

[Pothen] A. Pothen, H. Simon, K. Liou. *Partitioning sparse matrices with eigenvectors of graphs*, SIAM Journal on Matrix Analysis, Vol. 11, pp. 430 - 452. 1990

[Qt] Official Homepage of the Qt Cross-Platform Application Framework. `http://http://www.qtsoftware.com/`, Accessed February 09, 2009

[Quattrocchi] P. Quattrocchi, W. Heise. *Informations- und Codierungstheorie*, Springer Verlag. 1983

[Quinn] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill. 2004

[Ras] Supercomputer Software Department RAS. *VLSI Microprocessors – A Guide to High-Performance Microprocessor Resources*, `http://www.microprocessor.sscc.ru/`, Accessed May 26, 2006

[Rauber] Thomas Rauber, Gudula Rünger. *Parallele und verteilte Programmierung*, Springer Verlag. 2000

[Ritchie] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language*, Prentice Hall Software Series, 2nd Edition. 1988

[RPM] Official Homepage of the RPM Package Management System. `http://www.rpm.org/`, Accessed February 12, 2009

[Schmeh] Klaus Schmeh. *Kryptografie und Public-Key-Infrastrukturen im Internet*, dpunkt.verlag, 2nd Edition. 2001

[Sedgewick01] Robert Sedgewick. *Algorithmen*, Pearson Studium, 2nd Edition. 2002

[Sedgewick02] Robert Sedgewick. *Algorithmen in C++*, Pearson Studium, 3rd Edition. 2002

[Seti] Homepage of the SETI home project. `http://setiathome.ssl.berkeley.edu/`, Accessed July 20, 2006

[Siever] Ellen Siever. *Linux in a Nutshell – A Desktop Quick Reference*, O'Reilly, 2nd Edition. February 1999

[Šilc] J. Šilc, B. Robič, T. Ungerer. *Processor Architecture*, Springer. 1999

[Simon] H. D. Simon. *Partitioning of unstructured problems for parallel processing*, Conference Proceedings on Parallel Methods on Large Scale Structural Analysis and Physics Applications, Pergammon Pres. 1991

[Song] J. Song. *A partially asynchronous and iterative algorithm for distributed load balancing*, Parallel Computing, Vol. 20(6), pp. 853 - 868. 1994

[Stallings] William Stallings. *Sicherheit im Internet - Anwendungen und Standards*, Addison-Wesley. 2001

[Stroustrup] Bjarne Stroustrup. *The C++ Programming Language*, Addison-Wesley, 3rd revised Edition. 2004

[Suaris] P. Suaris, G. Kedem. *An algorithm for quadrisection and its application to standrad cell placement*, IEEE Transactions on Circuits and Systems, Vol. 35, pp. 294 - 303. 1988

[Synaptic] Official Hompepage of Synaptic. `http://www.nongnu.org/synaptic/`, Accessed February 12, 2009

[Top500] Official Top 500 Webpage. `http://www.top500.org/`, Accessed July 6, 2006

[Topping] B. H. V. Topping, A. I. Khan. *Parallel Finite Element Computations*, Saxe-Coburg Publications. 1996

[Ungerer01] Theo Ungerer, Winfried Görke, Detlev Schmid. *Rechnerstrukturen*, Manuskript zur Kernfachvorlesung "Rechnerstrukturen", 8. Auflage. 1997

[Ungerer02] Theo Ungerer. *Mikroprozessortechnik*, Thomson's Aktuelle Tutorien, Thomson Verlag. 1995

[Valgrind] Official Valgrind Webpage. `http://www.valgrind.org/`, Accessed February 06, 2009

[Walshsaw01] Chris Walshaw. *The serial JOSTLE library user guide, Version 3.0*, University of Greenwich Tech Report. July 8, 2002

[Walshsaw02] Chris Walshaw. *The JOSTLE executable user guide, Version 3.0*, University of Greenwich Tech Report. July 8, 2002

[Walshsaw03] Chris Walshaw. *The parallel JOSTLE library user guide, Version 3.0*, University of Greenwich Tech Report. July 8, 2002

[Walshsaw04] C. Walshaw, M. Cross. *Load-balancing for parallel adaptive unstructured meshes*, Proceedings on Numerical Grid Generation in Computational Field Simulations, pp. 781 - 790. 1998

[Walshsaw05] C. Walshaw, M. Cross. *Mesh partitioning: a multilevel balancing and refinement algorithm*, SIAM Journal of Scientific Computing, Vol. 22(1), pp. 63 - 80. 2000

[Walshsaw06] C. Walshaw, M. Cross. *Parallel Optimisation Algorithms for Multilevel Mesh Partitioning*, Parallel Computing, Vol. 26(12), pp. 1635 - 1660. 2000

[Walshsaw07] C. Walshaw, M. Cross. *Multilevel mesh partitioning for heterogeneous commuication networks*, Future Generation Computing Systems, Vol. 17(5), pp. 601 - 623. 2001

[Walshsaw08] C. Walshaw, M. Cross, R. Diekmann, F. Schlimbach. *Multilevel mesh partitioning for optimising domain shape*, International Journal of High Performance Computer Applications, Vol. 13(4), pp. 334 - 353. 1999

[Walshsaw09] C. Walshaw, M. Cross, M. G. Everett. *Parallel dynamic graph partitioning for adaptive unstructured meshes*, Journal of Parallel Distributed Computers, Vol. 47(2), pp. 102 - 108. 1997

[Welsh] Matt Welsh, Matthias Kalle Dalheimer, Lar Kaufman. *Running Linux*, O'Reilly. 3rd Edition, 1999

[Wilkinson] Barry Wilkinson, Michael Allen. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*, Pearson Prentice Hall. 2nd Edition, 2005

[Williams] R. Williams. *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency, No. 3, pp. 457 - 481. 1991

[Wilson] Gregory V. Wilson. *Practical Parallel Programming*, MIT Press, Cambridge. 1995

[Wisconsin] University of Wisconsin. *WWW Computer Architecture Homepage*, `http://www.cs.wisc.edu/ arch/www`, Accessed May 26, 2006

[XHTML] Official Homepage of the W3C and the XHTML working group. `http://www.w3.org/MarkUp/`, Accessed February 09, 2009

[Xie01] M. Xie, S. Bauer, O. Kolditz, T. Nowak, H. Shao. *Numerical simulation of reactive processes in an experiment with partially saturated bentonite*, Journal of Contaminant Hydrology, Vol. 83, pp. 122 - 147. 2006

[Xie02] M. Xie, H. Shao, O. Kolditz. *Numerical Simulation of Non-isothermal Reactive Transport Processes of Dual-Geochemical Systems in FEBEX-Type Repositories*, Proceedings of the 11th Int. High-Level Radioactive Waste Management Conference (IHLRWM), American Nuclear Society, pp. 663 - 668. 2006

[Ziefle] G. Ziefle, J. Massmann, M. Kohlmeier, W. Zielke. *Simulation of the saturation process in a bentonite-sand-mixture with the finite element code RockFlow*, Bericht an die Bundesanstalt für Geowissenschaften und Rohstoffe (BGR), Hannover. 2007

[Zielke] W. Zielke, R. Helmig, K.-P. Kröhn, H. Shao, J. Wollrath. *Discrete Modelling of Transport Processes in Fractured Porous Rock*, Proceedings on International Congress on Rock Mechanics, Aachen. 1991

[Zilm] Karsten Günther (Hrsg.), Kester Grelck, Thorsten Zilm. *Linux – Die User-Referenz*, MITP-Verlag. 1999

[Zinterhof] Peter Zinterhof. *Grid Computing in Österreich*, `http://www.gup.uni-linz.ac.at/austriangrid/publications/talks/GridComputingInOesterreich.pdf`, Accessed July 12, 2006

Index

Ω Notation, **210**, 212
Θ Notation, **210**, 212
Heurísko, ***100***
heuriskein, ***100***
100Mbit/sec Ethernet, ***48***
10Mbit/sec Ethernet, ***48***
1D Element, 19
1D Grid, ***12***
1D Model, ***12****, 16*
1D Problem, 13, 20
1D Scenario, 13
2D Element, ***12****, 19*
2D Grid, 14, 19
2D Mesh, ***12****, 13*
2D Model, 16, 103
2D Problem, 20
3D Computer Games, ***37***
3D Computer Model, 9
3D Element, ***12***
3D Grid, 14
3D Model, 11, 16, 21, 103
3D Picture, 12
3D Problem, 20
3D Representation, 12
3D Scenario, 12
3D World, 11
3D representation, 11
4-Stage Assembly Line, ***28***
8-Bit Computers, ***27***

Abstract Level of Service, ***52***
Abstract View, 11
Abstraction, ***11****,* ***48***
Abstraction, Level, ***11***
Academic Parties, 58
Academic Research Labs, ***42***
Acceleration of Computation, 28
Acceptance in Programming Community, 57
Access Control Unit, ***53***
Access of Mutual Data, ***45***
Access System, ***51***
Access Time, 210
Access, Random, 47
Accounting, ***44***
Accounting Specialist, ***44***
Accumulated Time, 93
Accumulation, 94
Accumulator, ***25****, 26*
Accuracy, 16, 20, 100
Accuracy of Model, 20
Acknowledgement of Message, 92
Activity, 61
Activity Column, 93
Activity Distribution, 61
Activity Plot, ***61****, 92, 94*
Adders, ***39***
Adding Overhead, 62
Addition of Congruences, ***147***
Addition of Elements, 34
Additional Library, 57
Address Decoding, 38
Address Reading, 38
Adjacent Element, 20
Administration, 26, 50, 62
Administration Expenses, ***50***
Administrative Domain, ***51****, 51*
Administrative Hardware Unit, 32
Administrative Program, 33
Administrative Task, 26, 27, 32, 60
Advanced Programmer, 210
Agreement Negotiation, ***51***
Air Conditioning System, ***53***
Algorith, Modeling, ***18***
Algorithm, 18, 20, 23, 47, 64, 89, 94, 96, 99, 100, 210
Algorithm Application, 96
Algorithm Design, ***47***
Algorithm Scaling, 64
Algorithm, Classical, 99
Algorithm, Computing, 18
Algorithm, Discretization, 19
Algorithm, First-Rate, ***210***
Algorithm, Heuristic, 100
Algorithm, Modeling, 18
Algorithm, Parallel, 64, 89
Algorithm, Searching, 210
Algorithm, Sorting, 210
Algorithm, Specialized, 23
Algorithm, Underlying, 210
Allrounders, ***44***
Alluvial, ***19***
Alluvial Aquifer, ***19***
Altered Variable, 45
Alternative, 100
Alternatives, 100
Altitude, ***2***
ALU, ***25****, 26, 27, 30, 40*
Amount of Data, 56
Amount of Operations, 65
Amount of Overhead, 62
AMP System, 43
Analog Clock, ***146***

Analogy, ***3***
Analysis, 59
Analysis of Data, ***52***
Analysis of Performance, 65
Analysis, Performance, 64, 209
Analytische Zahlentheorie, 210
Analyzation of Requirements, ***55***
Annual Power Consumption, ***53***
Annual Power Consumption, ES, ***53***
Annual Power Consumption, Linz, ***53***
Anti-Virus Scanner, 100
API, 58
Appearance, Pseudo-Realistic, ***14***
Applicable Solution, 100
Application, 209
Application of Algorithm, 96
Application Programming Interface, 58
Applied Geology, 10
Approach, 20
Approximation, 60, 61, 104
Approximation of Execution Time, 60
Approximation of Runtime, 60
Approximation Process, 104
Aqueous Phase, ***10***
Aquifer, ***13****, 19*
Aquifer, Alluvial, ***19***
Aquifer, Shallow Alluvial, ***19***
Architecture, 51, 55, 89, 97, 210
Architecture Classes, ***35***
Architecture Classes after Flynn, ***35***
Architecture Designs, ***41***
Architecture Family, 30
Architecture of Machine, 90
Architecture of Parallel Machine, 89
Architecture of Supercomputer, 89
Architecture Realization, ***47***
Architecture Types, ***40***
Architecture, Parallel Machine, 89
Architecture, Supercomputer, 89
Architecture, Von Neumann, 26
Architectures, 40
Area, 104
Area, Digging, 90
Area, Overlapping, ***18***
Area, Rocky, 90–92
Area, Sandy, 91
Area, Transition, ***20***
Area, Triangular, 104
Arithmetic Circuits, ***39***
Arithmetic Instruction, 39
Arithmetic Logical Unit, ***25****, 26, 27*
Arithmetic Operation, ***1***
Arithmetic Processor, ***53***
Arithmetic, Modular, ***146***
Arithmetical Logical Unit, 30
Array, 34, ***39****, 60, 209, 210*
Array of Nodes, ***47***
Array of Nodes, Linear, ***47***
Array Processor, ***38***
Array, Dynamically Allocated, ***158****, 209*
Array, Standard, 209
Artificial Neuronal Network, ***12***
Aspect, Practical, 210
Assembly Line, ***28****, 38*
Assembly of Result, ***52***
Assembly Stage, ***28***
Assignment of Value, ***18***
Asymmetrical MPS, ***42***
Asymmetrical Multiprocessor, 43
Asymmetrical Multiprocessor System, 43
Asymptotic Lower Bound, ***211***
Asymptotic Notation, 210, 213
Asymptotic Tight Bound, 212
Asymptotic Time, 210
Asymptotic Upper Bound, 211
Asynchronous MIMD Systems, ***40***
Atmosphere, ***2***
Atmosphere Model, ***2***
Attila, ***3***
Attila the Hun, ***3***
Attribute, 100
Authorization, 51
Authorization Technique, ***51***
Automaton, Cellular, ***12***
Automobile, 28
Automobile Assembly, 28, 33
Autonomous Processor, ***35***
Autonomous Processors, ***40***
Average Automobile, 98
Average Case, 209
Average Computer, 98
Average Condition, 210
Average Needs, 98
Average Problem Size, 56
Axisymmetric, ***12***

Bachmann Notation, ***210***
Bachmann, Paul, ***210***
Background, 33, 60
Background Operation, 60
Balance of Workload, ***43***
Balanced Work Load, 56
Balancing, 57
Balancing Factor, ***56****, 57*
Bandwidth, ***46****,* ***55****, 55*
Bank, ***33***
Bar Element, ***20***
Bar Element, Linear, ***20***
Barrel, ***10***
Barrier, ***46***
Barrier Synchronization, ***46***
Base of Data, 60
Basic Computer Hardware, 26
Basic Computer Structure, ***25***
Basic Feature, 209
Basic Instruction Set, 38
Basic Structure of Computer, ***25***
Basis of Experience, 63

*Bedrock, **10**, 19*
*Bedrock, Horizontal, **19***
*Bedrock, Layered, **19***
*Bedrock, Regular, **19***
Behavior, 210
Behavior, Program, 209
*Benefit, **4**, 209*
*Beowulf Cluster, **50**, 50*
*Big O Notation, **209**, 210*
*Billion, **1***
Billions of Operations, 1
*Binary Digit, **27***
*Bit, **27**, 149*
Bit Array, 209
*Bit Arrays, **145***
Bit Level, 27
Block, 16
Block Level, 59
*Block-Level Parallelism, **32**, **59***
BMW, 98
*Board, **39***
Bonus, 63, 64
*Border Line, **17***
*Border, Line, **17***
*Border, Natural, **17***
Bore Core, 104
*Bore hole, **1***
Bore Site, 104
*Borecore, **11***
Boss, 63
*Bottleneck, **46**, 46, 89*
*Bottleneck of Performance, **46***
*Bound, Asymptotic Lower, **211***
Bound, Asymptotic Tight, 212
Bound, Asymptotic Upper, 211
*Bound, Lower, **211***
Bound, Upper, 209, 211
Boundary, 19, 20, 209
*Boundary Condition, **19**, 20*
Boundary Node, 95
*Boundary, Condition, **19**, 20*
*Boundary, Curved, **19**, 20*
Boundary, Upper, 65
Brain of the Computer, 26
Branch Command, 27
Brute Force, 100
Brute Force Method, 100
Bucket, 62, 64
*Bucket Business, **64***
*Bucket Carrying, **64***
*Bulk Data Analysis, **52***
*Bus, **25**, 46*
*Bus as Bottleneck, **46***
*Bus Monitoring, **46***
*Bus Snooping, **46***
*Bus System, 40, **45**, 45*
*Bus Systems, **25***
*Bus, Data, **27***
*Bus-Based Shared Memory System, **44***
*Bus-Based SM System, **44***
*Byte, **27**, 149*

C Program, 58
*C Programming Language, **27**, 31, 32, 57*
C Programming, MPI, 58
C++ Program, 58
*Cabinet, **42***
*Cable, 45, **53***
*Cable Bundle, **53***
*Cable Length, **45***
*Cables of Earth Simulator, **53***
*Cables of ES, **53***
*Cabling, **53***
*Cabling of Earth Simulator, **53***
*Cabling of ES, **53***
*Cabling, ES, **53***
Cache, 32, 39, 43, 45, 46, 48
*Cache Coherence, **45***
*Cache Coherence Problems, **46***
*Cache Component, **32***
*Cache Consistency, **45**, **48***
*Cache Consistency Protocols, **46**, 48*
*Cache Hierarchy, 27, **29**, **45***
*Cache Model, **43***
*Cache Models, **67***
Cache, Local, 46
*Caching, **45***
Calculation, 2, 3
Calculation of Equation, 61
Calculation, Complex, 61
Calculation, Weather, 1, fettdr2
Calculator, 26
Calculus, 209
*California, **3***
*Canada, **2**, 3*
Car, 98
Car, Formula-1, 98
Car, Racing, 98
*Carbonhydrate, **10***
Care, 17
Case, Average, 209
Case, Optimal, 65
Case, Worst, 209, 210
*Categories of Computer Architectures, **34***
*Cellular Automaton, **12***
*Center of the Earth, **1***
*Central Computer System, **33***
*Central Machine, **52***
Central Processing Unit, 26
*Central Reception, **44***
*Central Server, **52***
*Century, **10***
Chain of Processes, 9
*Change in Shape, **20***
*Change in Size, **20***
*Change in Value, **14**, **45***
*Change, Rate, **14***
Changes in Code, 60

Changes in Environment, ***52***
Changing Parameters, 64
Characteristics, 11, 100, 209
Checkpoint, ***19***
Chemical, ***10***
Chemical Impact, ***4***
Chemical Model, 11
Chemical Reaction, ***10****, 11*
Chemistry, 11
Chicken, ***56***
Chicken and Egg Problem, ***56***
Chip, ***39***
Chip-external Processing Speed, ***29***
Chip-internal Processing Speed, ***29***
Class of Computer Architecture, 40
Classes of Computer Architecture, ***34***
Classical Algorithm, 99
Classification after Flynn, ***34****, 41*
Classification of Architectures, ***34***
Client, 52
Climate Investigation, ***52***
Clock Cycle, 27, 28, 30, 32, ***40***
Clock Tick, 27
Clock, Analog, ***146***
Cluster, 46, ***48****, 50, 51*
Cluster Architecture, 50
Cluster Component, ***48****, 50*
Cluster Computer, ***48***
Cluster Installation, ***50***
Cluster Management, ***51***
Cluster Node, ***48****, 50*
Cluster Node Administration, 50
Cluster Node Configuration, 50
Cluster Nodes, 50
Cluster of Computers, ***48***
Cluster Size, ***50***
Cluster Software, ***48***
Cluster Usage, ***50***
Cluster, Architecture, 50
Cluster, Beowulf, ***50****, 50*
Cluster, Commodity, ***48***
Cluster, Computer, ***51***
Cluster, Homogeneous, ***50***
Cluster, Inhomogeneous, ***50***
Cluster, Installation, 50
Cluster, Linux, 50
Cluster, Size, ***50***
Cluster, Virtual, ***51***
Cluster, Windows, 50
Cluster, Wolfpack, ***50***
CM-1, ***40***
CM-2, ***40***
Coarse Grain, 59
Coarse Grain Parallelization, ***59***
Coarse Grain Program Structure, 59
Coarse Grain Structure, 59
Code, 22, 92, 94, 97, 98
Code Change, 60
Code Construct, 60
Code Execution, 40
Code Segments, 39
Code Statement, 60
Code Switch, 96
Code, Change, 60
Code, Construct, 60
Code, Parallel, 60
Code, Program, 94
Code, Serial, 60
Code, Statement, 60
Coefficient, 213
Coherence Problems, ***46***
Coherence Protocol, 46
Colleague, 62, 91
Collection of Computers, ***3***
Collection of Methods, 12
Collection of Techniques, 12
Collection of Theories, 12
Command, 32, 33
Command Line, 32
Command Pipeline, ***27****, 29*
Commandline, ***31***
Commandline Interface, ***7***
Commercial Parallel Processors, ***41***
Commercial Parallel System, 57
Commercial Scanner, 100
Commercial Success, ***40***
Commodity Cluster, ***48***
Communication, ***39****, 47, 50, 65, 89, 91, 92, 94, 95, 99*
Communication among Nodes, ***47***
Communication Aspect, 89, 99
Communication Delay, ***47***
Communication Expenses, ***50***
Communication Extent, 92
Communication Method, 92
Communication Network, ***40***
Communication of Processes, 62
Communication Protocol, 92
Communication Rate, 60, 89, 96, 99
Communication Structure, ***46***
Communication Time, ***61****, 65, 92, 94*
Communication Time Slot, 93
Communication, Extent, 92
Communication, Method, 92
Communication, Minimal, 47
Communication, Minimum, 95
Communication, Process, 62
Communication, Rate, 60, 89, 96, 99
Compiler, ***31****, 32, 57, 58, 60*
Compiler Directive, 58
Compiler Extension, 57
Compiler System, 57
Compiler, optimized, ***31***
Completion of Execution, ***59***
Complex Calculation, 61
Complex Computations, 56
Complex Element, ***20***
Complex Geometry, 19
Complex Model, 23

Complex Modelling, ***53***
Complex Problem, 99
Complex Protocol Procedures, ***46***
Complexity, 12, 99
Complexity Class, 99, 100
Complexity of Problems, 56
Complexity Theory, 96, 99, 210
Component, ***53***
Component, Earth Simulator, ***53***
Component, ES, ***53***
Components of Earth Simulator, ***53***
Components of ES, ***53***
Components of Parallel Architectures, 41
Composition, 90
Composition of Ground, 16
Composition, Ground, 16
Compound, 11
Compromise, 100
Computable Model, 9
Computation, ***1****, 9, 17, 23, 43, 45, 51, 60, 62, 91, 92, 94*
Computation Acceleration, 28
Computation Intensive, ***51***
Computation of Signals, ***52***
Computation Process, 15
Computation State, 66
Computation Step, 89
Computation Time, ***4****,* ***15****, 20, 62, 66, 92, 95*
Computation Time Slot, 93
Computation, Mathematical, 26
Computation, Model, 16, 18–20, 23
Computation, Result, ***4***
Computation, State, 66
Computation, Time, ***15****, 20*
Computation-Intensive Problem, ***4***
Computation-Rich Problem, ***4***
Computation-rich Problem, ***3***
Computational Challenge, ***51***
Computational Complexity Theory, 99
Computational Grids, ***51***
Computational Intensive Problem, ***52***
Computational Modeling, 9
Computational Performance, 100
Computational Power, 3
Computational Resource, 64
Computational Science, ***40***
Computational Speed, ***3***
Computational Unit, ***3***
Computer, ***1****, 3, 4, 26, 32, 33, 48, 51, 98, 99*
Computer Architecture, 30, 34, ***40****, 58*
Computer Architecture Designs, ***41***
Computer Architectures, 35, 37
Computer Archticture, 52
Computer Boards, ***42***
Computer Calculation, ***2***
Computer Cluster, 38, 46, ***51***
Computer Clusters, ***48***
Computer Era, ***1***
Computer Games, ***37***
Computer Graphics, ***37***
Computer Hardware, ***1****, 26, 53*
Computer Instructions, ***40***
Computer Language, ***51***
Computer Model, 9
Computer Mouse, 26
Computer Program, ***50****, 103*
Computer Science, ***53****,* ***55****, 57, 99, 100, 209, 210*
Computer Scientist, 99
Computer Security, ***148***
Computer Simulation, ***1***
Computer System, 33, ***46***
Computer Systems, 41
Computer Technique, ***1***
Computer Terminal, ***33***
Computer User, 33
Computer, Average, 98
Computer, Modern, 99
Computer, Multiprocessor, 90
Computer, Non-Specialized, 98
Computer, Out-Of-The-Box, 210
Computer, Out-of-the-Box, 20, 22, 94
Computer, Parallel, ***1****, 3,* ***4****, 61, 64, 66*
Computer, Program, 103
Computer, Standard, 22, 94
Computer, Stored-Program, 26
Computer, Weather-Predicting, ***3***
Computing Algorithm, 18
Computing Center, ***48****,* ***53****, 97*
Computing Center, ES, ***53***
Computing Components, 29
Computing Effort, 100
Computing Elements, ***40***
Computing Environment, 56
Computing Job, 33, ***50***
Computing Node, ***4****, 43, 46, 48, 61, 89–91*
Computing Nodes, 46
Computing Platform, ***51***
Computing Power, ***1****, 42,* ***51****, 51*
Computing Problem, ***3****,* ***51***
Computing Resources, 51
Computing Task, ***52***
Computing Time, 16, ***60****, 65, 90, 91*
Computing Unit, 45, 47, 66
Computing, Distributed, ***52***
Computing, Time, 16
Concept of Distributed Memory, ***48***
Concept of Locality, ***47***
Concept of Pipelining, ***38***
Concept of Shared Memory, ***48***
Concepts of Pipelining, ***32***
Conceptual Simplicity, 100
Concorrency, ***36***
Concurrency of Execution, ***36***
Concurrency of Processing, ***36***
Concurrent Programming, ***47***
Concurrent Programming Techniques, ***47***
Condition of Excavation, 64
Condition, Average, 210
Condition, Boundary, ***19****, 20*

Condition, Initial, ***19****, 20*
Condition, Optimal, 60
Condition, Perfect, 60
Conditional Branches, ***38****, 39*
Conditionals, ***39***
Conflict, 30
Conflicting Tasks, 30
Conflicts, ***30***
Conflicts, Pipeline, ***30***
Congeneric Data, ***37***
Congeneric Operands, ***37***
Congruence, ***147***
Congruence, Addition, ***147***
Congruence, Division, ***148***
Congruence, Multiplication, ***148***
Congruence, Subtraction, ***148***
congruent, ***147***
Coning, ***13****, 16*
Coning Effect, 16
Coning Problem, ***13****, 16*
Coning, Effect, 16
Coning, Problem, 16
Connection Machine, ***40***
Connection Machine CM-1, ***40***
Connection Machine CM-2, ***40***
Consistency, 90
Consistency Protocol, 48
Constant, 212
Constant Factor, 211, 212
Constant, Positive, 212, 213
Constraints of an Algorithm, 100
Construction of Earth Simulator, ***52***
Construction of ES, ***52***
Construction of Hardware, ***42***
Constructs of Code, 60
Consumption, Memory, 209
Consumption, Power, ***53***
Contamination, ***10***
Contamination, Source, ***10***
Continuous Data Set, ***11***
Contract, 63
Contract, Excavation, 60
Contraction of the Heart, ***40***
Contractor, 62, 63
Contractor, Roman, ***3****, 60*
Control of Resources, ***51***
Control Processor, ***39***
Control Unit, ***25****, 26,* ***39****, 39,* ***40****,* ***53***
Controlled Management, ***51***
Controlled Sharing, ***51***
Controller, ***38****, 39*
Cooperation, 44
Coordinate, ***19***
Coordinate, Geometrical, ***19***
Coordinate, Nodal, ***19***
Coordination, 44, 65
Coordination of Work, 62
Copy of Reality, 21
Correspondence, 40
Cost of Network, ***47***
Cost Reduction, 38
Costs, ***47***
Countermeasures, ***10***
Counterpart, 62
Counting System, ***147***
Countryside, ***4***
Coupled Process, 9
Coupling, ***48***
CPU, 26, 27, 30, 32, 36, 38, 42, 43, 48, 51, 52, 58
CPU Cycle, ***51***
CPU Management, ***51***
CPU Time, 52
Cray-2, ***38***
Cream Cake, 20, 94
Criterion, 210
Crossbar Switch, ***47****,* ***53***
Cryptography, ***148***
Cube, 12, 104
Cube of Grid, ***2***
Cubic Foot, ***3***
Cubic Meter, 60, 63, 64
Cubical Grid, ***2***
Curve of Efficiency, 65
Curved Boundary, ***19****, 20*
Curved Interface, ***19****, 20*
Customer Relationssships, ***44***
Customer Request Handling, ***44***
Customer Requests, ***44***
Customer Service, ***44***
Cycle, ***40***
Cycle Scavenging, ***51***

Daily Life, 9, 99
Daily Situation, 62
Data, 11, 26, 32, 104
Data Access, 45
Data Analysis, ***52***
Data Base, 60
Data Bus, ***27***
Data Bus Width, ***27***
Data Dependency, 29
Data Elements, 34
Data Exchange, 61
Data Grid, ***51***
Data Handling, 26, 94
Data Input, 89
Data Loading, 27
Data Movement, ***51***
Data Parallelism, ***34***
Data Partitioning, ***47***
Data Pipelining, ***38***
Data Postprocessing, 96
Data Preprocessing, ***4****,* ***48****, 89, 96, 99*
Data Processing, 89
Data Processing Instruction, ***39***
Data Processing Logic, ***39***
Data Processing Units, ***38***
Data Reassembly, ***47***

Data Server, ***48***
Data Set, 11, 34, ***40****, 57, 60, 96, 100, 209*
Data Set, Input, 210
Data Set, Structured, 60
Data Size, ***149***
Data Size Units, ***149***
Data Storing, 16
Data Stream, ***34***
Data Structure, 209, 210
Data Structures, ***47***
Data Transport, 27
Data Value, ***34***
Data, Discrete, 11, 104
Data, Exchange, 61
Data, Mesh, 21
Data, Model, 96
Data, Node, 21
Data, Output, 100
Data, Storing, 16
Data, Test, 100
Day, 2, 3
De Facto Standard, ***51***
Deadtime, 32
Decade, ***52***
Deceleration, ***92***
Decision Problem, 99
Decoding of Instructions, 39
Decoding Time, ***38***
Decrease of Execution Time, ***59***
Decrease, Monotonous, ***65***
Decreasing Efficiency Curve, 65
Decryption, ***148***
Decryption Algorithms, ***148***
Definite Value, 19
Delay, 32, 33, 47
Demand, ***3***
Demonstration, 14
Dense NAPL, ***10***
Density, ***10***
Departmental Grid, ***52***
Dependency, 32, 56
Depth, ***19****, 104*
Design, 64, 98
Design on Paper,
Design Requirements, ***64***
Designer, 100
Desktop Computer, 27, ***50***
Desktop Machine, 20, ***51****,* ***56****, 94, 98*
Desktop System, 22, 94
Detail, 210
Detailed Test, 210
Detected Solution, 100
Developer, 100
Development, 98, 100
Development of Program, 59
Development of Software, 57
Development Process, 97
Device, External, 32
Device, Mechanical, 32
Differential Equation, 20
Differential-Equation Systems, ***37***
Digger, ***64****, 90, 91*
Digger, Rock, 92
Digger, Sand, 92
Digging, 62, 65, 92
Digging Area, 62, 90
Digging Efficiency, 65
Digging Ground, 90
Digging Job, 63, 90
Digging Material, 91
Digging Time, 65, 91
Diligence, 17
Dimension, 12–14, 19
Dimension, Element, ***19***
Direct Communication, 47
Directing of Work, 63
Direction, 16
Disadvantage, 209
Discrete Amount of Data, ***11***
Discrete Data, 11, 104
Discrete Nodes, 17
Discretization, 9, 11–14, 16
Discretization Algorithm, 19
Discretization Process, 19
Discretization Program, ***19***
Discretization Rule, ***17***
Discretization, Algorithm, 19
Discretization, Problem, ***11***
Discretization, Process, 19
Discretization, Program, ***19***
Discretization, Rule, ***17***
Disk Storage, ***16****,* ***51***
Disk, Storage, ***16***
Dispatcher, ***30***
Dispatching of Instructions, 30
Dispatching Unit, ***31***
Displacement, ***12***
Displaying Device, 26
Distorted Element, ***20***
Distributed Computing, ***51****, 52*
Distributed Computing Project, ***52***
Distributed Data, ***51***
Distributed Main Memory, ***48***
Distributed Memory, 39, ***46****,* ***48****, 48,* ***53***
Distributed Memory Architecture, 47
Distributed Memory Architectures, ***46***
Distributed Memory Concept, ***48***
Distributed Memory Design, ***47***
Distributed Memory Environments, 47
Distributed Memory MIMD Systems, ***39***
Distributed Memory MPS, 40, ***46****, 47, 59*
Distributed Memory Multiprocessor System, 40, 59
Distributed Memory System, ***46****, 47, 48*
Distributed Memory Systems, ***46****,* ***58***
Distributed Shared Memory, ***40****,* ***46****, 48*
Distributed Shared Memory (DSM), ***47***
Distributed Shared Memory MPS, ***40****,* ***47****, 59*
Distributed Shared Memory Multiprocessor System, 59

*Distributed Shared Memory System, **46**, **48**, 58*
Distributed Shared Memory Systems, 42
*Distributed SM Architecture, **48***
Distributed SM Systems, 48
Distribution of Activities, 61
Distribution of Information, 60
Distribution of Model, 56, 57
Distribution of Work, 91
Distribution of Workload, 90, 91
Distribution, Problem, 94
Distribution, Workload, 91
Diversity of Tools, 12
*Division of Congruences, **148***
DM Architecture, 47
*DM Architecture Realization, **47***
*DM Architectures, **46***
*DM Design, **47***
*DM Environment Programming, **47***
*DM Environments, **47***
DM Memory System, 58
*DM MPS, **46**, 47, 59*
DM Multiprocessor System, 59
*DM System, **46**, 47, 48, 58*
*DNAPL, **10***
*Domain, **51**, 51, 210*
*Domain Decomposition, 56, **89**, 96, 99*
Domain Decomposition Algorithm, 99
Domain Decomposition Implementation, 103
Domain Decomposition, Implementing, 103
Domain Decomposition, Practical Aspects, 103
*Domain Geometry, **19**, 20*
Domain, Administrative, 51
*Domain, Geometry, **19**, 20*
*Domain, Problem, **12**, 13, 14, 17, 19, 20*
*Donald E. Knuth, **210***
Dot, 104
*Double Floor, **53***
*Double Floors, **53***
*Drilling, **1***
Drilling Material, 21
Drive, DVD, 32
Drive, Hard, 32
Drive, Tape, 32
*DSM, **47***
*DSM Architecture, **48***
*DSM Architectures, **47***
*DSM Environment, **48***
*DSM MPS, **40**, **47**, 59*
DSM Multiprocessor System, 59
*DSM Multiprocessor Systems, **40**, **47***
DSM System, 48
*DSM System Realization, **48***
DSM Systems, 48
*Dual Paradigm Program, **59***
DVD Drive, 32
*Dynamic Interprocess Network, **47***
Dynamic Process, 9
*Dynamical Scheduling, **30***
*Dynamically Allocated Array, **158**, 209*
*Earth Simulator, 20, 42, **52***
*Earth Simulator Cables, **53***
*Earth Simulator Cabling, **53***
*Earth Simulator Center, **42***
*Earth Simulator Components, **53***
*Earth Simulator Construction, **52***
*Earth Simulator Installation, **52***
Earth, Center, 1
*Ease of Programming, **39***
*EBytes, **149***
*Edge, **12**, 13, 20, 21, **104***
Edges, Set, 104
*Effect of Coning, **13**, 16*
*Effect of Starvation, **43***
Effect of Symmetry, 16
*Effect, Coning, **13**, 16*
Effect, Symmetry, 16
*Effective Parallel Code, **67***
*Effective Trick, **16***
*Effectiveness, **63**, 63*
*Effects of Signal Propagation, **45***
Efficiency, 64–66, 89
Efficiency Curve, 65
Efficiency Curve, Decrease, 65
*Efficiency of a Program, **63***
*Egg, **56***
*Electric Power Grid, **51***
*Elegance, **210***
*Element, **12**, 14–17, 20, 21, **104**, 104*
*Element Dimension, **19***
*Element Mixture, **19***
*Element Number, 15, 20, **104***
*Element Overlap, **18***
*Element Shape, **12**, 14, 19, 20*
Element Size, 14, 19, 20
Element Type, 19
Element, 1D, 19
*Element, 2D, **12**, 19*
*Element, 3D, **12***
Element, Adjacent, 20
*Element, Complex, **20***
*Element, Dimension, **19***
*Element, Distorted, **20***
Element, Finite, 12
Element, Grid, 21
Element, Linear, 20
*Element, Mixture, **19***
Element, Number, 15, 20
*Element, Overlap, **18***
*Element, Parallelepiped, **20***
*Element, Rectangular, **20***
Element, Shape, 19, 20
Element, Size, 19, 20
Element, Triangular, 20
Element, Type, 19
Elements of Array, 34
Elements of Data Sets, 34
Elevation, 21
Employee, 34, 37, 63, 65, 90

Employer, 90, 91
*Encryption, **148***
*Encryption Algorithms, **148***
End of Execution, 59
*Engineer, **1**, 4, 100*
*Engineering, **1***
*Enterprise Grid, **51, 52***
Entity, 11
*Environment, **52***
Environment Variable, 58
Environment, Multiprocessor, 23
Environment, Supercomputer, 23
*Environmental Changes, **52***
*Environmental Pollution, **10**, **52***
*Environmental Problem, **52***
*Environmental Problem Simulation, **52***
Equation, 3, 12, 56
Equation Calculation, 61
Equation System, 20, 21, 23, 94
Equation, Differential, 20
*Equipment, **52***
*Equipment Grid, **52***
Equipment of Excavation, 62, 64
Equipment, Excavation, 62
*Equipment, Photographic, **53***
*Error Message, **18**, 18*
*Error, Message, **18***
Error, Source, 19
ES, 42
*ES Anual Power Consumption, **53***
*ES Cables, **53***
*ES Cabling, **53***
*ES Components, **53***
*ES Computing Center, **53***
*ES Construction, **52***
*ES Installation, **52***
*ES Power Consumption, **53***
*ES System Configuration, **53***
Estimate, 213
Estimation, Runtime, 209
*Ethernet, **48***
Evaluation, 9, 60, 104
Evaluation Purpose, 104
Evening, 3
*Evening Report, **3***
*ExaBytes, **149***
Example, Weather, 3
*Examples of Concurrency, **36***
Excavating Job, 64
*Excavation, **1**, **3**, 4, 64, 65*
*Excavation Company, **65***
Excavation Condition, 64
Excavation Contract, 60, 63
Excavation Equipment, 62, 64
Excavation Example, 65
*Excavation Job, **3**, 3, 63–65*
Excavation Location, 62
Excavation Process, 63
Excavation Site, 62, 65
Excavation Work, 62
Excavation Workspace, 64
Excavation, Company, 65
Excavation, Condition, 64
Excavation, Contract, 63
Excavation, Equipment, 64
Excavation, Job, 63, 65
Excavation, Process, 63
Excavation, Site, 62, 65
Excavation, Workspace, 64
Excavator, 64, 65
Exception, 100
Exchange of Data, 61
Execution, 63
Execution of Parallel Program, 61
Execution of Program, 65, 66
Execution Time, 61, 63, 64
Execution Time, Approximation, 60
*Execution Time, Decrease, **59***
Execution Time, Increase, 63
*Execution Time, Parallel Program, **61**, 62, 63*
*Execution Time, Ratio, **62***
Execution Time, Sequential, 60
Execution Time, Sequential Program, 62, 63
*Execution Time, Serial Program, **59**, 62, 63*
Execution Time, Total, 61
Execution, Approximation, 60
Execution, End, 59
Execution, Start, 59
Execution, Time, 63, 64
Expectation, 60
Expense, 65, 66
Expense for Administration, 50
Expense for Communication, 50
*Expenses for Administration, **50***
*Expenses for Communication, **50***
*Expensive Experiment, **1***
Experience, 63
Experienced Programmer, 210
*Experiment, **1***
*Experiment, Expensive, **1***
*Experiment, Numerical, **1***
*Experiment, Physical, **1***
*Experiment, Unethical, **1***
*Experimental Machines, **40***
Expert, 9
Exploitation of Grid Computing, 51
Expression, Mathematical, 20, 94
Extension, Parallel, 92
Extent of Communication, 92
External Device, 32
*External Devices, **25***
Extra Bonus, 63, 64
*Extraction, Water, **13***

Fact, 104
Fact, Implementational, 210
*Factor Rings, **148***
Factor, Constant, 211, 212
*Factoring, **146***

Fathers of the Grid, ***51***
Fault, ***19****, 21*
Fault, Presence, ***19***
Faulty Mesh, ***18****, 18*
FE, 12
Feature, 98
Feature, Basic, 209
Federal Computing Center, ***48****, 48*
Fetching Instructions, ***39***
Fetching of Instructions, 39
Field Variable, ***19***
Field, Variable, ***19***
Final Result, 59, 91
Fine Grain, 59
Fine Grain Parallelsization, ***59***
Fine Grain Program Structure, 59
Fine Grain Structure, 59
Finite Amount of Data, ***11***
Finite Element, 12
Finite Element Model, 95
First-Rate Algorithm, ***210***
Fixed Problem Size, 65
Fixed Size of Problem, 65
Flexibility, ***51***
Flexibility of MPI, ***59***
Floating Point Instruction, ***38***
Floating Point Unit, ***30***
Floating-Point Numbers, ***38***
Flow Direction, 21
Flux, 21
Flynn's Architecture Classes, ***35***
Flynn's Classification, 34, 41
Flynn's Taxonomie, 40
Flynn's Taxonomy, 34, 41
Foot, Cubic, ***3***
Force, ***14***
Forecast of Geoscience Matters, ***53***
Form, 19
Formula-1 Car, 98
Fortran, ***27****, 31, 32, 57*
Fortran Code, 28
Fortran Program, 32, 58
Fortran Programming Language, ***27***
Four-Stage Assembly Line, ***28***
FPU, ***30****, 30*
Front-End, ***50***
Front-End Computer, ***50***
Front-End Node, ***50***
FTP, ***52***
Fully Connected Network, ***47***
Function, 12, 209–211
Function of Programming Language, 57
Function, Set of, 211
Functional Parallelism, ***34***
Functional Unit, 30, 33
Functional Units, 32
Functionality, ***51***

Gap, ***18****, 18, 20*
Gap Situation, ***18***
Gap, Situation, ***18***
Gasoline, ***10***
GBytes, ***149***
General-Purpose Computer, 97
Generic distributed MPS, ***47***
Geo Problem, ***52***
Geo Problem Simulation, ***52***
Geo-Resources, 9
Geo-Resources, Management, 9
Geo-Resources, Planning, 9
Geo-Resources, Utilization, 9
Geo-System, 9
Geologic Structure, ***19***
Geological 3D Model, 21
Geological Problem, ***52***
Geological Problem Simulation, ***52***
Geologist, 1
Geology, 10, ***37***
Geology, Applied, 10
Geometric Element, 12
Geometric Object, ***12***
Geometric Shape, ***12****, 12, 14*
Geometrical Coordinate, ***19***
Geometrical Shape, 16, 19, ***104***
Geometry, 19
Geometry, Complex, 19
Geometry, Domain, ***19****, 20*
Geometry, Problem, 20
Geometry, Simple, 19
Geoscience, ***4****, 4, 13, 14,* ***53***
Geoscience Matters, ***53***
GeoSys/RockFlow, ***48****, 57, 67,* ***69***
Geothermal Reservoir, ***4***
GigaBytes, ***149***
Gigaflops, ***53***
Gigawatt, ***53***
Global Clock, ***40***
Global Data Structures, ***47***
Global Grid, ***52***
Globus, ***51***
Globus Toolkit, ***51***
Golf, 98
Government, 58
Grade of Digging Efficiency, 65
Grade of Parallelism, ***28****, 29*
Grade, Digging Efficiency, 65
Gradual Transition, ***20***
Grain, ***10***
Graph, 100
Graph-Partitioning, ***57***
Graph-Partitioning Algorithm, 57
Graphic Card, 26
Graphic Processors, ***37***
Graphical Model, 12
Graphical Simulations, ***37***
Gravity, ***11***
Grid, ***2****,* ***12****, 13, 15, 18–20,* ***40****, 40, 51, 52*
Grid Architecture, ***51***

Grid Computing, ***51****, 51, 52*
Grid Computing Exploitation, ***51***
Grid Element, 21
Grid Functionality, ***51***
Grid Node, 21
Grid Point, ***2****,* ***15***
Grid Resource, ***51****, 51*
Grid Size, 16, ***52***
Grid Solution, ***51***
Grid Structure, 16, 59
Grid Type, ***51***
Grid User, ***52***
Grid, 1D, ***12***
Grid, 2D, 14, 19
Grid, 3D, 14
Grid, Computational, ***51***
Grid, Cubical, ***2***
Grid, Data, ***51***
Grid, Departmental, ***52***
Grid, Enterprise, ***52***
Grid, Equipment, ***52***
Grid, Global, ***52***
Grid, Information, ***52***
Grid, Omnipotent, ***52***
Grid, Point, ***15***
Grid, Resource, ***52***
Grid, Service, ***52***
Grid, Size, 16, ***52***
Grid, Structure, 16
Grid, User, ***52***
Ground, 104
Ground Composition, 16
Ground Material, 60, 90
Ground, Composition, 16
Ground, Digging, 90
Ground, Hard, 90–92
Ground, Polluted, ***11***
Ground, Rocky, 90, 92
Ground, Sandy, 90, 92
Ground, Soft, 90, 92
Groundwater, ***10****, 21*
Groundwater Flow, ***4***
Groundwater Level, 21
Groundwater Table, ***10***
Group Members, ***44***
Group of Allrounders, ***44***
Group of Laborers, ***4***
Group of Specialists, ***44***
GWh, ***53***

Handling of Instructions, 30
Handling, Data, 26
Handling, Program, 26
Hard Disk, 32
Hard Drive, 32
Hard Ground, 90–92
Harddisk, ***44***
Hardware, 1, 4, 20, 23, 25, 27, 29, 32, 42, 46, 48, 50, 51, ***53****, 53, 55, 60, 89, 94, 97, 210*
Hardware Architecture, ***51****, 55*
Hardware Architectures, ***44***
Hardware Aspect, 89
Hardware Components, 29, ***38***
Hardware Designer, ***67***
Hardware Device, 32
Hardware for Memory Management, 27
Hardware Hierarchy, ***48***
Hardware Implementation, 45
Hardware Industries, 58
Hardware Industry, ***34***
Hardware Logic, ***39***
Hardware Matters, ***44***
Hardware Unit, 30, 32
Hardware Unit, Administrative, 32
Hardware, Computer, ***1***
Hardware, Specialized, 23
Heart Contraction, ***40***
Heart of the Computer, 26
Heterogeneous Resources, ***51***
Heuristic, 100
Heuristic Method, 100
Heuristic Signature, 100
Hierarchy in Hardware, ***48***
Hierarchy, Register, 32
High Performance Computing, ***37****, 48*
High Performance Computing Center, ***38***
High Power Consumption, ***53***
High Standard, 100
High-Order Term, 213
High-Performance Access, ***46***
High-Performance Center, ***38***
High-Performance Computing, 42, ***52****, 97*
High-Performance Computing Center, 97
High-Resolution Model, 9
Highly Parallel System, ***53***
Hiring, 63
Hiring of People, 65
HLRS, ***38****,* ***48***
Hole, ***18****, 18*
Home Computer, 58
Homogeneous Cluster, ***50***
Homogeneous MPS, ***42***
Horizontal Bedrock, ***19***
Hour, 2
HPC, ***37****, 42,* ***52****, 97*
Human Being, ***19****, 97*
Hunger, ***1***
Hybrid Architectures, ***48***
Hybrid MPS, ***48****, 48*
Hybrid Multiprocessing Architecture, 48
Hybrid Multiprocessor Systems, ***48***
Hybrid Parallelization, ***59***
Hybrid Programming, ***59***
Hybrid Programming Technique, ***59***
Hybrid System, ***48***
Hydrogeological Model, ***1***
Hydrogeology, ***13***

I/O, ***43***

I/O Handling, ***43***
I/O Operations, ***43***
I/O Processor, ***53***
IBM, 30
Ideal Interconnection Network, ***47***
Identical Value, 19
Idle Time, 27, 32, 33, 56, ***61****, 93, 94*
IEC, ***149***
Impact on Countryside, ***4***
Impact, Chemical, ***4***
Implementation, 32
Implementation Model, 58
Implementation of Thesis, ***7***
Implementation, Domain Decomposition, 103
Implementational Fact, 210
Imposed Problem, 100
Imprecision, 60
Improvement, 32, 59, 94
Improvement, Wrapper, 89
Incoherence, ***43***
Incoming Messages, ***47***
Inconsistency, 18
Increase in Execution Time, 63
Increase in Problem Size, ***64***
Increase in Processor Counts, ***64***
Increase in Workforce, ***3***
Increase of Performance, 50
Increase, Problem Size, ***64***
Increase, Processor Count, ***64***
Increase, Workforce, ***3***
Independent Iteration, 58
Independent Operation, 60
Independent Tasks, 34
Index, Parallel, 65, 66
Index, Standardized Parallel, 65
Inequality, 213
Inequality, Right-Hand, 213
Infinite Set, ***11***
Influence on Weather, ***52***
Information, 11, 52, 104
Information Aggregation, ***51***
Information Caching, ***45***
Information Distribution, 60
Information Exchange, ***3***
Information Grid, ***52***
Information Power Grid, ***52***
Information Sharing, 61, 92
Information Storage, ***47***
Information Transmission, 27
Information, Node, 21
Information, Spatial, 11, 12
Information, Transport, 32
Infrastructure, 51
Inhomogeneous Cluster, ***50***
Inhomogeneous MPS, ***42***
Initial Condition, ***19****, 20, 59*
Input Data, ***55****, 99, 100, 209*
Input Data Set, 210
Input Data, Set, 210
Input Processing Bottleneck, 89
Input Size, 59, 60, ***62***
Input Terminal, ***50***
Input Type, 60
Input Unit, ***25, 26***
Input-Data Sets, ***37***
Installation of Cluster, ***50***
Installation of Earth Simulator, ***52***
Installation of ES, ***52***
Instance of Problem, ***100***
Instruction, ***3****, 3, 34, 36, 39*
Instruction Cycle, ***39***
Instruction Dispatch Unit, ***30***
Instruction Dispatchment, 30
Instruction Fetch, ***28***
Instruction Handling, 30
Instruction Pipeline, ***27***
Instruction Pipelining, ***36****,* ***38****, 38*
Instruction Prefetching, ***36***
Instruction Processing Unit, ***38***
Instruction Set, 38, ***40***
Instruction Stream, ***34***
Instruction Word, ***30***
Instruction, Precise, ***3***
Instructions, 38
Int. List of Supercomputers, ***52***
Integer, 210
Interconnection, ***47****, 55*
Interconnection Network, ***47****, 47, 55*
Interconnection Networks, ***55***
Interconnection of Nodes, 55
Interface, ***17****, 18–20*
Interface, Curved, ***19****, 20*
Interface, Material, ***17****, 18*
Interface, Natural, ***17***
Interior, 98
Interlocking, 27
Interlocking of Processes, 33
Intermediate Result, 92
Intermediate Solution, 20, 94
Intermittent Demands, ***51***
Internal Communication Structure, ***46***
Internal Operation, 29
Internal Processes, 26
Internal Processing, 27
Internet, ***51****, 52, 58*
Internet Service, ***52***
Internet-Connected Computers, ***52***
Interpolation, 104
Interpolation Puprpose, 104
Interpretation, ***4***
Interpreting Instructions, ***39***
Interprocess Communication, ***44****,* ***47****, 47*
Interprocess Network, ***47***
Interprocessor Communication, ***46***
Interrupt, ***47***
Interruption, ***47***
Interval, 100
Investigation, 9, 210

Investigation of Climate, ***52***
Investigative Method, ***11***
Irregular Depth, ***19***
Irregular Structures, ***38***
Isolation System, ***53***
Italy, ***3***
Iteration, 20, 58, 94

Japan, ***42***
Jetstream, ***3***
Jetstream-Predicting Processor, ***3***
Job, 33
Job Level, 27
Job Level Parallelism, 41
Job of Digging, 63
Job of Excavating, 64
Job, Digging, 90
Job, Excavation, 3, 64
John Public, 98
Jordan Valley, 21
Jostle, 57, 89, 96

KBytes, ***149***
Key Concept, ***51***
Key Term, ***51***
Keyboard, 25, 50
Keystroke, 26
KiloBytes, ***149***
Kilometer, 2
Knowledge, ***1****, 94*
Knuth, Donald E., ***210***

L1 Cache, ***29***
Laborer, ***3****, 4, 62–65, 90–92*
laborer, 60
Laborer Synchronization, 62
Laborer, Motivated, 63
Laborer, Skilled, ***3***
Laborer, Unskilled, ***3***
Lack of Scaling Ability, 38
Landfill, ***4***
Landscape, 21
Language Extension, 57
Language Layer, 57
Language Structure, 210
Large Scale, ***41***
Large Scale Parallel Machines, ***41***
Latency, ***32***
Latency Time, ***28****,* ***55****, 55*
Law, Physical, 11
Layer, Sandy, ***16***
Layer, Silty, ***16***
Layered Bedrock, ***19***
LCD, ***50***
Lee's Principle, ***62****, 63, 66*
Leo I, ***3***
Level of Abstraction, ***11***
Level of Parallelism, ***26****, 33, 59*
Level of Parallelization, ***26****, 59*
Level of Service, ***52***
Level, Bit, 27
Level, Job, 27
Level, Original, ***13***
Level, Parallelization, 27
Level, Program, 27
Level, Water, ***13***
Library, 57, 58
Library of Support Functions, 58
Life, Scientific, 9
Light NAPL, ***10***
Light-Weighted Process Management, 32
Lightning Proctection, ***53***
Lightning Proctection System, ***53***
Limitation, 100
Line, ***12****, 19*
Line, Border, ***17***
Line, Straight, ***19***
Linear Array of Nodes, ***47***
Linear Bar Element, ***20***
Linear Parallelepiped, ***20***
Linear Speedup, ***62****, 63*
Linked List, 209
Linux, ***50***
Linz, Austria, ***53***
Liquid Crystal Display, ***50***
Liquid, Pollutant, ***10***
List of Nodes, 95
List of Supercomputers, ***52***
List of Top 500, ***41***
List of Top500, 50
List, Linked, 209
LNAPL, ***10***
Load and Store Units, ***30***
Load Balancing, 50, 56, ***89****, 96, 99, 100*
Loading of Data, 27
Local Cache, ***46***
Local Memory, ***46***
Local Memory Access, ***46***
Local Networks of Workstations, ***41***
Locality, ***47***
Location, 104
Location of Excavation, 62
Location, Excavation, 62
Locking Mechanism, ***46***
Long-Term Memory, ***44***
Long-Term Source, ***10***
Loop, 32, 34, 58, 60
Loop Level, 59
Loop-Level Parallelism, 59
Los Angeles, ***3***
Low-Order Term, 213
Lower Bound, ***211***
Lust, ***1***

Machine, 98
Machine Architecture, 90
Machine Code, 60
Machine Command, 27
Machine Construction, ***47***
Machine Instruction, ***30****, 31*

Machine Instruction Word, ***31***
Machine Organisation, 39
Machine Parameter, 64
Machine, Desktop, 20, 94, 98
Machine, Driving, 98
Machine, Non-Specialized, 91
Machine, Parallel, ***4****, 92, 94, 98*
Machine, Processing, 21
Machine, Turing, 99
Machine, Von Neumann, 26
Main Memory, 48, 53
Main Memory System, ***53***
Mainboard, ***42***
Maintainance, 210
Making Money, 63
Making of Plans, 62
Malware, 100
Management of Distributed Data, ***51***
Mankind, ***1***
Manpower, ***44***
Mapping Problem, ***56***
MasPar MP-1, ***39***
Mass Product, 98
Mass-Produced Computer, ***48***
Massive Parallel Computer Clusters, ***38***
Material, 17, 90
Material Interface, ***17****, 18*
Material Numbers, 19
Material of Ground, 60
Material Property, ***17****, 19, 20*
Material Type, 19, 20
Material, Interface, ***17***
Material, Numbers, 19
Material, Poisonous, ***10***
Material, Property, ***17****, 19, 20*
Material, Type, 19, 20
Mathematical Algorithm, 20, 94
Mathematical Computation, 26, 56
Mathematical Equation, 20, 94
Mathematical Expression, 20, 94
Mathematical Model, 21, 97, 104
Mathematical Problems, ***37***
Mathematical Representation, 95, 103
Mathematically Proven, 100
Mathematics, ***53****, 210*
Matrix, 20, 21, 23, 94
Matrix Operations, ***37***
Matrix-Vector Multiplications, ***37***
Matrix-Vector Product, 22, 94, 95
Maximum, 10
MBytes, ***149***
Meal, 65
Means of Synchronization, ***46***
Measurement, 60
Measurement, Physical, 104
Mechanical Device, 32
MegaBytes, ***149***
Megaphone, ***37***
Megaphone Coordinator, ***37***
Memory, ***3****, 26, 32, 36,* ***38****, 38–40, 43, 45, 46*
Memory Access, 30, ***46***
Memory Consumption, ***7****, 209*
Memory Incoherence, ***43***
Memory Latency, ***32***
Memory Location, 27
Memory Management, 27
Memory Model, ***43***
Memory Module, ***3***
Memory Programming, ***48***
Memory Requirements, ***64***
Memory Space, 11
Memory System, ***53***
Memory Units, 41
Memory Usage, ***7***
Memory Word, ***3***
Memory, Latency, ***32***
Memory, Local, 46
Memory, Short Term, 26
Memory, Space, 11
Mercedes, 98
Mesh, ***12****, 15, 16, 18–21*
Mesh Data, 21
Mesh, 2D, ***12****, 13*
Mesh, Faulty, ***18****, 18*
MESI Protocol, ***46***
Message, 47, 89
Message Acknowledgement, 92
Message Construction, ***47***
Message Exchange, ***47***
Message Passing, ***57***
Message Passing Interface, ***57***
Message Passing Specification, 57
Message Sending, ***47***
Message Transfer, ***55****, 55*
Message, Error, ***18****, 18*
Message, Incoming, 47
Message, Status, 89
Messages, ***47***
Metaphor, ***51***
Meteorolgy, ***37***
Meter, ***1***
Method, 15, 23
Method of Communication, 92
Method, Brute Force, 100
Method, Heuristic, 100
Method, Investigative, ***11***
Method, Modeling, 16, 17
Method, Trial and Error, 100
Metis, 57, 89, 96
Metric, 63
Metric of Effectiveness, 63
Micro Command, 27
Micro Instructional Level, ***32***
Micro-Instruction, ***30***
Microcontroller, 30
Micrometer, ***11***
Microprocessor, 32
Microprocessors, ***41***

Millimeter, 11
MIMD Architectures, ***41***
MIMD Computer Architectures, ***41***
MIMD Computing System, ***35***
MIMD system, 52
MIMD Systems, 39, 40, ***41***
Minimal Communication, ***47***
Minimum, 10
Minimum of Communication, 95
MISD, 38
MISD Architectures, ***40***
MISD Computer Architectures, ***40***
MISD Computers, ***40***
MISD Machines, ***38***
MISD Systems, ***40***
Mismanagement, 90
Misorganization, 90
Misplaced Node, ***19***
Mistake, 18
Mixture, Element, ***19***
mod n, ***147***
Model, 11, 12, 18, 19, 21, 56, 94–96, 104
Model Accuracy, 20
Model Computation, 9, 16, 18–20, 23
Model Data, 96
Model Distribution, 56, 57
Model Node, 95
Model of Virtual Computer, 51
Model Processing, 9, 17, 19
Model Representation, 97
Model Simulation, ***4***
Model Solution, 20, 23
Model Structure, 17
Model Subdivision, ***4***
Model, 1D, ***12****, 16*
Model, 2D, 16, 103
Model, 3D, 9, 11, 16, 103
Model, Accuracy, 20
Model, Atmosphere, ***2***
Model, Chemical, 11
Model, Complex, 23
Model, Computable, 9
Model, Computation, 16, 18–20
Model, Computer, 9
Model, Data, 96
Model, Exemplary, 104
Model, Finite Element, 95
Model, High-Resolution, 9
Model, Hydrogeological, ***1***
Model, Mathematical, 21, 104
Model, Modern, 9
Model, Processing, 9, 17, 19
Model, Real-Life, 103
Model, Simulation, ***4***
Model, Solution, 20
Model, Structure, 17
Model, Test, 97
Modeling, ***4****, 9, 11, 14*
Modeling Algorithm, ***18****, 18*
Modeling Method, 16, 17
Modeling Process, 9, 11, 15, 17
Modeling Technique, 12
Modeling, Algorithm, 18
Modeling, Computational, 9
Modeling, Method, 16, 17
Modeling, Process, 17
Modeling, Realistic, 19
Modeling, Repository, ***4***
Modeling, Scientific, 9
Modelling, ***53***
Modelling, Complex, ***53***
Modern Desktop Computer, 27
Modern Microprocessor, 31
Modern Model, 9
Modern Processors, 29
Modification of Problem, 3
Modular Arithmetic, ***146***
Modulo, ***147***
Modulo Computation, ***146***
Money, 63
Money Making, 63
Monitor, 25, 47, ***50***
Monitoring, ***51***
Monitoring of Network Bus, ***46***
Motivated Laborer, 63
Motivation, 60
Motorola, 30
Mouse, 26
MP System, 65, 66
MP-1, ***39***
MPI, ***57****, 58*
MPI C Extension, 57
MPI C Programming, 58
MPI Commands, ***57***
MPI Decomposition, ***59***
MPI Documentation, 58
MPI Fortran Extension, 57
MPI Library Extension, 57
MPI Programming, ***57***
MPI Standard, ***57****, 57, 58*
MPI Website, ***58***
MPI, Flexibility, ***59***
MPI_Comm_rank, ***57***
MPI_Comm_size, ***57***
MPI_Finalize, ***57***
MPI_Init, ***57***
MPI_Reduce, ***57***
MPS, ***40****, 40, 51, 55, 58*
MPS, Asymmetrical, ***42***
MPS, Distributed Memory, 40, 47
MPS, Distributed Memory (DM), ***46***
MPS, Distributed Shared Memory, ***40****,* ***47***
MPS, DM, ***46****, 47*
MPS, DSM, ***40****,* ***47***
MPS, Generic Distributed, ***47***
MPS, Homogeneous, ***42***
MPS, Hybrid, ***48****, 48*
MPS, Inhomogeneous, ***42***

*MPS, Shared Memory, 40, **44**, 46*
*MPS, SM, **44***
*MPS, Symmetrical, **42***
*Multi-User Environment, **51***
*Multicomputer, **40**, **47***
*Multiple Data Stream, **34***
*Multiple Data Streams, **36***
Multiple Instruction - Multiple Data, 41
Multiple Instruction - Single Data, 40
*Multiple Instruction Stream, **34***
*Multiple Processing Units, **46***
*Multiplication of Congruences, **148***
*Multiplier, **39***
Multiprocessing Architecture, 55
Multiprocessing Environment, 46, 48
Multiprocessing System, 55, 66
*Multiprocessing System, Shared Memory, **44***
Multiprocessing Systems, 45
*Multiprocessing Systems (MPS), **40***
*Multiprocessor, **40**, **46**, 58*
Multiprocessor Architecture, 47
Multiprocessor Computer, 90
Multiprocessor Environment, 23, 42, 44, 46, 58
Multiprocessor System, 40, 51, 55, 58, 65
Multiprocessor System, Distributed Memory, 40
Multiprocessor System, Shared Memory, 40
*Multiprocessor Systems (MPS), **40***
*Multiprocessor Systems, DSM, **40***
Multiprocessors, 41
Multithreaded Processor Technique, 32
*Municipalities, **53***
*Mutual Data, **45***
*Mutual Exclusions, **46***

*NAPL, **10***
*NAPL, Dense, **10***
*NAPL, Light, **10***
*NASA, **52***
*NASA's Information Power Grid, **52***
*National Science Foundation, **52***
*National Technology Grid, **52***
*Native Rock, **10***
*Natural Border, **17***
*Natural Interface, **17***
Natural Number, 210
*Nature of the Application, **58***
NEC, 42
*NEC SX-8, **48***
*NEC SX4, **38***
*NEC SX6, **38***
*NEC SX8, **38***
*Neighbor Elements, **40***
*Neighbors, **40***
*Network, 44, 46, 47, **48**, 51*
*Network Communication, **41***
*Network Communication Improvement, **41***
*Network Connection, **45***
Network Connections, 25
*Network Costs, **47***
*Network Infrastructure, **51***
Network Interface, 26
Network of Computing Elements, 40
*Network Switch, **48***
*Network Topology, **47***
*Network, Fully Connected, **47***
*Network, Neuronal, **12***
*Networked Computer, **51***
*Networking Architectures, **67***
Networking Bus, 46
*Networking Bus System, **45***
*Networking Structures, **40***
*Networking Technique, **48***
*Networks of Workstations, **41***
*Networks, Local, **41***
*Neuronal Network, **12***
*Neuronal Network, Artificial, **12***
*Nodal Coordinate, **19***
Nodal Placement, 17, 18
Node, 12, 19, 21, 42, 51, 53, 65, 66, 92, 104
Node Communication, 47, 61
*Node Connection, **47***
Node Data, 21
Node Information, 21
Node Interconnection, 55
Node Listing, 95
*Node Memory, **46***
Node Number, 15, 16, 19, 20, 104
Node Placement, 14
Node, Boundary, 95
*Node, Computing, **4**, 61, 89–91*
Node, Discrete, 17
Node, Grid, 21
Node, Independent, 61
*Node, Misplaced, **19***
Node, Model, 95
Node, Number, 15, 16, 19, 20, 104
Node, Operating, 61
Node, Placement, 14
Node, Processing, 65, 66, 90–92, 94, 99
Node, Processor, 61, 65, 89
*Non-Aqueous Phase, **10***
*Non-Aqueous Phase Liquid, **10***
Non-Computer Scientist, 95, 99, 103
Non-deterministic Machine, 99
Non-specialized Computer, 98
Non-Specialized Machine, 91
*Nonlocal Memory Access, **46***
Notation Theta, 212
*Notation, Ω, **210**, 212*
*Notation, Θ, **210**, 212*
Notation, Asymptotic, 210, 213
*Notation, Bachmann, **210***
*Notation, Big O, **209***
*Notation, Omega, **210**, 212*
*Notation, Theta, **210***
*Notification Mechanism, **51***
Novice, 210
*NP Complete, **56***

NP Problem, 99
NP-Complete, 99, 100
NP-Complete Problem, 99
NPC, 99
NPC Problem, 99
Nuclear Repository, ***4***
NUMA, ***45****, 210*
NUMA Architecture, 210
Number, 100, 104
Number Cruncher, 90
Number of Elements, 20
Number of Materials, 19
Number of Nodes, 20, 65
Number of Processors, 46, ***62****, 64*
Number of Vertices, 16
Number Theorist, ***210***
Number Theory, ***146****, 148*
Number, Element, 15, 104
Number, Material, 19
Number, Natural, 210
Number, Node, 15, 16, 19, 104
Number, Vertices, 16
Numerical Problem, 22, 94
Numerical Simulation, ***1***

O Notation, 211
Object, Geometric, ***12***
Observation Well, ***13***
Off the Shelf Microprocessors, ***41***
Off-the-Shelf Computer, ***50***
Off-the-Shelf Desktop Computer, ***50***
Office, 36
Office Clerk, ***36, 37***
Office Work, ***36***
Oil, ***10***
Oil Reservoir, ***1***
Oil Reservoir Structure, ***1***
Oil Tank, ***10***
Old Greek, 100
Omega Notation, ***210****, 212*
Omnipotent Grid, ***52***
On-Chip Storage, ***39***
Open Multi Processing (OpenMP), ***58***
Open Standard, ***51***
Open-Plan Office, ***37***
OpenMP, ***58****, 58*
OpenMP Directive, 59
OpenMP Directives, ***59***
OpenMP Documentation, 58
OpenMP Homepage, ***58***
OpenMP Website, ***58***
Operand, 29
Operating Expenses, ***50***
Operating Node, 43, 47, 53, 55, 61
Operating System, ***32****, 33, 43,* ***48****, 50, 60*
Operating System Software, ***48***
Operation, 65, 66
Operation Independency, 60
Operation, Amount, 65
Operation, arithmetic, ***1***
Operation, Independent, 60
Operation, Total, 65
Operations, 34
Optimal Case, 65
Optimal Conditions, 60
Optimal Set, 99
Optimal Solution, 99, 100
Optimization Potential, 89
Optimization, Wrapper, 89
Organic Part, ***10***
Organisation, 44
Organization, 44
Organizational Matters, ***44***
Original Level, ***13***
Out-Of-The-Box Computer, 210
Out-of-the-Box Computer, 20, 22, 94
Out-of-the-Box Solution, 97
Outcome, 19, 97
Outcome, Proper, ***18***
Outcome, Realistic, ***15***
Outdated Information, ***46***
Output Data, 100
Output Processing Bottleneck, 89
Output Unit, ***25, 26***
Overhead, 47
Overhead Addition, 62
Overhead, Amount, 62
Overlap, ***18****, 20*
Overlapping Area, ***18***
Overlapping of Phases, 28
Overlapping Zone, ***18***

Paper Design, ***1***
Paradigm, 92
Paradigm of Science, ***1***
Parallel Algorithm, ***47****, 64, 89*
Parallel Algorithm Design, ***47***
Parallel Architectures, 41
Parallel Code, 60, ***67***
Parallel Computer, ***1****,* ***3, 4****, 55–57, 60, 61, 64, 66, 97*
Parallel Computer Architectures, ***34***
Parallel Computer Interconnection, ***55***
Parallel Computing, ***4****, 23,* ***55****, 56*
Parallel Computing Architectures, ***35***
Parallel Computing Environment, 56
Parallel Computing Hardware, ***34***
Parallel Computing Systems, 34
Parallel Computing Systems, Architecture, 34
Parallel Execution, 58
Parallel Extension, 92
Parallel Heuristics, 57
Parallel Index, 65, 66
Parallel Index, Standardized, 65
Parallel Machine, ***3****, 55, 92, 94, 98*
Parallel Machine Organization, ***39***
Parallel Machines, ***38****, 41*
Parallel Processing, 41
Parallel Processor, ***40***
Parallel Program, 57–59, 61, 62, 64, 65, 92
Parallel Program Execution, 60, 61

Parallel Program, Execution, 60, 61
Parallel Program, Execution Time, 63
*Parallel Program, Runtime, **60***
*Parallel Programming, 52, **57**, 58, 59*
Parallel Programming Community, 57
*Parallel Programming Extensions, **58***
*Parallel Programming Language, **57***
Parallel Programming Language Layer, 57
*Parallel Programming Techniques, **57***
*Parallel Project, **48**, 55*
Parallel System, 59
Parallel Version, 55, 63
Parallel Versions, 48
*Parallelepiped, **20***
*Parallelepiped Element, **20***
*Parallelepiped, Linear, **20***
Parallelism, 27, 32, 67
*Parallelism at Bit Level, **27***
*Parallelism at Block Level, **31**, 59*
*Parallelism at Instruction Level, **27**, 59*
*Parallelism at Job Level, **33***
*Parallelism at Process Level, **32***
*Parallelism at Task Level, **33***
Parallelism of Pipelining, 28
*Parallelism, Bit Level, **27***
*Parallelism, Block Level, **31***
*Parallelism, Data, **34***
*Parallelism, functional, **34***
*Parallelism, Instruction Level, **27***
Parallelism, Job Level, 41
*Parallelism, Process Level, **32***
*Parallelism, Shared Memory, **58***
*Parallelism, SM, **58***
*Parallelism, Spatial, **33***
*Parallelism, Task Level, **32***
*Parallelism, temporal, **33***
Parallelization, 27, 34, 56, 59, 67, 89
parallelization, 67
Parallelization of Serial Program, 59
Parallelization Parameter, 65
*Parallelization Process, **4**, **67***
Parallelization Project, 94
*Parallelization, Coarse Grain, **59***
Parallelization, Faster, 59
*Parallelization, Fine Grain, **59***
*Parallelization, Hybrid, **59***
Parallelization, Simpler, 59
*Parallism at Task Level, **32***
Parameter, 10, 14, 15, 18, 59, 210
Parameter, Changing, 64
*Parameter, Machine, **64***
Parameter, Starting, 55
Parameter, Time-Dependent, 59
*Part, Organic, **10***
Participating Node, Processor, 65
Participating Processor Node, 65
*Partitioning Algorithms, **57***
*Partitioning Tools, **57***
*Paul, **3**, 60, 62, 63, 90, 91*
*Paul Bachmann, **210***
*PC, **56***
*Peak Performance, **53***
*PentiumPro Processor, **30***
People Hiring, 65
Perfect Conditions, 60
Performance, 32, 33, 38, 53, 63, 100, 209, 210
*Performance Analysis, **59**, 64, 65, 209*
*Performance Bottleneck, **46***
Performance Measure, 63
Performance Measuring, 63
Person, 12
Personal Preference, 58
*Peter, **3**, 60, 62–65, 90, 91*
Phase of Programming, 92
*Phase, Aqueous, **10***
*Phase, Non-Aqueous, **10***
Phenomenon, 11
Photographic Equipment, 53
*Physical Effect, **45***
*Physical Experiment, **1***
Physical Law, 11
Physical Measurement, 104
Physical Shape, 91
*Physical Signal, **45***
*Physically Distributed Memory, **48***
*Physics, **37***
Picture, 3D, 12
Pipeline, 29, 38
*Pipeline Conflicts, **30***
*Pipeline Hazards, **30***
Pipeline Limitations, 29
Pipeline Stage, 29
*Pipeline Stages, **28**, 33*
*Pipelined Execution of Instructions, **36***
*Pipelining, **27**, 32, 38*
*Pipelining Concept, **38***
*Pipelining of Data, **38***
Placement, 14, 17
Placement, Nodal, 17, 18
Plan, 62
Plan Making, 62
*Plane, **19***
*Plane, Straight, **19***
*Plume, **10**, 11*
*Plume, Poisonous, **10***
Point A, 98
*Point of Failure, **43***
*Point, Grid, **15***
*Poisonous Material, **10***
*Poisonous Plume, **10***
Pollutant, 10, 11
*Pollutant Liquid, **10***
*Polluted Ground, **11***
*Pollution, **10**, 11, **52***
*Pollution of Environment, **52***
*Pollution Potential, **10***
*Pollution Problem, **52***
*Pollution Scenario, **11**, 12*

Pollution, Environmental, ***10***
Pollution, Potential, ***10***
Pollution, Source, ***10***
Pollution, State, 11
Polynomial Time, 99
Pope, ***3***
Pope Leo I, ***3***
Pore Space, ***10***
Portability, 210
Porting, 98
Porting a Problem, 65
Porting of Program, 60
Positive Constant, 212, 213
Post Processing, 96
Potential, Pollution, ***10***
Power Consumption, ***53***
Power Consumption, Annual, ***53***
Power Consumption, ES, ***53***
Power Consumption, Linz, ***53***
Power Supply, ***53***
Power Supply Facility, ***53***
Power, Computing, ***1***
PowerPC Processor, ***30***
Practical Aspect, 210
Practical Life, 99
Precaution, ***18****, 18*
Precision, 16, 100
Prediction, 32
Prediction, Weather, ***2****, 3*
Preprocessing, ***4****, 21,* ***48****, 96*
Preprocessing Activity, ***4***
Prerequisite, 97
Presence of Faults, ***19***
Prevention, 32
Primary Cache, ***29***
Principle of Caching, ***45***
Principle of Pipelining, ***28***
Principles of Pipelining, ***38***
Print Job, 33
Printer, 25, 33
Problem Complexity, 56
Problem Description, 9
Problem Discretization, 9, ***11***
Problem Distribution, ***50****, 94*
Problem Domain, ***12****, 13, 14, 17, 19, 20*
Problem Domain Dimension, ***12***
Problem Domain, Shape, 19
Problem Domain, Size, 19
Problem Geometry, 20
Problem of Coning, 16
Problem of Pollution, ***52***
Problem Parallelization, 56
Problem Porting, 65
Problem Scenario, 16
Problem Size, ***4****, 16, 56, 64, 65*
Problem Size, Fixed, 65
Problem Sizes, ***39***
Problem Subdivision, ***4***
Problem Type, 55
Problem, 1D, 13, 20
Problem, 2D, 20
Problem, 3D, 20
Problem, Complex, 99
Problem, Computation-Intensive, ***4***
Problem, Computation-Rich, ***4***
Problem, Computation-rich, ***3***
Problem, Computing, 3
Problem, Coning, ***13****, 16*
Problem, Description, 9
Problem, Discretization, 9
Problem, Domain, 14, 17, 19, 20
Problem, Geometry, 20
Problem, Modification, 3
Problem, Real-Life, 20
Problem, Real-World, 12, 19, 22, 94
Problem, Scenario, 16
Problem, Size, 16, 64
Problem, Subdivision, ***4***
Problem, Traveling Salesman, 100
Procedure, 32, 97
Procedure, Searching, 210
Procedure, Sorting, 210
Process, 11, 12, ***32****, 33*
Process Communication, 62
Process Management, 32
Process of Excavation, 63
Process of Parallelization, ***4***
Process, Approximation, 104
Process, Computation, 15
Process, Coupled, 9
Process, Discretization, 19
Process, Dynamic, 9
Process, Modeling, 9, 11, 15, 17
Process, Parallelization, ***4***
Process, Transformation, 11
Process, Transient, 9
Processing Environment, 20
Processing Instruction, 36
Processing Machine, 21
Processing Node, 42, 45–48, 53, 55–57, 59, 65, 66, 90–92, 94, 99
Processing Node Interconnection, ***55***
Processing Node, Number, 65
Processing Nodes, 42, 44, 47
Processing of Program, 26
Processing Speed, 29, ***45***
Processing Stage, ***15***
Processing Steps, 21
Processing Time, 16, 30
Processing Unit, ***1****, 27, 39, 42, 46, 47, 56, 65*
Processing Units, 39, ***40****, 40*
Processing, Model, 17, 19
Processing, Time, 16
Processor, ***3****, 3, 22, 32, 39, 40, 42, 43, 46, 56, 60–62, 65, 92, 94, 95*
Processor Architecture, ***55***
Processor Arrays, 36, ***38****, 38, 39*
Processor Chip, ***39***

Processor Count, 64
Processor Node, 45, 53, 61, 65, 89
Processor Request, 61
Processor Startup Time, 62
Processor System, 39
Processor Technique, 32
Processor Utilization, 63
*Processor, Count, **64***
*Processor, Jetstream-Predicting, **3***
Processor, Number, 64
Processor, Receiving, 47
Processor, Startup Time, 62
Processors, 41, 43
Processors, Indenpendent, 61
*Procssing Units, **39***
*Product, **1***
Product, Matrix-Vector, 22, 94
*Production Well, **13***
Professional, 9
Professor, 210
Program, 26, 32, 97, 210
Program Analysis, 59
Program Behavior, 209
Program Call, 209
Program Code, 32, 94, 97
Program Development, 59, 98
Program Execution, 32, 60, 65, 66
Program Execution, Parallel, 60, 61
Program Handling, 26
Program Level, 27
Program Line, 32
*Program Memory, **36***
Program Operation, 65
Program Parallelization, 59
*Program Performance Analysis, **59***
Program Porting, 60
Program Processing, 26
Program Restructuring, 94
Program Result, 26, 61
Program Runtime, 60
Program Step, 27, 32
Program Structure, 59
Program, Computer, 103
*Program, Discretization, **19***
Program, Execution, 65, 66
Program, Execution Time, 63
Program, Fortran, 32
Program, Line, 32
Program, Operation, 65
Program, Parallel, 59, 61, 62, 64, 92
Program, Restructuring, 94
Program, Result, 61
Program, Run, 32
Program, Scalable, 65
Program, Sequential, 59, 64–66, 90
Program, Step, 32
Program, Wrapper, 104
*Programmer, 3, 4, 22, 47, **67**, 89, 91, 94, 97, 100, 103, 210*
Programmer, Advanced, 210
Programmer, Experienced, 210
*Programming, **40**, 47, 98*
*Programming Language, **27**, 31, 58, 60, 92, 210*
*Programming Language C, **27***
*Programming Language Fortran, **27***
Programming Language Function, 57
Programming Language, C, 32
*Programming Language, Parallel, **57***
Programming Library, 57
*Programming Model, **48***
Programming Paradigm, 92
Programming Phase, 92
*Programming Techniques, **47**, **57***
Programming, Language, 92
Programming, Paradigm, 92
Programming, Phase, 92
Project, 52
Propagation Delays, 45
*Proper Outcome, **18***
Property, 18
*Property, Material, **17**, 19, 20*
*Protection System, **53***
Protocol, 48, 52, 92
*Protocol Procedures, **46**, 46*
Protocol, Communication, 92
*Prototype, **1***
*Prototype Design, **1***
*Prototype Testing, **1***
*Prototype, Design, **1***
*Prototype, Testing, **1***
*Prototyping, **1***
Pseudo Code, 34
*Pseudo-Realistic, **14***
*Pseudo-Realistic Appearance, **14***
*Pump-and-Treat, **10***
*Pumping, **13***
*Pumping Scenario, **13***
*Pumping Test, **11***
Pumping Well, 14, 16
Purpose, Evaluation, 104
Purpose, Interpolation, 104

*Quadrillion, **1***
Quality, 100
Quarter, 16

Racing Car, 98
Racing Team, 98
*RAM, **44***
*Random Access, **47***
*Random Access Memory, **44***
Randomness, 210
*Rate of Change, **14***
Rate of Communication, 60, 89
*Reaction, Chemical, **10**, 11*
*Reactive System, **10***
Readability, 104
Reader, 103
Real Life, 104

Real-Life Model, 103
Real-Life Problem, 20
Real-World Problem, 9, 12, 19, 22, 94
Real-World Scenario, 11
Realistic Modeling, 19
Realistic Outcome, ***15***
Realization of Architecture, ***47***
Realization of DM Architecture, ***47***
Realization of DSM System, ***48***
Reassembly, ***47***
Reception, ***44***
Recreation, 65
Rectangular Element, ***20***
Redability Purposes, 104
Region of Interest, ***2***
Register, 26, 32, 39
Register Hierarchy, 32
Registers, ***39***
Regular Bedrock, ***19***
Regular Grid, ***40***
Regular Program Structures, ***39***
Regular Structures of Code, ***39***
Remainder, ***147***
Remote Access, ***53***
Remote Access Control, ***53***
Remote Access Control Unit, ***53***
Remote Controlling, ***52***
Remote User, ***51***
Reorganization, 32
Repository Modeling, ***4***
Repository, Nuclear, ***4***
Representation, 11
Representation, 3D, 11, 12
Representation, Mathematical, 95, 103
Request, 61
Request by Processor, 61
Request for Synchronization, 61
Request, Processor, 61
Requirement, 55, 100
Requirement Analyzation, ***55***
Requirements, Design, ***64***
Requirements, Memory, ***64***
Research, 210
Research Area, 57
Research Labs, ***42***
Research of the Universe, ***1***
Researcher, 9, 20
Reservoir, Oil, ***1***
Resource, 52
Resource Grid, ***52***
Resource Sharing, ***51***
Resource, Computational, 64
Resource, Grid, ***51***
Resources, 90
Restricting Factor, 29
Result, 59, 92
Result Assembly, ***52***
Result of Computation, ***4***
Result of Program, 61
Result, Final, 91
Result, Intermediate, 92
Result, Program, 26
Result, Routine, 26
Right-Hand Inequality, 213
Ring Topology, ***47***
Rock, ***10****, 11, 64, 90, 91, 104*
Rock Digger, 92
Rock Laborer, 92
Rock, Native, ***10***
Rock-Digging, 91
Rocky Area, 90–92
Rocky Ground, 90, 92
Roman Contractor, ***3****, 60, 62, 63*
Roman Empire, 64
Roman Excavation, ***4***
Rome, ***3***
Routine, ***3****, 26*
Routing, ***47***
Routing Algorithm, ***47****,* ***55****, 55*
Rubber Pads, ***53***
Rule, 100
Rule of Thumb, ***16***
Rule, Discretization, ***17***
Rule, Thumb, ***16***
Runtime, ***59****, 99, 209, 210*
Runtime Approximation, ***59***
Runtime Estimation, 209
Runtime Maximum, 209
Runtime of Program, 60
Runtime Prediction, 60
Runtime, Approximation, 60

Salary, ***34***
Sand, ***17****, 91*
Sand Digger, 92
Sandy Area, 91
Sandy Ground, 90, 92
Sany Layer, ***16***
Saving, Time, 16
Scalability, 43, 50, ***55****, 55,* ***64***
Scalable Program, 65
Scalar Processing, ***37***
Scalar Processor Architecture, ***36***
Scaling, ***38****, 39,* ***46***
Scaling of Algorithms, 64
Scenario, 16
Scenario, 1D, 13
Scenario, 3D, 12
Scenario, Pollution, ***11****, 12*
Scenario, Problem, 16
Scenario, Pumping, ***13***
Scenario, Real-World, 11
Scheduling of Instructions, 39
Scheduling, dynamically, ***30***
Scheduling, statically, ***31***
Science, 11, 210
Science, Computer, 210
Scientific Field, 99
Scientific Life, 9

*Scientific Method, **1***
Scientific Modeling, 9
*Scientific Tool, **1***
*Scientist, **1**, **4**, 9, 20, 96, 100, 210*
Scientist, Non-Computer, 95
*Sea Level, **2***
Search Algorithm, 210
Searching Procedure, 210
*Second, **1**, 2*
*Secure Authorization, **51***
*Security, **51***
*Security Provisioning, **51***
*Segments of Code, **39***
*Seismic Isolation System, **53***
*Semaphore, **46**, 47*
Sequence of Numbers, 100
*Sequence of Operations, **39***
Sequential Code, 58
Sequential Counterpart, 62
Sequential Execution Time, 60
Sequential Heuristics, 57
Sequential Program, 57, 59, 64–66, 90
Sequential Program Porting, 60
Sequential Program, Execution Time, 63
Sequential Program, Porting, 60
*Sequential Program, Runtime, **59***
*Sequential Solution, **55***
*Serial Instructions, **39***
Serial Program, 60
Serial Program Code, 60
Serial Program, Code, 60
Serial Program, Execution Time, 63
Serial Program, Parallelization, 59
*Serial Program, Runtime, **59***
*Series of Prototypes, **1***
Server, 52
Service, 51, 52
*Service Grid, **52***
*Service Provider, **52***
*Set of Data, **40***
*Set of Edges, **104***
Set of Functions, 211
*Set of Independent Tasks, **47***
*Set of Information, **11***
*Set of Instructions, **40***
*Set of Vertices, **104***
Set, Data, 11
*Set, Edges, **104***
*Set, Infinite, **11***
Set, Input Data, 210
*Set, Vertices, **104***
*home, **52***
*Setup of Parallel System, **59***
*Shallow Alluvial Aquifer, **19***
Shape, 14, 104
Shape of Element, 19, 20
Shape of Problem Domain, 19
*Shape, Change, **20***
*Shape, Element, **12**, 14, 19, 20*
*Shape, Geometric, **12**, 12, 14*
*Shape, Geometrical, 16, 19, **104***
Shape, Physical, 91
Shape, Problem Domain, 19
*Shared Information, **46***
*Shared Memory, 43, **44**, 45, 48, 53*
*Shared Memory (SM), **44***
*Shared Memory Architecture, **46***
*Shared Memory Concept, **48***
*Shared Memory Environment, 46, **58***
*Shared Memory Implementations, **45***
*Shared Memory MPS, 40, **44**, 46, 58, 59*
Shared Memory Multiprocessing Environment, 46
*Shared Memory Multiprocessing System, **44**, 46*
Shared Memory Multiprocessor System, 59
Shared Memory Multiprocessor Systems, 40
*Shared Memory Parallelism, **58***
*Shared Memory Programming, **48***
*Shared Memory Programming Model, **48***
*Shared Memory Standard, **58***
*Shared Memory Support, **48***
Shared Memory System, 47, 48, 53
*Shared Memory System, Bus-Based, **44***
*Shared Memory System, Switch-Based, **44***
Shared Memory Systems, 58
*Shared Variable, **45**, 45*
*Sharing of Distributed Data, **51***
Sharing of Information, 61, 92
Short Term Memory, 26
*Short-Term Memory, **44***
Shortest Paths, 100
*Shovel, **3**, 60, 62*
*Side Effect, **51***
*Side-Product, **10***
Signal, 45
*Signal Computation, **52***
*Signal Delaying, **45***
*Signal Propagation, **45***
*Signal Propagation Delay, **45***
*Signal Propagation Effects, **45***
*Signal Travel, **45***
Signature, 100
Signature, Heuristic, 100
*Silt, **17***
*Silty Layer, **16***
SIMD, 36
SIMD Architectures, 36
SIMD Class, 36
*SIMD Computer Architectures, **36***
SIMD Machine, 39
SIMD Machines, 39
*SIMD System, **38***
SIMD Systems, 40
Simple Array, 60
Simple Geometry, 19
Simplicity, 60, 100
*Simulation, 1, **52**, 56*
*Simulation of Environmental Problems, **52***
*Simulation of Geo Problems, **52***

Simulation, Numerical, ***1***
Simultaneous Communication, ***47***
Single Data Stream, ***34****, 36*
Single Instruction - Multiple Data, 36
Single Instruction Stream, ***34****,* ***36****, 36*
Single Instruction, Single Data (SISD), 35
Single Instrustion Stream, 36
Single Processor Machine, ***43***
Single Processor Systems, ***35***
Single Program Execution, ***41***
Single-Processing Environment, 66
Single-Processor System, 66
Single-Stage, ***53***
Single-Stage Crossbar Switch, ***53***
Sink of a Signal, ***45***
SISD, 36, 38
SISD Architectures, 36
SISD Class, 36
SISD Computer Architectures, ***35***
SISD Computing System, ***34***
Site, Bore, 104
Situation, Daily, 62
Situation, Gap, ***18***
Situation, Specific, 210
Size, 14, 210
Size of a Grid, ***52***
Size of Element, 19
Size of Input, 60
Size of Problem, 64
Size of Problem Domain, 19
Size of Subproblem, 95
Size of Time Step, ***20***
Size, Change, ***20***
Size, Element, 14, 19, 20
Size, Grid, 16
Size, Problem, 16
Size, Problem Domain, 19
Skilled Laborer, ***3***
Slowdown, ***62****, 91*
SM, ***44***
SM Architecture, ***46***
SM MPS, ***44****, 58, 59*
SM Multiprocessing System, 46
SM Multiprocessor System, 59
SM Parallelism, ***58***
SM Standard, ***58***
SM System, 47, 48
SM System, Bus-Based, ***44***
SM System, Switch-Based, ***44***
SM Systems, ***44***
Small Scale, ***41***
Small Scale Multiprocessors, ***41***
Snapshot, 11
Snippet of Fortran, 28
Snoopy Protocol, ***46***
Soft Ground, 90, 92
Software, 19, 21, 22, 44, 48, 50, 51, 89, 90, 94, 98, 209
Software Architecture, ***51***
Software Development, ***41****, 57, 97*
Software Development Process, 97
Software Engineer, 22, 89, 94, 97
Software Industries, 58
Software Package, 89
Software Packages, ***41***
Software Unit, 32
Software, Specialized, ***19***
Soil, 63, 64
Solution of Model, 20
Solution Space, 100
Solution, Applicable, 100
Solution, Computer, 9
Solution, Detected, 100
Solution, Intermediate, 20, 94
Solution, Model, 20, 23
Solution, Optimal, 100
Solution, Stable, ***20***
Solution, Universal, 210
Solving a Problem, 59
Sorting Algorithm, 210
Sorting Procedure, 210
Source Code, 60
Source Code Translation, 60
Source of a Signal, ***45***
Source of Contamination, ***10***
Source of Error, 19
Source of Overhead, ***47***
Source of Pollution, ***10***
Source, Contamination, ***10***
Source, Error, 19
Source, Long-Term, ***10***
Source, Pollution, ***10***
Space of Storage, 20
Space, Storage, 15, 20
Spatial Information, 11, 12
Spatial Parallelism, ***33***
Specialist, 97
Specialists, ***44***
Specialized Algorithm, 23
Specialized Hardware, 23, 26
Specialized Software, ***19***
Specific Situation, 210
Speed of CPU, 38
Speed, Computational, ***3***
Speeding up Computations, 59
Speedup, 56, ***62****, 63, 66*
Speedup, Linear, 63
Speedup, linear, ***62***
Spot, 104
Square, ***19****, 104*
Square Kilometer, ***2****, 3*
Stability, 210
Stable Solution, ***20***
Stage of Assembly, 28
Stage, Processing, ***15***
Standard, 51
Standard Array, 209
Standard Computer, 22, 94
Standard Data Sizes, ***149***

Standard Networking Technique, ***48***
Standard Processors, ***38***
Standard Units, ***149***
Standardized Parallel Index, 65
Stanford, 210
Start of Execution, 59
Start of Work, 62
Starting Parameter, ***55***
Starting Point, 60
Starting Situation, 59
Startup Time, ***60****, 64*
Startup Time of Processor, 62
Starvation, ***43***
State of Computation, 66
State of Pollution, 11
Statement Execution, 39
Statements, ***39***
Statements of Code, 60
Static Interconnection Network, ***47***
Statical Scheduling, 30, ***31***
Statistical Output, ***7***
Status Message, 89
Steel Construction, ***53***
Storage Issue, 89
Storage Management, ***51***
Storage Mode, ***51***
Storage Requirements, ***3***
Storage Space, 15, 20
Storage, Disk, ***16***
Storage, Space, 15, 20
Stored-Program Computer, 26
Storing of Data, 16
Storing, Data, 16
Straight Line, ***19***
Straight Plane, ***19***
Strength, 98, 213
Structure of Model, 17
Structure of Supercomputer, 89
Structure, Data, 210
Structure, Geologic, ***19***
Structure, Grid, 16
Structure, Language, 210
Structure, Model, 17
Structured Data Set, 60
Study, 210
Stuttgart, 97
Sub-Unit, 38
Subcommand, 29
Subcommands, 33
Subdivision, 23, 91, 99
Subdivision of Tasks, ***50***
Subdivision, Model, ***4***
Subdivision, Problem, ***4***
Subdomain, 94–97
Subdomain Distribution, 56
Subdomains, ***56***
Submodel, 95
Subordinate Processing Unit, ***39***
Subordinate Processor, 39
Subproblem Size, 95
Subproblems, ***4***
Subset, 99
Subset of Services, ***51***
Substance, 11
Substitution, 44
Subtraction of Congruences, ***148***
Supercomputer, 48, 50–53, ***56****, 90, 91, 97, 98*
Supercomputer Architecture, 89
Supercomputer Environment, 23
Supercomputer Structure, 89
Supercomputer System, ***53***
Supercomputers, ***38****,* ***40****,* ***42***
Superpipelining, ***29****, 30*
Superscalar Architecture, 31
Superscalar Architectures, 36
Superscalar Implementations, ***32***
Superscalar Microprocessor, ***30***
Superscalar Processors, ***30***
Superscalar Technique, ***30***
Superscalar Techniques, 32
SuperSPARC Processor, ***30***
Supervising Unit, 30
Supervision of Work, 63
Supervisor, ***4****,* ***37****, 65*
Supply Facility, ***53***
Support Functions, 58
Support of Shared Memory, ***48***
Surface, ***11***
Switch, ***46****, 47,* ***48****,* ***53***
Switch Connection, ***46***
Switch-Based Shared Memory System, ***44***
Switch-Based SM System, ***44***
SX-8, ***48****, 48*
Symmetrical MPS, ***42***
Symmetrical Multiprocessing Architecture, ***42***
Symmetrical Multiprocessor Environment, ***43***
Symmetry, ***12****, 16, 20*
Symmetry Effect, 16
Symmetry, Effect, 16
Synchronization, ***40****,* ***46****, 61, 62*
Synchronization of Laborers, 62
Synchronization Request, 61
Synchronization, Laborers, 62
Synchronizing, ***40***
Synchronous Execution, ***39***
Synonym, ***52***
System Configuration, ***53***
System Hierarchy, 48
System of Denomination, ***1***
System of Grid Nodes, 21
System Performance, 33
System, Equation, 23
System, Reactive, ***10***
Systole, ***40***
Systolic Array, ***40***

Table of Computer Classes, ***34***
Table, Water, 16
Tank, ***10***

Tank, Oil, ***10***
Tape Drive, 32
Task, 33, 43
Task, Administrative, 32, 60
Tasks, 34
Taxonomy, 40
Taxonomy after Flynn, ***34****, 40, 41*
Taxonomy of Parallel Computers, ***34***
TBytes, ***149***
Team, ***44****, 44*
Technique of Superpipelining, 33
Technique, Modeling, 12
Technological Improvement, ***38***
Telecommunications Infrastructure, ***51***
Telescope, ***52****, 52*
Temporal Parallelism, ***33***
Tendency of Branching, ***38***
TeraBytes, ***149***
Term, High-Order, 213
Term, Low-Order, 213
Terminal, ***33****,* ***50***
Terrabyte, ***53***
Terraflop, ***53***
Terrain, 91
Test, 210
Test Data, 100
Test Model, 97
Test, Detailed, 210
Testing, 100
Tetrahedron, ***104***
Theoretical Computer Science, 96
Theoretical Performance, ***53***
Theory, 99, 209
Theory, Complexity, 210
Thesis Software, 89
Theta Notation, ***210****, 212*
Thinking Machines, ***40***
Thread, 58
Throughput, 27, ***33***
Thumb, Rule, ***16***
Time Approximation, 60
Time Dependency, 59
Time for Communication, 65
Time for Cooperation, 65
Time for Digging, 65
Time for Meal, 65
Time Loss, ***44***
Time of Communication, 65
Time of Computation, 62
Time of Cooperation, 65
Time of Dormancy, 94
Time of Inactivity, 94
Time of Recreation, 65
Time Saving, 16, 90, 91
Time Slot, 93
Time Slot for Communication, 93
Time Slot for Computation, 93
Time Step, ***20****,* ***40****, 65*
Time Step, Size, ***20***
Time Unit, 65
Time, Access, 210
Time, Accumulated, 93
Time, Asymptotic, 210
Time, Communication, ***61****, 65, 92, 94*
Time, Computation, 4, ***15****, 20, 66, 92, 95*
Time, Computing, 16, 90, 91
Time, Coordination, 65
Time, Digging, 65, 91
Time, Dormancy, 94
Time, Idle, 32, 93, 94
Time, Inactivity, 94
Time, Polynomial, 99
Time, Processing, 16
Time, Recreation, 65
Time, Saving, 16
Time, Slot, 93
Time, Startup, 64
Time, Step, ***20****, 65*
Time, Transfer, 64
Time, Waiting, 61
Time-Consuming Experiment, ***1***
Time-Dependent Aspect, 59
Time-related Matter, 64
Tool, 23, 96, 97, 99, 209, 210
Tool, Scientific, ***1***
Toolkit, ***51****, 51*
Top 500, ***52***
Top500 List, 50
Topology, ***47***
Total Execution Time, ***61****, 61, 63*
Total of Operations, 65
Transaction Volume, ***43***
Transfer of Message, 55
Transfer Time, 64
Transformation, 9
Transformation Process, 11
Transient Process, 9
Transition, 20
Transition Area, ***20***
Transition, Area, ***20***
Transition, Gradual, ***20***
Translation, 22, 94
Translation Process, 94
Translator Program, ***31***
Transmission of Information, 27
Transport of Data, 27
Transport of Information, 32
Transport of Physical Signals, ***45***
Travel Agency, ***33***
Travelling Salesman, 100
Trial and Error, 100
Triangle, ***104****, 104*
Triangular Area, 104
Triangular Element, 20
Trick, ***16***
Trick, Effective, ***16***
Trigger Services, ***51***
Trillion, ***1***

Trillions of Operations, 1
Turing Machine, 99
*Twins, **42***
*Two-story Steel Construction, **53***
Type of Computer Architecture, 58
Type of Element, 19
*Type of Hardware, **42***
Type of Input, 60
Type of Material, 19, 20
*Type of Parallelism, **33***
Type of Problem, 55
Type, Element, 19
Type, Material, 19, 20
*Types of Architectures, **40***

*UltraSPARC Processor, **30***
*UMA, **45***
*Underlaying Network, **55***
Underlying Algorithm, 210
Underlying Hardware, 45
*Unethical Experiment, **1***
*Uniprocessor Architectures, **36***
*Unit, Computational, **3***
Unit, Input, 26
Unit, Output, 26
Unit, Processing, 65
*United States, **2**, 3*
Universal Solution, 210
*Universe, **1***
*Universe Research, **1***
*University of Berkeley, **52***
*Unskilled Laborer, **3***
uperpipelining, 33
Upper Bound, 209, 211
Upper Boundary, 65
*Usage of Cluster, **50***
User, 33, 52
*User Computation, **29***
*User Program, **43***
User Request, 33
*User-Access System, **51***
*Utilization, **65**, 65, 66, 89, 99*
Utilization Factor, 65
Utilization of Processor, 63

Validation, 9
*Value Assignment, **18***
*Value, Assignment, **18***
*Value, Change, **14***
Value, Definite, 19
Value, Identical, 19
Variable, 10, 20, 45
*Variable, Altered, **45***
Variable, Environment, 58
*Variable, Field, **19***
*Variable, Shared, **45***
Variance, 210
*Vector, **37***
Vector Computer, 38, 42
*Vector Computers, **37, 38***
*Vector Instructions, **38**, 38*
Vector Machine, 38, 97
*Vector Machines, **37***
*Vector Processing, **37***
*Vector Processing Systems, **38***
*Vector Processor, **38***
*Vector Processor Connection, **41***
*Vector Processors, 36, **37**, 41, 42*
*Vector Registers, **38***
*Vector Supercomputer, **53***
*Vector Supercomputer System, **53***
*Vector-Type Arithmetic Processor, **53***
*Vectorization, **38***
Version, Parallel, 63
*Vertex, **2**, **12**, 15, 19, **104**, 104*
Vertices, Set, 104
*Very Large Instruction Word, **30***
View, Abstract, 11
*Virtual Cluster, **51***
*Virtual Computer, **51***
*Virtual Shared Memory System, **48***
*Virtualizing Computing Resources, **51***
*Virual SM System, **48***
Virus, 100
*VLIW Machine, **31***
*VLIW Processors, **30***
*VLIW Technique, **31***
*VLIW, Advantage, **30***
*VLIW, Disadvantage, **30***
*Volume of Transaction, **43***
Von Neumann, 26
*Von Neumann Architecture, **25**, 26, 35, 36, 38*
*Von Neumann Design, **25***
Von Neumann Machine, 26, 35

Waiting Time, 61
*Waste of Manpower, **44***
Waste of Money, 90
Waste of Resources, 90
Waste of Time, 90
Waste of Work Power, 90
*Waste Water, **4***
Waste, Money, 90
Waste, Resources, 90
Waste, Time, 90
*Water Extraction, **13***
*Water Level, **13***
Water Table, 13, 16
Water, Table, 16
*Water-Saturated Zone, **10***
Weakness, 98
Weather, 2, 3
*Weather Calculation, **1, 2***
Weather Example, 3
*Weather Forecast, **1**, 4, 56*
*Weather Influence, **52***
Weather Predicition, 3
*Weather Prediction, **2***
*Weather, Forecast, **1***

Weather-Predicting Computer, ***3***
Week, 3
Well, ***13****, 16*
Well, Oberservation, ***13***
Well, Production, ***13***
Well, Pumping, 14, 16
Well-Equipped Home Computer, ***58***
Width of Data Bus, ***27***
Windows, ***50***
Wisdom, 19
Wolfpack Cluster, ***50***
Words of Memory, ***3***
Work, 65
Work Area, 90
Work Coordination, 62
Work Day, ***36***
Work Directing, 63
Work Distribution, 91
Work Group, ***4***
Work Load, ***56***
Work Load Balancing, 56
Work of Excavation, 62
Work Power, Waste, 90
Work Supervision, 63
Workday, 90
Worker, 90
Workforce, ***3***
Workhorse, 94
Working Pace, ***37***
Workload, ***43****, 89, 91, 92*
Workload Distribution, 90, 91
Workspace, 64
Workspace of Excavation, 64
Workstation, ***51***
World, 4
World of Computer Science, ***53***
World of HPC, ***52***
World of Mathematics, ***53***
World of Science, ***53***
World, 3D, 11
World, whole, ***3***
Worst Case, 209, 210
Wrapper, 57, 96, 97, 104
Wrapper Development, 97
Wrapper Development Process, 97
Wrapper Improvement, 89
Wrapper Optimization, 89
Wrapper Program, 57, 89, 96, 104
Wrapper Software, 89
WWW, ***52***

Yokohama, ***42***

Zahlentheorie, 210
Zahlentheorie, Analytische, 210
Zone, Overlapping, ***18***
Zone, Water-Saturated, ***10***