# Untersuchungen über objektorientierte Design-Patterns für massiv-parallele Teilchensimulationsverfahren anhand von Smoothed Particle Hydrodynamics

**Dissertation**
der Fakultät für Informatik
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Dipl. Phys. **Stefan Gathmann genannt Hüttemann**
aus Tübingen

**Tübingen**
**2001**

| | |
|---|---|
| Tag der mündlichen Qualifikation: | 19. Dezember 2001 |
| Dekan: | Prof. Dr. Andreas Zell |
| 1. Berichterstatter: | Prof. Dr. Wolfgang Rosenstiel |
| 2. Berichterstatter: | Prof. Dr. Herbert Klaeren |

Diese Arbeit ist meiner Mutter gewidmet.

# Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Durchführung der vorliegenden Arbeit unterstützt haben. Mein besonderer Dank gilt dabei Herrn Prof. Dr. W. Rosenstiel für seine großzügige Unterstützung und wertvolle Anleitung.

Bei meinen Kollegen Till Bubeck, Marcus Ritt und Michael Hipp möchte ich mich für die kooperative und freundschaftliche Zusammenarbeit im Projekt bedanken. Ein weiterer Dank geht an die von mir betreuten Diplomanden Andreas Nagel, Frank Heuser, Sven Ganzenmüller und Thorsten Laib.

Mein herzlichster Dank gilt jedoch meiner Frau Brenda für Ihre Unterstützung und Ihr Verständnis in den arbeitsintensiven Phasen der zurückliegenden Monate und Jahre.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Algorithmenverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

Seit der Erfindung von *Elektronenrechnern* konnten Simulationsverfahren physikalischer Vorgänge auf Computern und die Entwicklung von Computer-Hardware und -Software voneinander profitieren. Ein neues numerisches Simulationsverfahren brachte und bringt die leistungsfähigsten Computer an ihre Grenzen. Neue, leistungsfähigere Computer motivieren dazu, aufwendigere Simulationen zu entwickeln. Aufwendigere Simulationsverfahren brauchen neue Vorgehensweisen bei der Softwareentwicklung von wissenschaftlichen Anwendungen.

Numerische Simulationsverfahren sind ein nicht mehr wegzudenkendes Bindeglied zwischen Experiment und Theorie in der modernen Physik. Durch Computersimulationen können Theorien auf quantitative Weise mit Experimenten verglichen werden. Es können *virtuelle Experimente* in Raum und Zeit durchgeführt werden, die in einem realen Experiment technisch nicht möglich sind.

Die vorliegende Arbeit untersucht die Parallelisierung und Implementierung eines aktuellen Simulationsverfahrens, das zur Lösung von partiellen Differentialgleichungen aus dem Bereich der Hydrodynamik eingesetzt wird. *Smoothed Particle Hydrodynamics* (SPH) ist ein gitterfreies Simulationsverfahren zur Lösung von hydrodynamischen Bewegungsgleichungen mit starken Dichtegradienten und freien Rändern. Durch seine Eigenschaft, ohne festes Interpolationsgitter zu arbeiten, stellt SPH eine besondere Herausforderung bei der Parallelisierung dar. Im Gegensatz zu gitterbasierten Gleichungslösern bewegen sich die Stützstellen — SPH-Teilchen genannt — in jedem Zeitintegrationsschritt zueinander. SPH, das für astrophysikalische Problemstellungen entwickelt wurde, wird mittlerweile auch auf andere Fragestellungen aus dem Bereich der Hydrodynamik angewendet.

## 1.1 Motivation

Schon die sprichwörtlichen *astronomischen Zahlen* lassen die von astrophysikalischen Simulationsprogrammen an die Rechnerressourcen gestellten Anforderungen erahnen. Um bei den Simulationen der physikalischen Modelle durch die SPH-Methode auf eine aussagekräftige numerische Genauigkeit zu kommen, ist es notwendig, mit hohen SPH-Teilchendichten, also mit großen SPH-Teilchenzahlen, zu rechnen. Eine Erhöhung der SPH-Teilchendichte bei den Simulationsprogrammen erfordert mehr Hauptspeicher zur Berechnung. Sobald zur Programmlaufzeit Teile des Hauptspeichers auf ein anderes Speichermedium (i.A. die Festplatte des Rechners) ausgelagert werden müssen (*engl.: paging*), steigt die Zugriffszeit auf die Daten um eine Größenordnung — vom *ns*-Bereich für Hauptspeicherzugriffe in den *ms*-Bereich für Festplattenzugriffe. Bei einer Programmlaufzeit für eine gängige SPH-Simulation von mehreren Tagen hätte dies einen Anstieg auf eine Programmlaufzeit von mehreren *Jahren* zur Folge. Dieses Problem kann nur durch die Verwendung von Rechnern mit größerem Hauptspeicher gelöst werden.

Um durch die Simulationen quantitativ näher an die physikalische Wirklichkeit — d.h. die experimentellen Ergebnisse — heranzukommen, müssen numerisch aufwendigere SPH-Terme implementiert werden. Die Berechnung dieser SPH-Terme erfordert mehr Rechenleistung. Um die benötigte Rechenleistung zur Verfügung stellen zu können, müssen immer leistungsfähigere CPU verwendet werden.

Diesen Anforderungen kann man nur durch Kombination mehrerer Hauptspeicher- und CPU-Elemente, also durch massiv-parallele Rechnerarchitekturen, gerecht werden. Beispiele für solche Rechner sind die NEC SX-4 oder die Cray T3E. Selbst wenn durch verbesserte Hardware (GHz-Prozessor, GB-RAM) ein Simulationslauf auf einem seriellen Rechner ablaufen könnte, haben Astrophysiker bereits Fragestellungen zur Hand, die auch die Kapazitäten dieses Rechners sprengen werden. Parallelrechner werden immer ein aktuelles Thema für technisch-wissenschaftliche Anwendungen bleiben. Die Fragestellung der vorliegenden Arbeit könnte also kurz überschrieben werden mit: Wie programmiert man SPH-Verfahren auf Parallelrechnern?

Zur Programmiermethodik für massiv-parallele Rechner haben sich mittlerweile Standardbibliotheken für Parallelisierungsprimitiven wie *MPI* oder *OpenMP* durchgesetzt. Die Implementierung von komplexen Simulationsverfahren wie SPH mit diesen Standards verlangt eine genaue Kenntniss der eingesetzten Parallelrechner und Parallelisierungsprimitiven. Daraus ergibt sich das Problem, dass zwei komplexe Fragestellungen, die der Implementierung von SPH-Verfahren und die der effizienten Parallelisierung auf verschiedenen Parallelrechnern, in einem Programm zusam-

mengefügt werden müssen. Bestehende Simulationsprogramme und Bibliotheken bieten für SPH-Verfahren dabei keine geeignete Unterstützung. Der Programmierer muss Experte in SPH und Experte für Parallelisierung sein. Experten für numerische Simulationsverfahren müssen demnach gleichzeitig auch Experten für Parallelisierungskonzepte sein. Ein geeignetes Programmiermodell sollte diesen Konflikt lösen können und eine kooperative Zusammenarbeit von verschiedenen Experten erleichtern.

Die Problematik, wie geeignet programmiert werden sollte, ist von allgemeinem Charakter für die Softwareentwicklung. Es wurden dafür verschiedene Lösungsstrategien entwickelt. Angefangen von der Strukturierung von Programmen durch *Prozeduren* oder *Subroutinen* über die Programmierung mit einem *modularen* Konzept bis hin zu *objektorientierter* Programmierung (OOP). Des Weiteren sind noch Programmierparadigmen wie das der *funktionalen* Programmierung oder *Logik*-Sprachen zu nennen.

In der vorliegenden Arbeit hat die Entscheidung für die *objektorientierte* Programmierung als Programmierparadigma zur Implementierung für SPH-Verfahren pragmatische Gründe:

- In dem Entwicklungszweig der *prozeduralen*, *modularen* und *objektorientierten* Paradigmen bietet die OOP die umfangreichsten Strukturmöglichkeiten zur Programmierung an.

- Es hat sich bereits gezeigt, dass die Programmierung mit Prozeduren als Strukturelementen nicht ausreicht, um erweiterbare und wiederverwendbare Programme zu schreiben. Es existieren für viele Simulationsverfahren zwar Bibliotheken, aber durch deren Verwendung werden im Allgemeinen Lösungen erzeugt, die selbst nicht gut erweiterbar und wiederverwendbar sind.

- Auf den für die hier besprochenen Simulationsverfahren eingesetzten Rechnern gibt es Compiler für objektorientierte und prozedurale Sprachen (`C++` und `FORTRAN`), nicht aber für Logik- oder funktionale Sprachen.

- Der Überblick über bestehende Simulationspakete (siehe Abschnitt 3.2) zeigt, dass objektorientierte Konzepte zunehmend Verwendung finden. Eine Auseinandersetzung darüber, *wie* für wissenschaftliche Anwendungen objektorientiert programmiert werden sollte, ist ein aktuelles Thema.

Im Bereich der objektorientierten Programmierung haben sich Entwurfsmuster (*engl.: design pattern*) zur Beschreibung von bewährten objektorientierten Lösungen

verschiedener Probleme durchgesetzt.[1] Design-Pattern-Kataloge fassen dabei verschiedene Design-Pattern thematisch sortiert zusammen. Für die in der vorliegenden Arbeit untersuchte Problemstellung der Parallelisierung von SPH-Verfahren gibt es noch keine speziellen Design-Pattern, die direkt angewendet werden könnten. Im Rahmen der Auseinandersetzung mit der OOP für technisch-wissenschaftliche Anwendungen sollten Design-Pattern auf ihre Verwendbarkeit hin untersucht werden.

## 1.2 Zielsetzung

Aus der in Abschnitt 1.1 beschriebenen Problematik, die zwei Fragestellungen der Implementierung von SPH-Verfahren und der effizienten Parallelisierung auf verschiedenen massiv-parallelen Rechnern zu vereinen, zusammen mit der Rahmenbedingung, SPH-Verfahren durch objektorientierte Methoden zu implementieren, können folgende Zielsetzungen für die vorliegende Arbeit abgeleitet werden:

#### Entkopplung von SPH-Verfahren und Parallelisierungsmethoden

Die Aufgabenstellung der Parallelisierung eines SPH-Verfahrens und die der Implementierung der SPH-Terme selbst muss voneinander entkoppelt werden können. Damit muss es möglich sein, ein neues SPH-Verfahren zu implementieren und dabei bereits implementierte Parallelisierungskonzepte wiederzuverwenden. Eine Optimierung der Parallelisierungskonzepte darf keine Auswirkungen auf die implementierten SPH-Verfahren haben.

Welche Programmiermethodik erlaubt es, die Problemstellungen der Implementierung eines SPH-Verfahrens und die der Parallelisierung so voneinander zu entkoppeln, dass sich die jeweiligen Spezialisten auf diesen Gebieten voll auf ihre eigentliche Arbeit konzentrieren können und trotzdem ein gemeinsames Programm entsteht?

Für die OOP sind Design-Pattern zu finden, die für diese Fragestellung eine Lösung angeben.

#### Wiederverwendbare Klassenbibliothek für paralleles SPH

Einmal gefundene Lösungen für Teilprobleme müssen in einer wiederverwendbaren Form niedergelegt werden können. Neu entwickelte Lösungen müssen durch das Einhalten eines Design-Pattern eine wiederverwendbare Form annehmen.

[1] Im Folgenden wird der Begriff *Design-Pattern* für Entwurfsmuster verwendet.

Welche Programmstruktur/Bibliothek erlaubt es, einmal gefundene Lösungen in neuen SPH-Programmen effizient wieder einzusetzen und für neue Lösungen einen Rahmen vorzugeben, nach dem diese Lösungen in einer wiederverwendbaren Form implementiert werden können?

#### Anpassung an verschiedene Parallelrechner

Das Design der Bibliothek muss so geartet sein, das eine Portierung auf verschiedene Rechnerarchitekturen ohne Veränderung in den Simulationscodes möglich ist. Einmal parallelisierte SPH-Verfahren müssen mit der Bibliothek auf verschiedenen Parallelrechnerplattformen effizient ausführbar sein. Die Anpassung an die Parallelrechner geschieht unabhängig von der Implementierung des eigentlichen SPH-Verfahrens.

#### Rahmenprogramm für parallele SPH-Programme

Durch das Design muss es möglich sein, den anfallenden Problemstellungen und Aufgaben klar definierte Verantwortungsbereiche zuweisen zu können. Dadurch muss ein Rahmenprogramm entstehen, bei dem Experten aus verschiedenen Fachgebieten ihre jeweiligen Lösungen einbringen können und trotzdem eine einheitliche, wiederverwendbare und erweiterbare Klassenbibliothek entsteht.

Besonders interessant ist diese Fragestellung in Projekten, in denen die Mitarbeiter öfter wechseln, wie es beispielsweise im universitären Bereich der Fall ist. Dabei ist es wichtig, dass sich die neuen Mitarbeiter schnell und gezielt in die vorhandene Programmierumgebung einarbeiten können, sodass sie sich rasch auf ihre jeweilige Aufgabe konzentrieren können.

Gibt es eine einfache und leicht erlernbare Struktur für ein SPH-Programm, sodass ein Experte gezielte Änderungen einbringen kann ohne Auswirkungen auf den Rest des Programms?

Die Dokumentation dieses Rahmenprogramms muss in einem einheitlichen, möglichst standardisierten Format geschrieben werden. Für OOP bietet sich eine Dokumentation in UML an.

#### Anwendung der Methoden: Effiziente Parallelisierung von SPH-Verfahren

Nicht zuletzt bleibt noch die Frage nach der Optimierung und Anwendbarkeit der in der vorliegenden Arbeit dargestellten Verfahren.

Die objektorientierten Design-Pattern für paralleles SPH müssen sich an konkreten SPH-Problemstellungen als effizient beweisen.

## 1.3 Aufbau der Arbeit

Die vorliegende Arbeit ist in zwei Teile gegliedert. Den ersten Teil bilden die Kapitel 2 und 3, in denen Grundbegriffe und aktuelle Arbeiten aus dem Gebiet des parallelen objektorientierten wissenschaftlichen Rechnens dargestellt werden. Im zweiten Teil, Kapitel 4 bis Kapitel 6, werden die eigenen Arbeiten des Autors beschrieben. Ein Ausblick im Kapitel 7 auf mögliche weitere Arbeiten, die sich aus den Untersuchungen der vorliegenden Arbeit ergeben könnten, bildet einen Abschluss.

**Kapitel 2** vermittelt einen Überblick über das Teilchensimulationsverfahren SPH (*engl.: Smoothed Particle Hydrodynamics*) und über den aktuellen Stand der parallelen Hardware- und Software-Technologie sowie über die Grundbegriffe der objektorientierten Programmierung mit Design-Pattern. Das Grundlagen-Kapitel behandelt diese Themen nicht vollständig, sondern gibt nur eine Begriffserklärung der in der vorliegenden Arbeit verwendeten Fachbegriffe.

**Kapitel 3** ist eine Zusammenfassung der aktuellen Arbeiten anderer Autoren. Dabei werden Implementierungen paralleler SPH-Verfahren analysiert und bewertet. Ein Überblick über parallele objektorientierte Programmpakete zur Lösung von partiellen Differentialgleichungen, wie der Navier-Stokes-Gleichung, zeigt die zurzeit laufende Diskussion über die Einsatzmöglichkeiten von objektorientierter Technologie für wissenschaftliche Anwendungen auf. Eine Darstellung der Entwicklung von objektorientierten Design-Pattern schließt dieses Kapitel.

**Kapitel 4** Bevor in Kapitel 5 die Frage erörtert wird, wie SPH mit objektorientierten Methoden parallelisiert werden kann, werden in diesem Kapitel Grundlagen zur Parallelisierung von SPH untersucht. Bestehende Parallelisierungen von SPH-Verfahren gehen von speziellen Vorbedingungen, SPH-Methoden oder sogar von Spezialhardware aus. Eine allgemeine Darstellung der nebenläufig ausführbaren Teile sowie eine Diskussion über die möglichen Parallelisierungskonzepte für Maschinen mit verteiltem und für Maschinen mit gemeinsamem Hauptspeicher werden in diesem Kapitel erarbeitet. Außerdem werden SPH-Verfahren, die ein nichttriviales Problem angehen, auf ihre allgemeine Parallelisierbarkeit hin untersucht.

**Kapitel 5** gibt eine Antwort auf die Frage, wie SPH-Verfahren parallelisiert werden können. Hier werden Methoden entwickelt, um eine effiziente Parallelisierung von SPH mit objektorientierten Design-Pattern auf massiv-parallelen Rechnerarchitekturen implementieren zu können. Dabei werden die im Abschnitt 1.2 beschriebenen Ziele der vorliegenden Arbeit umgesetzt.

**Kapitel 6** beschreibt die Implementierungen von SPH-Verfahren mit den in der vorliegenden Arbeit entwickelten Methoden. Dabei werden massiv-parallele Rechnerarchitekturen wie die NEC SX-4, die Cray T3E und das Kepler-Cluster der Universität Tübingen eingesetzt. Diese Darstellung von Produktionscodes umfasst Simulationen zu Themen aus der Astrophysik (Simulationen von Neutronensternen) und der Automobiltechnik (Dieseleinspritzung). Hier wird die Leistungsfähigkeit der in der vorliegenden Arbeit entwickelten Methoden demonstriert, mit denen SPH-Verfahren auf der NEC SX-4 mit bis zu 90% paralleler Effizienz implementiert wurden. Auf der Cray T3E wurde bei 512 Knoten noch über 65% Effizienz erreicht. Die Implementierungen auf dem Kepler-Cluster erreichen eine Effizienz von 62% auf 82 Knoten mit 200.000 SPH-Teilchen.

**Kapitel 7** fasst die Ergebnisse der vorliegenden Arbeit zusammen und gibt einen Ausblick auf noch offene Fragen.

**Anhang** Im Anhang ist ein im Rahmen der vorliegenden Arbeit entstandener, Design-Pattern-Katalog für objektorientierte parallele SPH-Verfahren aufgeführt.

# Kapitel 2

# Grundlagen

Der interdisziplinäre Charakter der Arbeit verlangt zum Verständnis, dass aus den verschiedenen Bereichen grundlegende Begriffe eingeführt und erläutert werden. Es wird dabei nur auf die für die vorliegende Arbeit wichtigen Aspekte eingegangen. Für Interessierte wird in den einzelnen Abschnitten auf Literatur zu den jeweiligen Themen verwiesen.

Im Abschnitt 2.1 werden Grundlagen zur Simulation von physikalischen Prozessen anhand der hier verwendeten Teilchensimulationsmethode *Smoothed Particle Hydrodynamics* (SPH) erläutert.

Eine Zusammenfassung zu den Grundlagen über parallele Hardware und den Softwareentwicklungsmethoden zur parallelen Programmierung wird im Abschnitt 2.2 gegeben.

Erläuterungen zu Begriffen aus der objektorientierten Programmierung folgen im Abschnitt 2.3.

## 2.1 Simulation physikalischer Prozesse

Simulationen spielen in der modernen Physik eine zentrale Rolle. Die Vorgehensweisen in den Naturwissenschaften verlangen einen ständigen Vergleich zwischen Theorie und Experiment. Experimente können ohne eine gute Theorie weder entwickelt noch verstanden werden. Theorien ohne experimentelle Bestätigung sind wertlos.

Der Physiker wird experimentelle Ergebnisse mit theoretischen Voraussagen (Berechnungen) vergleichen wollen. Nur sehr wenige nichttriviale physikalische Fragestellungen lassen sich aber mit mathematischen Methoden exakt lösen. Ein bekanntes Beispiel ist das zweidimensionale Ising-Modell [2]. Die meisten theoretischen Voraussagen müssen mit Hilfe von numerischen Methoden näherungsweise bestimmt werden.

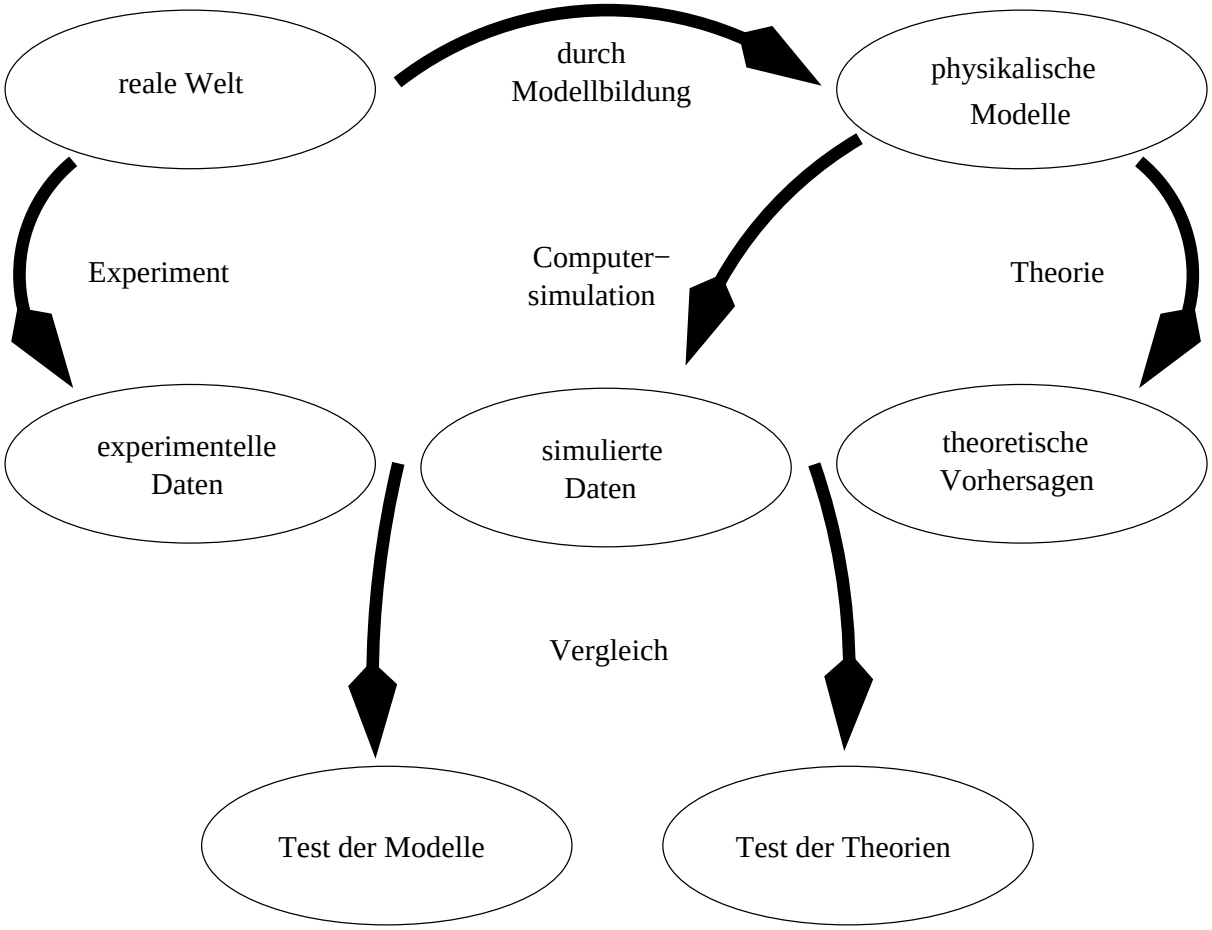



Abbildung 2.1: Simulationen sind das Bindeglied zwischen Theorie und Experiment in der Physik (nach [2]).

Gleichungsysteme mit tausenden von Variablen können aber nicht auf dem Papier gelöst werden. Computer und Simulationsprogramme schließen die Lücke zwischen Experiment und Theorie (siehe Abbildung 2.1). Computersimulationen ermöglichen es, den Ausgang eines Experiments quantitativ vorherzusagen, und lassen damit einen direkten Vergleich zwischen Theorie und Experiment zu . Auch bieten Computersimulationen die Möglichkeit, ein sonst sehr teures Experiment erst einmal virtuell zu erproben.

In den in dieser Arbeit beschriebenen Problemstellungen der Physik werden Computersimulationen verwendet, um „Gedankenexperimente“ durchzuführen, die auf experimenteller Basis so nicht möglich wären: Experimente mit Sternen und Galaxien.

### 2.1.1 Hydrodynamische Grundgleichungen

Die Arbeit bezieht sich auf Simulationsverfahren, deren physikalische Modelle alle auf Methoden der klassischen Hydrodynamik aufbauen. Es wird ein schematischer Überblick über die notwendigen Grundbegriffe der Hydrodynamik gegeben. Der Überblick kann nicht vollständig sein, auch werden die Sachverhalte hier vereinfacht dargestellt, da die notwendigen mathematischen Vorkenntnisse hier nicht vorausgesetzt werden sollen. Dennoch sollten die Grundbestandteile einer hydrodynamischen Gleichung verstanden werden. Für eine mathematisch und physikalisch korrekte Darstellung wird auf einführende Literatur der Physik wie [34] verwiesen.[1]

#### Die Bewegungsgleichung

Hydrodynamik ist Bestandteil der *klassischen* Mechanik. Unter klassischer Physik versteht man die Physik im *newtonschen* Weltbild in Abgrenzung zum *relativistischen* Weltbild von Einstein und der Quantenmechanik.

In der klassischen Mechanik interessiert man sich dafür, wie sich eine Masse $m$ durch den Raum bewegt, d.h., man möchte wissen, zu welchem Zeitpunkt $t$ sich die Masse $m$ an welchem Ort $\vec{r}$ im Raum befindet. Die Funktion $\vec{r}(t)$ nennt man die Bahnkurve einer Masse $m$. Um die Bahnkurve einer Masse $m$ berechnen zu können, braucht man eine *Bewegungsgleichung*. Die Bewegungsgleichung ist die Grundgleichung der klassischen Mechanik. Sie ist eine Differentialgleichung in der Zeit $t$, und die Lösung dieser Gleichung gibt uns die Bahnkurve $\vec{r}(t)$.

Die Grundgleichung von Newton lautet:

$$\vec{F}(t) = m\,\frac{d^2}{dt^2}\vec{r}(t) = m\,\vec{a}(t) \quad (= \frac{d}{dt}\vec{p}(t)) \tag{2.1}$$

Dabei ist $m$ die Masse eines beobachteten Objekts, $\vec{F}(t)$ sind die zu einem Zeitpunkt $t$ auf das Objekt einwirkenden Kräfte und $\vec{a}(t)$ die zu einem Zeitpunkt $t$ bei dem Objekt auftretenden Beschleunigungen. (Das Produkt aus Masse und Geschwindigkeit ist der Impuls einer Masse, womit die rechte Seite von Gleichung 2.1 auch durch die zeitliche Ableitung des Impulses $\vec{p}$ beschrieben werden kann.) Dabei wird der Zusammenhang zwischen der Bahnkurve $\vec{r}$ einer Masse $m$ und den auftretenden Beschleunigungen $\vec{a}$ — d.h. Richtungsänderungen jeder Art — mathematisch durch eine Zeitableitung beschrieben.

Diese klassische Bewegungsgleichung besagt also, wenn alle auf eine Masse $m$ einwirkenden Kräfte $\vec{F}$ zu jedem Zeitpunkt $t$ bekannt sind, kann durch die Lösung der Differentialgleichung 2.1 in der Zeit die Bahnkurve $\vec{r}(t)$ errechnet werden.

[1] Hydrodynamik ist ein fortgeschrittenes Thema der Physik, deshalb setzt auch *einführende* Literatur zur Hydrodynamik bereits Grundlagenwissen über Methoden der Physik voraus.

Um einen physikalischen Vorgang zu beschreiben, muss der Physiker also ein Modell des Vorgangs entwickeln und dann eine Differentialgleichung der Form 2.1 aufstellen. Dazu müssen die für diesen Vorgang relevanten Kräfte identifiziert und Funktionen gefunden werden, die die auftretenden Kräfte beschreiben. Auch die Masse $m$, die in diesem Vorgang eine Rolle spielt, muss sinnvoll beschrieben werden. Nachdem die Differentialgleichung gefunden wurde, kann diese mit mathematischen Verfahren gelöst werden.

### Die Navier-Stokes-Gleichung

Die Hydrodynamik beschäftigt sich nun mit der Bewegung — d.h. der Strömung — von Fluiden. In der lagrangeschen Beschreibung (siehe [34],[52]) werden dazu kleine, mit der Strömung mitbewegte Fluidelemente betrachtet. Die Fluidelemente sollen eine feste Masse besitzen und so klein sein, dass die Feldgrößen[2] sich innerhalb ihres Volumens nicht ändern. Die Grundgleichung der Hydrodynamik (für viskose Fluide) ist die Navier-Stokes-Gleichung. Die Lösung dieser Gleichung beantwortet alle physikalisch interessanten Fragen über ein (viskoses) Fluid.

Im lagrangeschen Bild in Komponentendarstellung besitzt die Navier-Stokes-Gleichung die Form (siehe [34])

$$\underbrace{\rho\frac{dv_\alpha}{dt}}_{\mathcal{A}} = \underbrace{-\frac{\partial p}{\partial x_\alpha}}_{\mathcal{B}} + \underbrace{\frac{\partial T_{\alpha\beta}}{\partial x_\beta}}_{\mathcal{C}} + \underbrace{\rho f_\alpha}_{\mathcal{D}} \ . \tag{2.2}$$

Es wird hier angenommen, dass alle Größen Funktionen der Zeit und des Orts sind, also: $\rho \equiv \rho(\vec{r},t)$. Mit $\rho$ als Massendichtefunktion, $v_\alpha$ dem Geschwindigkeitsfeld des Fluids, $p$ dem Druckterm und $f_\alpha$ der Summe aller externen Kräfte auf ein Fluidelement. Dabei gilt die einsteinsche Summenkonvention.[3] $T_{\alpha\beta}$ heißt viskoser Spannungstensor und hat im allgemeinen Fall die Form

$$T_{\alpha\beta} = \eta\left(\frac{\partial v_\alpha}{\partial x_\beta} + \frac{\partial v_\beta}{\partial x_\alpha} - \frac{2}{3}\delta_{\alpha\beta}\frac{\partial v_\gamma}{\partial x_\gamma}\right) + \zeta\,\delta_{\alpha\beta}\frac{\partial v_\gamma}{\partial x_\gamma}, \tag{2.3}$$

[2]Die physikalischen Größen wie z.B. die Geschwindigkeit $\vec{v}$ gehen in der Kontinuumsmechanik über in ortsabhängige Funktionen (Felder) $\vec{v}(\vec{r})$, da die Geschwindigkeit einer Flüssigkeit an jedem Punkt im Raum eine andere sein kann (z.B. Wirbel in einem Fluss oder Wasser aus einem Wasserhahn sind anschauliche Beispiele dafür).

[3]Die einsteinsche Summenkonvention summiert über doppelt auftretende Koordinatenindizes auf.

Beispiel:

$$\text{aus} \quad \frac{\partial v_\alpha}{\partial x_\alpha}$$

$$\text{wird} \quad \sum_{\alpha=1}^{3}\frac{\partial v_\alpha}{\partial x_\alpha}.$$

wobei $\eta$ der Koeffizient der Scherviskosität und $\zeta$ der Koeffizeint der Volumenviskosität ist. Für eine detaillierte Herleitung der Gleichungen 2.2 und 2.3 siehe [34].

Zum besseren Verständnis der Ausführungen der folgenden Kapitel dieser Arbeit wird hier die Gleichung 2.2 schematisch diskutiert. Zu Darstellungszwecken sind die einzelnen Terme der Gleichung 2.2 mit $\mathcal{A}, \mathcal{B}, \mathcal{C}$ und $\mathcal{D}$ gekennzeichnet, sodass in den folgenden Abschnitten darauf Bezug genommen werden kann.

**Die Bedeutung der Terme einer Navier-Stokes-Gleichung** Zunächst werden die einzelnen Elemente einer Navier-Stokes-Gleichung anhand von Gleichung 2.2 erläutert. Die Größe $\rho$ (genauer: $\rho(\vec{r}, t)$) ist die *Massendichte*-Funktion. $\rho$ gibt die Massendichte eines Fluides an jedem Raumpunkt $\vec{r}$ zu jedem Zeitpunkt $t$ an.

Die im Term $\mathcal{B}$ vorkommende Größe $p = p(\vec{r}, t)$ beschreibt die in einem Fluid an der Stelle $\vec{r}$ auftretenden Drücke.

Im Term $\mathcal{C}$ kommt der in Gleichung 2.3 dargestellte viskose Spannungstensor $T_{\alpha\beta}$ vor. Dieser Tensor beschreibt die auf ein Volumenelement eines Fluides wirkenden Scherkräfte, Reibungskräfte und materialabhängigen Kräfte. Im Tensor $T_{\alpha\beta}$ (siehe Gleichung 2.3) kommen die materialabhängigen Koeffizienten wie die Scherviskosität $\eta$ und die Volumenviskosität $\zeta$ vor.

Der Term $\mathcal{D}$ beschreibt nun über die Funktion $f_\alpha$ alle Volumenkräfte, die von außen auf ein Volumenelement einwirken können. Dazu gehört z.B. die Gravitationskraft.

**Die Form der Navier-Stokes-Gleichung** Die Navier-Stokes-Gleichung (2.2) muss also in ihrer Form der Grundgleichung der klassischen Mechanik (2.1) gleich sein. Untersuchen wir hierzu die einzelnen Terme anhand der Gleichung 2.2.

Der Term $\mathcal{A}$ entspricht der Impulsänderung, also dem Term $m\vec{a}$ der Gleichung 2.1. Dabei entspricht die Dichtefunktion $\rho$ also der Masse $m$, und die zeitliche Ableitung des Geschwindigkeitsfeldes eines Fluides $\vec{v}(\vec{r})$ entspricht der Beschleunigung der Masse. Über eine Zeitintegration erhält man aus dem Geschwindigkeitsfeld die gesuchte Bahnkurve — bei einem Fluid das Strömungsverhalten.

Damit sind die Terme $\mathcal{B}, \mathcal{C}$ und $\mathcal{D}$ die auf das Fluid einwirkenden Kräfte, in Gleichung 2.1 mit $\vec{F}$ bezeichnet. Der Term $\mathcal{B}$ beschreibt die auf ein Fluid wirkenden Druckkräfte. Der Term $\mathcal{C}$ beschreibt die geometrisch komplexen Scher- und Reibungskräfte, die auf ein Volumenelement eines Fluids wirken. Der Term $\mathcal{D}$ beschreibt alle äußeren auf ein Fluid wirkenden Kräfte (Gravitationskraft).

Die Navier-Stokes-Gleichung (2.2) ist also in ihrer Form tatsächlich gleich der Grundgleichung der klassischen Mechanik (2.1). Für Fluide haben die einzelnen Ter-

me aufgrund der komplexen dreidimensionalen Geometrie ein mathematisch komplexes Aussehen.

Die Terme einer Navier-Stokes-Gleichung korrekt anzugeben ist ein nichttriviales Vorhaben. Um die einzelnen Terme wie die Druckkräfte oder die viskosen Kräfte für eine bestimmte Aufgabenstellung und ein bestimmtes Fluid richtig zu beschreiben, sind ganze Arbeitsgruppen der Physik beschäftigt (siehe auch [45], [16]).

**Die Mathematik einer Navier-Stokes-Gleichung** Wenn eine Navier-Stokes-Gleichung richtig aufgestellt wurde, ist der Physiker an der Lösung der Differentialgleichung interessiert. Im Fall der in Gleichung 2.2 beschriebenen Differentialgleichung muss dazu eine gewöhnliche Differentialgleichung in der Zeit und eine partielle Differentialgleichung über die Ortskoordinaten gelöst werden.

Zur Lösung der Differentialgleichung in der Zeit können bekannte Zeitintegrationsverfahren aus der Literatur verwendet werden (siehe [47]).

Die Lösungsverfahren zur Lösung der partiellen Differentialgleichung im Ort, also die Berechnung der rechten Seite der Gleichung 2.2, hängen nun stark von den gewählten Randbedingungen ab.

In den in dieser Arbeit diskutierten Simulationen werden Modelle mit offenen Randbedingungen und starken Dichtegradienten verwendet. Speziell für diese Voraussetzungen kommt eine in der Numerik sehr junge Methode zum Einsatz: Smoothed Particle Hydrodynamics.

### 2.1.2 Smoothed Particle Hydrodynamics

*Smoothed Particle Hydrodynamics* — kurz SPH — wurde 1977 von Lucy [38] sowie Gingold & Monaghan [21] vorgestellt, um damit kompressible hydrodynamische Strömungen mit freien Rändern zu simulieren. SPH ist ein numerisches Verfahren, um das System der gekoppelten partiellen Differentialgleichungen der Hydrodynamik (siehe Gleichung 2.2) in ein System gekoppelter gewöhnlicher Differentialgleichungen zu überführen. Speziell für die Fragestellungen der Astrophysik wird SPH häufig verwendet (für einen Überblick siehe [52] und [40]). Inzwischen wird SPH auch im Bereich der Materialwissenschaften zur Simulation von Festkörpern eingesetzt (z.B. [53] ). Interessant zu bemerken ist, dass für das SPH-Verfahren noch keine mathematischen Beweise über Konvergenz und Stabilität vorliegen.

Der wesentliche Unterschied zwischen gitterbasierten Lösungsmethoden für gekoppelte partielle Differentialgleichungen und SPH ist, dass die Kontinuumsgrößen bei SPH nicht auf festen Gittern diskretisiert werden, sondern auf mit der Strömung

mitbewegten massebehafteten Stützstellen.[4] Diese mitbewegten Stützstellen nennt man *particles* oder SPH-Teilchen, um sie von realen Teilchen der Physik zu unterscheiden. Per Konstruktion ist SPH also ein lagrangesches Verfahren und automatisch adaptiv. Aufgrund dieser Eigenschaft wird auch bei der Modellbildung ein hydrodynamisches Gleichungssystem im lagrangeschen Bild, so wie in Gleichung 2.2 beschrieben, verwendet.[5]

### Grundprinzipien des SPH-Verfahrens

Das SPH-Verfahren transformiert ein gekoppeltes System partieller Differentialgleichungen in ein System gekoppelter gewöhnlicher Differentialgleichungen. Dies geschieht in drei Schritten (siehe [52]):

1. Kernfunktions-Glättung, bei der alle ortsabhängigen Funktionen mit einer Kernfunktion gefaltet werden (dies entspricht der Berechnung der Terme $\mathcal{B}, \mathcal{C}, \mathcal{D}$ in Gleichung 2.2);

2. Monte-Carlo-Integration, durch die die im ersten Schritt gewonnenen Faltungsintegrale numerisch berechnet werden;

3. numerische Integration in der Zeit der übrig bleibenden gewöhnlichen Differentialgleichung (entspricht Berechnung des Terms $\mathcal{A}$ in Gleichung 2.2).

Der dritte Schritt verwendet Zeitintegrationsverfahren aus der Literatur (siehe [47]). Im Folgenden werden die ersten beiden Schritte, das eigentliche SPH-Verfahren, erläutert.

Das SPH-Verfahren beruht auf der Methode der interpolierenden Funktionen (*engl.: interpolation by kernel estimation*) [40]. Dazu wird eine Funktion $\langle f(\boldsymbol{r})\rangle$ eingeführt, auf die die gesuchte Lösung $f(\boldsymbol{r})$ der Differentialgleichung abgebildet wird (Faltung):

$$\langle f(\boldsymbol{r})\rangle = \int f(\boldsymbol{r'})W(|\boldsymbol{r}-\boldsymbol{r'}|,h)d\boldsymbol{r'}. \tag{2.4}$$

Wobei die Funktion $W(\boldsymbol{r},\boldsymbol{r'},h)$ folgende Eigenschaften haben muss:

1. $\lim_{h\to 0} W(\boldsymbol{r},\boldsymbol{r'},h) = \delta(\boldsymbol{r},\boldsymbol{r'},h)$[6]

2. $W(\boldsymbol{r},\boldsymbol{r}',h) = W(|\boldsymbol{r}-\boldsymbol{r}'|,h)$, d.h., $W$ hängt nur vom Abstand zwischen $\boldsymbol{r}$ und $\boldsymbol{r}'$ ab. Damit folgt für die Kernableitungen: $\nabla W(\boldsymbol{r},\boldsymbol{r'},h) = -\nabla W(\boldsymbol{r},\boldsymbol{r'},h)$.

3. W ist auf 1 normiert, d.h. $\int_{-\infty}^{\infty} W(\boldsymbol{r},\boldsymbol{r}',h)d\boldsymbol{r}' = 1$.

[4] Methoden zur Simulation von Fluiden finden sich z.B. in [2]

[5] Für eine Beschreibung eines hydrodynamischen Systems in Euler-Darstellung siehe [34].

[6] Mit dieser Eigenschaft wird die Gleichung 2.4 im Grenzfall $h \to 0$ zur identischen Abbildung.

4. $W(\boldsymbol{r}, \boldsymbol{r'}, h) = 0$ für $|\boldsymbol{r} - \boldsymbol{r'}| > h$.

Die Funktion $W$ wird häufig als Kern (*engl.: kernel*) bezeichnet, $h$ ist ein Maß für die Ausdehnung des Kerns und wird im Englischen als *Smoothinglength* bezeichnet. $h$ hat die Funktion eines *Wechselwirkungsradius* für ein SPH-Teilchen. In der vorliegenden Arbeit wird der im Deutschen gebräuchliche Begriff Wechselwirkungsradius für die Größe $h$ verwendet (siehe auch [52]).

Es läßt sich nun zeigen, dass gilt

$$\langle f(\boldsymbol{r})\rangle = f(\boldsymbol{r}) + O(h^2) \; . \tag{2.5}$$

Dies bedeutet, dass die Funktionen $\langle f(\boldsymbol{r})\rangle$ und $f(\boldsymbol{r})$ im Grenzfall, in dem der Wechselwirkungsradius $h \to 0$ geht, ineinander übergehen. Eine ausführliche Diskussion der Genauigkeit von Kernfunktionsglättungen bei SPH-Verfahren findet sich in [52].

Zur numerischen Berechnung muss die Gleichung 2.4 diskretisiert werden. Damit wird die genäherte Lösungsfunktion $\langle f(\boldsymbol{r})\rangle$ durch eine diskretisierte Funktion $\langle\langle f(\boldsymbol{r})\rangle\rangle$ angenähert. Ist die Funktion $f(\boldsymbol{r})$ nur an $N$ diskreten Stützstellen $\boldsymbol{r}_i$ im Raum bekannt, kann das Integral aus Gleichung 2.4 näherungsweise durch eine Summe dargestellt werden:

$$\langle f(\boldsymbol{r})\rangle \approx \langle\langle f(\boldsymbol{r})\rangle\rangle \equiv \sum_{j=1}^{N} \frac{f(\boldsymbol{r}_j)}{n_j} W(|\boldsymbol{r} - \boldsymbol{r}_j|, h) \; , \tag{2.6}$$

wobei $n_j = n(\boldsymbol{r}_j)$ die Stützstellendichte bezeichnet. Da diese Stützstellen sich mit der Geschwindigkeit der Strömung mitbewegen, nennt man diese Stützstellen auch *particles* oder SPH-Teilchen.

Dieser Formalismus wird nun auf die entsprechenden Gleichungssysteme angewendet. Im Spezialfall, in dem die Funktion $f = \rho$ gleich der Massendichte selbst ist, wird die Approximation der Massendichte $\rho$ eines Fluids zu:

$$\langle\langle \rho(\boldsymbol{r})\rangle\rangle = \sum_{j=1}^{N} m_j W(|\boldsymbol{r} - \boldsymbol{r'}|, h). \tag{2.7}$$

Mit Hilfe dieser Vorschrift werden nun die Differentialgleichungen transformiert. Dazu muss die Ableitung einer Funktion $f(\boldsymbol{r}_i)$ nach einer Koordinate $(x_\alpha)$ betrachtet werden. Für eine partielle Ableitung nach der Koordinate $x_\alpha$ in SPH-Darstellung gilt:

$$\frac{\partial}{\partial x_\alpha} f(\boldsymbol{r}_i) = \sum_j \frac{f(\boldsymbol{r}_j) + \tilde{f}(\boldsymbol{r}_i)}{n_j} \frac{\partial W(|\boldsymbol{r}_i - \boldsymbol{r}_j|, h)}{\partial x_\alpha} \; . \tag{2.8}$$

Entsprechend werden alle höheren Ableitungen transformiert. Eine ausführliche Berechnung der hier zusammengefassten Formeln ist in [44] gegeben. Dort werden auch Ableitungen höherer Ordnung in der SPH-Darstellung angegeben. Für eine ausführliche Diskussion über SPH siehe [52].

**Die Navier-Stokes-Gleichung in SPH-Formulierung**

Um eine Simulation mit der Navier-Stokes-Gleichung durch das SPH-Verfahren zu lösen, muss zuerst die SPH-Formulierung der Navier-Stokes-Gleichung 2.2 gefunden werden. In [15] wird eine SPH-Formulierung der Navier-Stokes Gleichung dargestellt. Bei der hier angegebenen Gleichung wurde der Druckterm bereits weggelassen und die Formel in zwei Teile zerlegt. Damit entspricht diese SPH-Formulierung den Termen $\mathcal{A}$ auf der linken Seite und $\mathcal{C}$ und $\mathcal{D}$ auf der rechten Seite der Navier-Stokes-Gleichung 2.2 auf Seite 12. Eine Herleitung ist in [52] oder [44] zu finden.

Die Navier-Stokes-Gleichung in SPH-Formulierung ist

$$\frac{dv_{i\alpha}}{dt} = \sum_j \frac{m\nu}{\rho_i \rho_j}(\rho_j \sigma_{j\alpha\beta} + \rho_i \sigma_{i\alpha\beta})\frac{\partial W_{ij}}{\partial x_\beta} - \rho_i \frac{\partial}{\partial x_\alpha}\Phi_M \tag{2.9}$$

mit dem viskosen Schertensor in SPH-Formulierung

$$\begin{aligned}\sigma_{i\alpha\beta} \;=\; & \sum_k \frac{m}{\rho_k}\left([v_\alpha(\boldsymbol{r}_k) - v_\alpha(\boldsymbol{r}_i)]\frac{\partial W_{ik}}{\partial x_\beta} + [v_\beta(\boldsymbol{r}_k) - v_\beta(\boldsymbol{r}_i)]\frac{\partial W_{ik}}{\partial x_\alpha}\right. \\ & \left. -\frac{2}{3}\delta_{\alpha\beta}[v_\gamma(\boldsymbol{r}_k) - v_\gamma(\boldsymbol{r}_i)]\frac{\partial W_{ik}}{\partial x_\gamma}\right),\end{aligned}$$

wobei $\rho_i = m\sum_j W_{ij}$ die SPH-Dichte am Ort des Teilchens $i$ darstellt. Die griechischen Indizes $\alpha, \beta, \gamma$ laufen über die Raumdimension. Im Zweidimensionalen stellt die SPH-Dichte $\rho_i$ eine Oberflächendichte dar. Die lateinischen Indizes laufen über die Anzahl der SPH-Teilchen $N$.

### 2.1.3 Monte-Carlo-Verfahren

Die Monte-Carlo-Methode wurde von Neumann, Ulam und Metropolis entwickelt, um die Diffusion von Neutronen in spaltbarer Materie zu simulieren. Für einen Überblick verweisen wir auf [2] und [47].

Der Name *Monte-Carlo* kommt durch den extensiven Gebrauch von „Zufallszahlen“.[7] Zur Funktionsweise einer Monte-Carlo-Methode nur dieses Beispiel:

> Der französische Naturalist Buffon entdeckte im 18. Jahrhundert ein Theorem über geometrische Wahrscheinlichkeit. Wenn eine Nadel der Länge $l$ zufällig über eine Menge Linien mit Abstand $d$ (wobei $d > l$) geworfen wird, so ist die Wahrscheinlichkeit, dass die Nadel eine der Linien berührt, $\frac{2l}{\pi d}$.
>
> Der italienische Mathematiker Lazzerini simulierte dieses Theorem im Jahre 1901 und warf dabei eine Nadel $3407$-mal über ein Blatt Papier mit

[7] Mit Zufallszahlen sind immer Umkehrfunktionen von Verteilungsfunktionen gemeint.

aufgezeichneten parallelen Linien. Dabei zählte er die Fälle, bei denen die Nadel eine der Linien traf, und konnte so die Zahl $\pi$ auf $3,1415929$ berechnen.

Dies ist die früheste bekannte Monte-Carlo-Integration zur Berechnung von $\pi$. Das Prinzip aller folgenden Monte-Carlo-Verfahren ist ähnlich (siehe [2]).

## 2.2 Paralleles Rechnen

In diesem Abschnitt werden alle für das Verständnis der folgenden Kapitel notwendigen Begriffe aus dem Bereich des parallelen Rechnens vorgestellt. Zuerst wird auf die parallele Hardware eingegangen und im Anschluss werden die Begriffe der parallelen Softwareentwicklung erläutert.

Zunächst soll der Begriff *paralleles Rechnen* genauer definiert werden. Im Duden der Informatik [14] wird unter *parallel* folgendes verstanden:

**Terminus 2.1 (parallel)** *Arbeitsabläufe bzw. deren Einzelschritte heißen* parallel, *wenn sie gleichzeitig und voneinander unabhängig durchgeführt werden können.*

Beispiele für parallele Arbeitsprozesse sind die parallele Datenübertragung, bei der die einzelnen Bits eines zu übertragenden Datums gleichzeitig über eine entsprechende Anzahl von Leitungen gesendet werden — im Gegensatz zur seriellen Datenübertragung, bei der nur eine Leitung zur Verfügung steht.

**Terminus 2.2 (Parallelverarbeitung)** *Gleichzeitige Verarbeitung eines Programms durch mehrere Prozessoren.*

Unter der Voraussetzung, dass es sich dabei um ein *paralleles* Programm handelt, ergibt sich bei $n$ Prozessoren im Idealfall eine Beschleunigung der Verarbeitung um den Faktor $n$. D.h., erst durch die Parallelverarbeitung werden die parallelen Arbeitsabläufe eines Programms auch nebenläufig ausgeführt und führen zu einer Laufzeitbeschleunigung.

Parallele Programme, die nicht durch Parallelverarbeitung ausgeführt werden, können auch keine Beschleunigung erfahren. Trotzdem kann es sinnvoll sein, ein paralleles Programm zu entwickeln, ohne den Vorteil einer Beschleunigung auszunutzen. Ein Beispiel dafür wäre ein WWW-Server, der viele gleichartige Anfragen *gleichzeitig* abarbeiten muss. Obwohl der WWW-Server auch auf einer Maschine mit nur einem Prozessor läuft, ist es sinnvoll, hier ein paralleles Programm zu entwickeln, da sich dieses Programmierparadigma besser an die vorgegebene Aufgabenstellung

anpasst. In einem seriellen Programmierparadigma müsste die gleichzeitige Bearbeitung der Anfragen mit in das Serverprogramm implementiert werden. Durch die Verwendung eines parallelen Programmierparadigmas etwa durch Threads wird die Aufgabe der Verwaltung der parallelen Programmeinheiten dem Betriebsystem des Rechners überantwortet. Das Betriebsystem wird auf einer Maschine mit nur einer CPU ein Scheduling-Verfahren für die parallelen Threads verwenden, um einen parallelen Programmablauf zu ermöglichen (siehe [56], [51]).

### 2.2.1 Hardware

Zunächst werden die grundlegenden Begriffe zu paralleler Hardware dargestellt. Anstelle von Hardware werden in dieser Arbeit auch austauschbar die Begriffe *Rechner* oder *Maschine* verwendet. Gemeint sind dabei immer *Parallelrechner* (siehe [3]):

**Terminus 2.3 (Parallelrechner)** *Als Parallelrechner bezeichnet man den Zusammenschluss mehrerer Ausführungseinheiten — auch* Knoten *genannt — unter Verwendung eines gemeinsamen Kommunikationsmittels.*

Bevor die Rechner, die für diese Arbeit verwendet wurden, vorgestellt werden, wird eine allgemeine Klassifikation von Parallelrechnern dargestellt.

#### Klassifikation von Parallelrechnern

Zur besseren Unterscheidung der verschiedenen Parallelrechnertypen wurden verschiedene Klassifikationen vorgeschlagen. Die bekannteste Klassifikation stammt von Flynn [17], mit einer Unterteilung in vier Klassen. Flynn unterscheidet die Parallelrechner dabei anhand der vorhandenen *Datenströme* und *Ausführungsströme*.

| | | Datenströme | |
|---|---|---|---|
| | | Single | Multiple |
| Ausführungsströme | Single | **SISD** | **SIMD** |
| | Multiple | **MISD** | **MIMD** |

Tabelle 2.1: Klassifikation von Parallelrechnertypen nach Flynn.

Unter einem *Ausführungsstrom* versteht man hier die serielle Abarbeitung von Instruktionen. Ein *Datenstrom* sind die Ein- und Ausgabedaten einer Verarbeitung.

**SISD-Computer** Unter einem *Single-Instruction-Single-Data*-Computer wird der klassische Von-Neumann-Rechner verstanden.

**MISD-Computer** Ein *Multiple-Instruction-Single-Data*-Computer hätte mehrere Ausführungsströme, die auf einem Datenstrom arbeiten würden. Es ist aber keine Implementierung der Klasse der MISD-Rechner bekannt.

**SIMD-Computer** Bei einem *Single-Instruction-Multiple-Data*-Rechner wird eine Instruktion auf mehrere Daten gleichzeitig angewendet. Implementierungen dieses Rechnertyps sind die *Vektorrechner* oder *Feldrechner* (siehe [50]). Mit diesen Rechnern lassen sich spezielle Programme besonders leicht und effizient parallelisieren. Dazu zählen vor allem Probleme aus der Physik und Mathematik, die mit Vektoren und Matrizen arbeiten. Allerdings erlaubt diese Klasse der Rechner keine unabhängige Verarbeitung von verschiedenen Arbeitsabläufen. Der einzig existierende Arbeitsablauf läuft immer synchron auf allen Daten ab.

**MIMD-Computer** Die Klasse der *Multiple-Instruction-Multiple-Data*-Computer ist der allgemeinste Typ von Parallelrechner. Hier sind mehrere eigenständig arbeitende Knoten (CPU) ohne eine zentrale Steuerungsinstanz mit einem Kommunikationsmedium miteinander verbunden. Auf jedem Knoten kann ein eigenständiges Programm (Ausführungsstrom) mit eigenem Programmzähler auf lokalen Daten (Datenstrom) arbeiten. MIMD-Rechner können mit hochintegrierten Standard-Prozessoren (*engl.: of-the-shelf-processor*) und entsprechenden Hochleistungs-Netzwerken implementiert werden.

Die Klasse der MIMD-Rechner ist die flexibelste Art der Parallelrechner. Für die Problemstellung der Parallelisierung der in der vorliegendenArbeit besprochenen Simulationsverfahren werden Rechner der MIMD-Klasse verwendet. Deshalb soll diese Klasse der Parallelrechner noch genauer dargestellt werden.

Um eine weitere Klassifizierung von MIMD-Rechner angeben zu können, sind in Abbildung 2.2 die nach [50] wesentlichen Merkmale von MIMD-Architekturen angegeben.

Das Hauptkriterium zur Einteilung von MIMD-Architekturen ist die *physikalische Speicheranordnung*, nach der Rechner in Systeme mit gemeinsamem Hauptspeicher (*engl.: shared memory architectures*) und Systeme mit verteiltem Hauptspeicher (*engl.: distributed memory architectures*) unterteilt werden. Zu beachten ist, dass die physikalische Anordnung sich von der logischen Sicht des *Adressraums* unterscheiden kann. Beide Arten von Speicheranordnungen können einen *globalen, gemeinsamen* oder einen *lokalen* Adressraum haben. Es ist üblich, auch für die Art des globalen Adressraums die Bezeichnung *gemeinsamer Speicher* (*engl.: shared memory*) zu verwenden. In der vorliegenden Arbeit wird der Begriff *gemeinsamer Speicher* oder *shared memory* ausschließlich als Kennzeichnung der physikalischen Speicher-

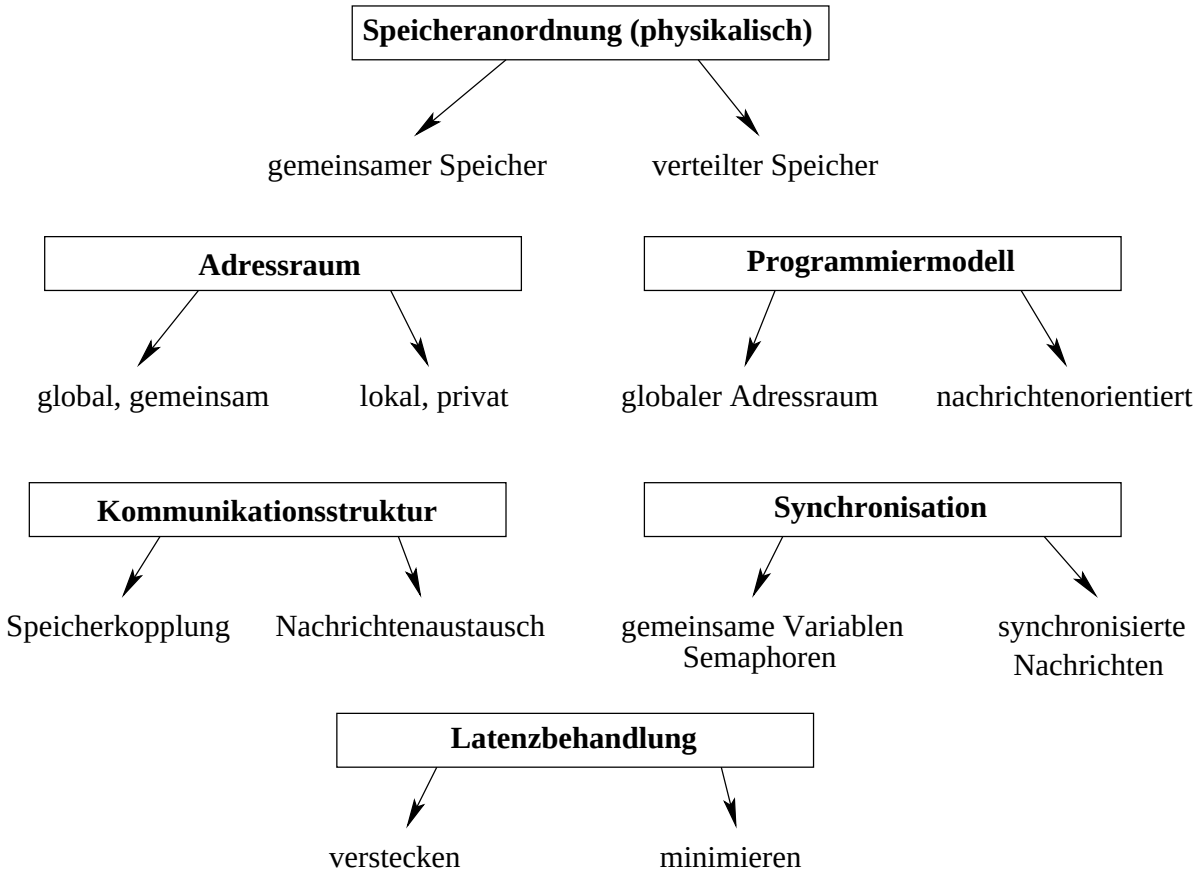



Abbildung 2.2: Merkmale zur Klassifikation von MIMD-Rechnern nach [50].

anordnung verwendet. Weitere Merkmale sind das *Programmiermodell,* das sich an der möglichen Adressierungsart des Rechners orientieren wird. Auf Maschinen mit lokalem Adressraum wird ein *nachrichtenorientiertes* Programmiermodell angewendet. Auf Maschinen mit globalem Adressraum kann dieser direkt angesprochen werden.

In Abbildung 2.3 sind die Klassifikationsmöglichkeiten für MIMD-Rechner nach [50] anhand der oben besprochenen Merkmale angegeben. Im Folgenden werden die für die vorliegende Arbeit verwendeten Parallelrechner vorgestellt. Dabei sind für die weiteren Ausführungen vor allem die *physikalische Speicheranordnung* sowie das *Programmiermodell* relevant.

#### Cray T3E

Die Cray T3E ist ein MIMD-Computer mit verteiltem Hauptspeicher als physikalische Speicheranordnung. Als Programmiermodell bietet die Cray T3E einen globalen Adressraum mit der Cray-eigenen SHMEM-Bibliothek an. Damit fällt die

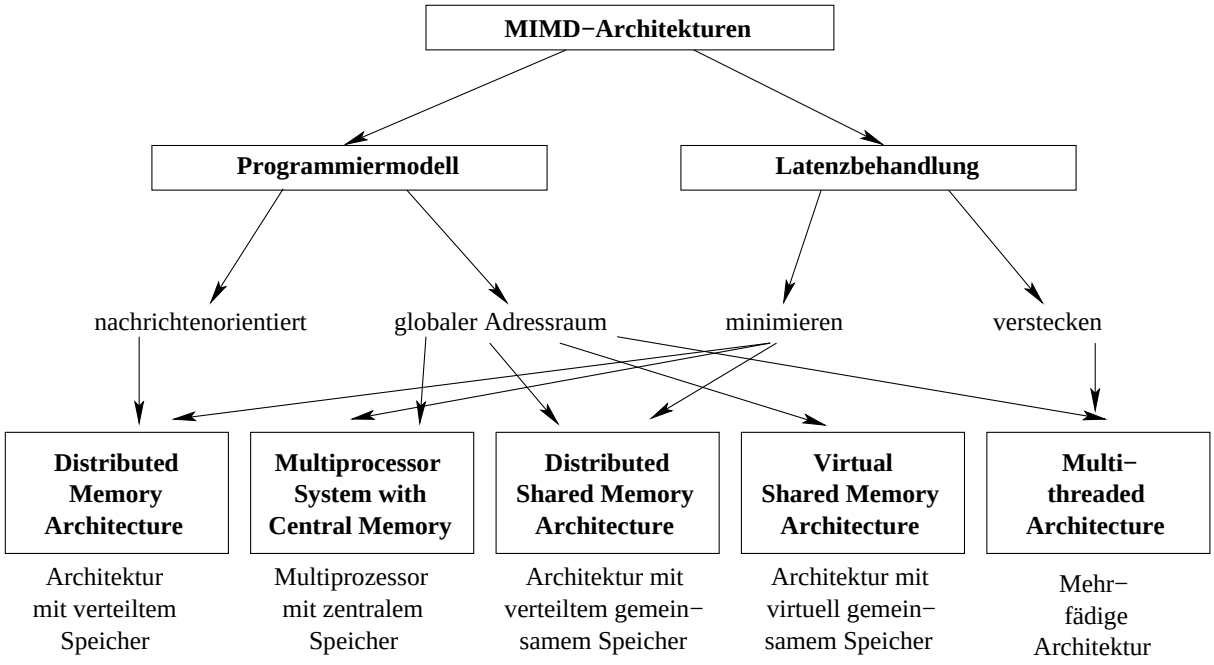



Abbildung 2.3: Klassifikation von MIMD-Rechnern nach [50].

Abbildung 2.4: Die Cray T3E

Cray T3E laut Abb. 2.3 in die Klasse der *Distributed-Shared-Memory*-Architekturen. Die Cray T3E bietet außerdem noch mit MPI[8] ein nachrichtenorientiertes Programmiermodell an, sodass die Cray T3E auch als *Distributed-Memory*-Architektur angesprochen werden kann.

Die am Hochleistungsrechenzentrum (HLRS) in Stuttgart installierte Cray T3E hat

[8]Message Passing Interface

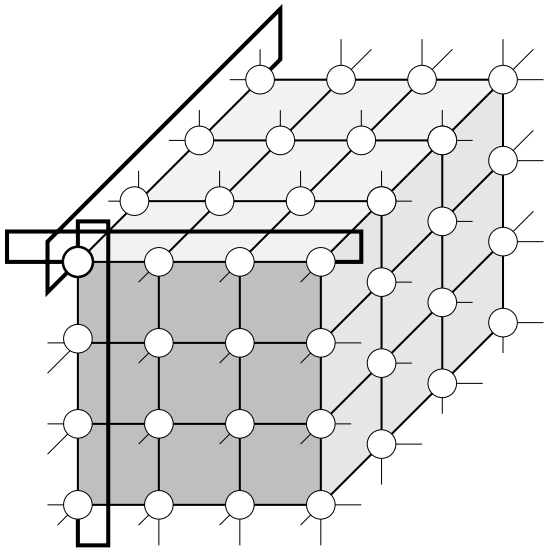


Abbildung 2.5: Skizze des Kommunikations-Torus der Cray T3E

512 Knoten mit je 128 MB lokalem Hauptspeicher. Als Prozessoren wurden handelsübliche DEC Alpha 21164 EV4 mit 450MHz verwendet. Als Kommunikationsnetzwerk wurde ein 3D-Torus verwendet (siehe Abbildung 2.5). Mit dem Kommunikationstorus sind Datenübertragungsraten von bis zu 500 MB/s in jede der drei Koordinatenrichtungen möglich. Datenpakete werden mit einem effizienten *Cut-Through*-Verfahren[9] bis zum Zielknoten weitergeleitet. Damit wird eine hohe Bandbreite wie auch eine geringe Latenzzeit erreicht.

In Tabelle 2.2 sind die Leistungsmerkmale der Cray T3E zusammengefasst.

| | |
|---|---:|
| Spitzenleistung | 461 GFLOP/s |
| Anzahl der Knoten | 512 |
| Taktfrequenz | 450 MHz |
| Speicher pro Knoten | 128 MB |
| Gesamtspeicher | 64 Gigabyte |
| Programmiermodelle | SHMEM, MPI, PVM |
| Betriebssystem | UNICOS/mk |

Tabelle 2.2: Leistungsmerkmale der Cray T3E


Abbildung 2.6: Die NEC SX-4.

**NEC SX-4**

Die NEC SX-4 ist ein MIMD-Computer mit gemeinsamem Speicher als physikalische Speicheranordnung. Die NEC SX-4 bietet einen globalen Adressraum als Programmiermodell an. In der Klassifizierung der MIMD-Rechner würde die NEC SX-4 als Multiprozessor mit zentralem Speicher bezeichnet werden. Dabei hat die NEC SX-4 als Knoten Vektor-CPU und verbindet dadurch die Vorteile der SIMD-Computer mit denen der MIMD-Computer. Programme können die Flexibilität der MIMD-Parallelrechner ausnützen und zusätzlich noch die Beschleunigungen durch eine Vektorisierung bekommen.

Die am HLRS in Stuttgart installierte NEC SX-4 hat 32 Knoten und 8 GB SSRAM[10] Hauptspeicher. Zusätzlich bietet die NEC SX-4 noch 16 GB DRAM als *extended memory* an. Die Speicherbänke sind über einen Kreuzschienenverteiler (*engl.: crossbar-switch*) an die Vektor-CPU angebunden.

In Tabelle 2.3 sind die Leistungsmerkmale der NEC SX-4 zusammengefasst.

| | |
|---|---|
| Spitzenleistung | 64 GFLOP/s |
| Anzahl der Knoten | 32 |
| Hauptspeicher | 8 GB SSRAM |
| erweiterter Speicher | 16 GB DRAM |

Tabelle 2.3: Leistungsmerkmale der NEC SX-4

Das Nachfolgemodell der NEC SX-4, die NEC SX-5, konnte in der vorliegenden Arbeit nicht mehr Verwendung finden. Die NEC SX-5 ist ein Cluster aus zwei Rechnern mit gemeinsamem Hauptspeicher und Vektor-CPU. Die Leistungsmerkmale der NEC SX-5 sind in Tablle 2.4 dargestellt.

[9] *Cut Through* bedeutet, dass von einem Paket lediglich der Kopf, der alle Informationen für die Weiterleitung enthält, ausgewertet wird und daraufhin sofort die Weiterleitung des Paketes durchführt wird.

[10] SSRAM: Synchronous-Static-RAM

| | |
|---|---|
| Spitzenleistung | $2 \times 64 = 128$ GFLOP/s |
| Anzahl der Prozessoren | $2 \times 16 = 32$ |
| Hauptspeicher | 32 + 48 = 80 GB SDRAM |
| Anzahl der Knoten | 2 |
| Knoten-zu-Knoten Transferrate | 8 GB/s |

Tabelle 2.4: Leistungsmerkmale der NEC SX-5.

**Kepler**

Das im Zentrum für Datenverarbeitung der Universität Tübingen im Rahmen des SFB 382 installierte PC-Cluster *Kepler* kann als Architektur mit verteiltem Speicher klassifiziert werden. Kepler bietet mit MPI ein nachrichtenorientiertes Programmiermodell an.

Das Kepler-Cluster basiert auf handelsüblichen PC-Hardware-Komponenten und läuft unter dem freien UNIX-Clone Linux (genauer: SuSE-Linux mit Kernel 2.2.16). Als Kommunikationsnetzwerk wird ein 1,28-GBit/s-Switched-LAN-Myrinet verwendet. Kepler ist damit eine äußerst preiswerte Alternative zu den Hochleistungsrechnern NEC SX-4 und Cray T3E. Kepler steht diesen Rechnern mit folgenden Werten gegenüber:

| | |
|---|---|
| Anzahl der Knoten | 98 |
| Doppel-Prozessor pro Knoten | Dual Pentium `III`, 650 MHz |
| Gesamtspeicher | 100 Gigabyte |
| Verbindungsnetzwerk | 1,28 GBit/s Myrinet |

Tabelle 2.5: Leistungsmerkmale des Kepler-Cluster.

Zur Installation von Kepler im November 2000 wird das Cluster in der TOP-500[11] auf Platz 215 notiert. Im Juni 2001 findet sich Kepler noch immer auf Platz 290 wieder. Im Vergleich steht die NEC SX-5 auf Platz 215 und die NEC-SX4/40 auf Platz 440. Die genannte Cray T3E ist auf Platz 67.

### 2.2.2 Software

Im Abschnitt 2.2.1 wurden die Grundbegriffe der für paralleles Rechnen notwendigen Hardware vorgestellt. Im Folgenden werden die zur Parallelverarbeitung von Programmen notwendigen Softwarekonzepte erläutert. Zuerst werden allgemeine Begriffe der parallelen Softwareentwicklung erklärt, dann wird eine Klassifikation für

[11] TOP-500 siehe: `http://www.netlib.org`

parallele Anwendungen gegeben. Die in der vorliegenden Arbeit verwendeten Programmierparadigmen werden im Anschluss gesondert besprochen.

### Allgemeines zum parallelen Programmieren

Beim Entwickeln paralleler Software spricht man vom *Parallelisieren* von Programmen. Die bereits definierten Begriffe *parallel* (2.1) und *Parallelverarbeitung* (2.2) werden ergänzt um die Definition aus [14]:

**Terminus 2.4 (Parallelisieren)** *Unter dem Vorgang des Parallelisierens versteht man das Offenlegen von Parallelität, die in dem untersuchten Algorithmus vorhanden ist. Teilweise muss die inhärente Parallelität durch korrekte Umformungen erst nutzbar gemacht werden.*

Das Parallelisieren eines Algorithmus erfordert eine genaue Kenntnis der Problemstellung. Um die einem Problem inhärente Parallelität offen legen zu können, müssen meist Umformungen in der Formulierung der Problemstellung gemacht werden. Ein bereits entwickelter (serieller) Algorithmus stellt im Allgemeinen keinen guten Ausgangspunkt für eine gute Parallelisierung dar, da dieser (serielle) Algorithmus von Annahmen ausgeht, die die inhärente Parallelität des Problems verdecken können. Ein Beispiel dafür ist in der vorliegenden Arbeit zu finden (siehe Kapitel 4).

Im Allgemeinen umfasst ein Verfahren zur Parallelisierung von Programmen explizit oder implizit folgende Punkte:

1. Parallele Abläufe starten und beenden:

   In einem parallelen Programm müssen Programmstellen kennzeichenbar sein, an denen ein nebenläufiger Programmfluss anfängt beziehungsweise aufhört. Zwischen den im Programm als parallel gekennzeichneten Stellen wird ein Progamm seriell ablaufen.

2. Kommunikation zwischen parallelen Abläufen:

   Im Allgemeinen werden nebenläufige Programmteile mit anderen nebenläufigen Programmteilen kommunizieren müssen. Ein Verfahren zur Parallelisierung muss eine Möglichkeit anbieten, damit zwischen nebenläufigen Programmteilen Daten übertragen werden können.

3. Synchronisation von konkurrierenden, parallelen Abläufen:

   Unter der Voraussetzung, dass sich nebenläufige Programmteile eine Ressource teilen, muss die Integrität der Ressource gewahrt bleiben.

Als Beispiel sei das Drucken von Daten auf einem Drucker genannt. Versuchen zwei nebenläufige Arbeitseinheiten Daten auf einem Ausgabemedium wie einem Drucker auszugeben, ohne dass diese parallelen Prozesse synchronisiert werden, dann wird die Datenausgabe auf dem Drucken eine zufällige [12] Reihenfolge der Daten der beteiligten parallelen Prozesse sein.

Der Sachverhalt, dass ein Ergebnis von der Reihenfolge von parallelen Prozessen abhängt, wird als Wettlaufbedingung (*engl.: race condition*) bezeichnet. Dies führt zu der Definition eines weiteren Begriffs der parallelen Programmierung.

**Terminus 2.5 (Kritischer Abschnitt)** *Ein Programmbereich, dessen Ergebnis bei unsynchronisierter Ausführung von der Anzahl und der Ausführungsreihenfolge der nebenläufig in diesem Abschnitt laufenden Arbeitseinheiten abhängt, heißt* kritischer Abschnitt *(engl.: critical section) (siehe [56]).*

### Klassifikation von parallelen Anwendungen

Nachdem eine Analyse der Problemstellung die inhärente Parallelität offen gelegt hat, muss ein geeignetes Softwarekonzept zur Implementierung gewählt werden. In [48] werden als die zwei wichtigsten Techniken die *Datenparallelität* und die *Kontrollflussparallelität* genannt. Für Kontrollflussparallelität wird auch der Begriff *Funktionsparallelität* verwendet [50].

**Datenparallelität** Bei einer datenparallelen Implementierung werden die Einzelschritte eines Algorithmus in gleicher Weise auf große Mengen von Daten angewandt ([50]). Mit [48] wird Datenparallelität bestimmt als:

**Terminus 2.6 (Datenparallelität)** *Unter Datenparallelität versteht man die gleichzeitige Benutzung von mehreren Funktionseinheiten, um eine identische Operation auf mehreren Elementen einer Datenmenge anzuwenden.*

SIMD-Computer eignen sich besonders gut zur Ausführung von datenparallelen Anwendungen. Es wird dabei eine Operation auf einem Feld von unterschiedlichen Daten ausgeführt. Ein einfaches Beispiel ist die Addition von Vektoren. Die Elemente der Vektoren werden unabhängig voneinander aufaddiert. Die Operation *addiere* wird auf verschiedenen Daten (Elemente der Vektoren) gleichzeitig ausgeführt.

[12] *Zufällig* bedeutet hier, dass die Reihenfolge der Ausführung paralleler Prozesse von externen Parametern, wie beispielsweise anderen Systemprozessen, beeinflusst wird.

**Kontrollflussparallelität** Bei der Kontrollflussparallelität können verschiedene Operationen auf unterschiedlichen Daten ausgeführt werden.

**Terminus 2.7 (Kontrollflussparallelität)** *Unter Kontrollflussparallelität versteht man die gleichzeitige Benutzung von mehreren Funktionseinheiten, um verschiedene Operationen auf unterschiedlichen Teilproblemen des Gesamtproblems durchzuführen (siehe [48]).*

Die Kontrollflussparallelität verlangt im Gegensatz zur Datenparallelität nicht, dass die parallelen Teilprobleme alle identisch sein müssen. Es können mit der Kontrollflussparallelität auf verschiedenen Knoten eines Rechners unterschiedliche Teilprobleme bearbeitet werden. Kontrollflussparallelität verlangt damit Rechner der MIMD-Klasse (siehe 2.2.1).

In Anlehnung an das Klassifikationsschema von [17] können parallele Anwendungen in die Klassen *SPMD* und *MPMD* unterteilt werden.

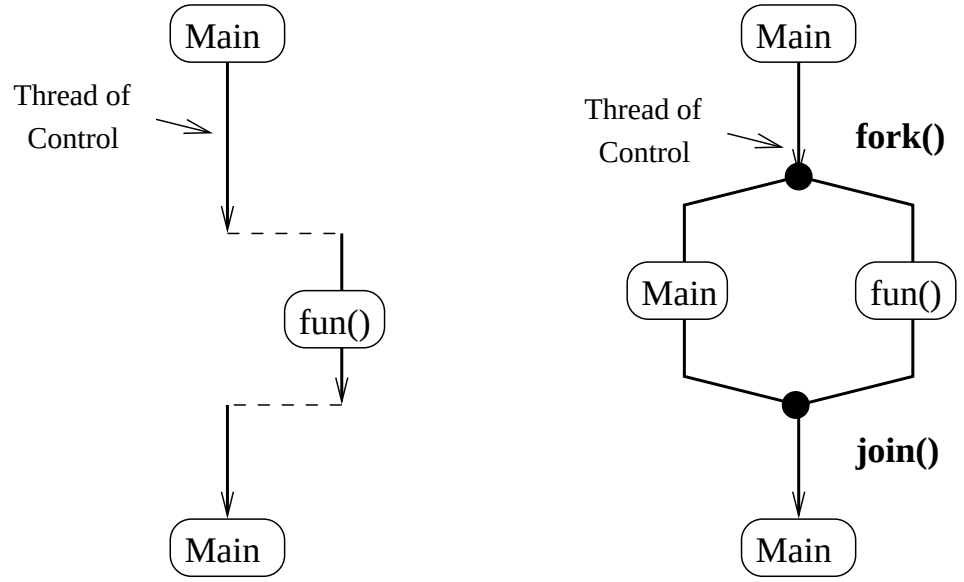
**SPMD — Single Programm Multiple Data** SPMD bedeutet, dass ein einziges Programm auf allen parallelen Ausführungseinheiten ausgeführt wird. Dabei arbeitet jeder parallele Programmteil auf seinen eigenen, lokalen Daten. SPMD bedeutet nicht zwingend, dass alle parallelen Ausführungseinheiten die gleichen Anweisungen abarbeiten, d.h., auch kontrollflussparallele Programme sind aus der SPMD-Klasse. Datenparallelität fordert aber immer ein Programm der SPMD-Klasse.

**MPMD — Multiple Programm Multiple Data** Hier wird eine Anwendung auf verschiedene Programme und Rechner verteilt. Der typische Fall eines MPMD-Programms ist eine Client-Server-Anwendung.

### Threads

Thread (*engl.: Faden*) steht für *thread of control* und könnte mit *Ausführungsstrang* übersetzt werden. Im Folgenden wird der Fachbegriff *Thread* verwendet. Das Verfahren der Threads wurde erstmals von Conway [9] für Multiprozessormaschinen mit dem Programmiermodell des globalen Adressraums vorgestellt. In [6] wurde das Verfahren der Threads auch auf Computer mit nachrichtenorientiertem Programmiermodell ausgeweitet. Dadurch lässt sich ein kontrollflussorientiertes, paralleles Programm, das mit dem Verfahren der Threads implementiert wurde, auf allen Arten von MIMD-Computern ausführen. Programme die mit dem Thread-Verfahren implementiert werden gehören zur SPMD-Klasse.

Mittlerweile unterstützen alle modernen Betriebssysteme das Konzept der Threads. Auf der Ebene des Betriebssystems stellt sich ein Thread wie ein *leichtgewichtiger* Prozess dar. Threads laufen im Kontext eines Prozesses ab und teilen sich den Prozesskontrollblock sowie die Register und Daten des Prozesses (siehe [56]).




(a) Single−Threaded (b) Multi−Threaded

Abbildung 2.7: Nebenläufiger Prozeduraufruf mit Threads. (a) zeigt einen seriellen Programmablauf mit Prozeduraufruf. (b) zeigt eine parallele Verzweigung eines Prozeduraufrufs mit `fork` und `join`.

Aus Sicht der Programmierung bieten Threads ein Verfahren zur Parallelisierung auf der Ebene von Prozeduraufrufen[13] in einem Programm an. Ein Thread kann als *nebenläufiger* Prozeduraufruf verstanden werden. Dabei wird die Prozedur in einem eigenen Thread nebenläufig ausgeführt. Bei einem seriellen Programm wird der Prozeduraufruf von dem einzig existierenden Thread ausgeführt. Die Programmkontrolle wechselt dabei in die Prozedur und springt nach Abarbeitung dieser in das aufrufende Programm zurück. In Abbildung 2.7 ist Prozeduraufruf mit Threads schematisch dargestellt.

Bei einem Programm, das mittels des Verfahrens der Threads parallelisiert wurde, spricht man auch von *multi-threaded*.

[13] Unter Prozedur versteht der Autor das in der Sprache PASCAL definierte Sprachkonstrukt eines Unterprogrammaufrufs. In anderen Sprachen wird dies als Funktion — wie in `C` — oder Subroutine — wie in `FORTRAN` — bezeichnet.

Die oben genannten drei Punkte zur Parallelisierung von Programmen werden bei dem Verfahren mit Threads durch folgende Konstrukte implementiert:

1. Das Fork/Join-Paradigma zum Starten und Beenden von Parallelität. Eine Prozedur wird über einen Systemaufruf vom aktuellen Thread abgezweigt (*engl.: fork* — die Gabel). Durch *join* (*engl. für:* Zusammenführen) werden Threads eines vorhergegangenen *forks* mit dem aktuellen Thread wieder vereinigt. D.h., der aktuelle Thread wartet bei einem *join*, bis ein bestimmter — oder alle vorher mit *fork* erzeugten — Thread abgearbeitet ist. Das Ende eines Threads wird immer durch das Programmende der Prozedur bestimmt.

2. Kommunikation zwischen Threads erfolgt über das Konstrukt des gemeinsamen Speichers. In allen gängigen Implementierungen des Thread-Verfahrens, wie z.B. den POSIX-Threads, wird eine Ausführung auf Maschinen mit gemeinsamem Speicher vorausgesetzt.

   Eine Kommunikation zwischen zwei Threads erfolgt demnach über eine gemeinsame, globale Variable. Damit stellt jede Kommunikation zwischen zwei Threads gleichzeitig auch einen *kritischen Abschnitt* dar, der durch Synchronisation explizit geschützt werden muss.

3. Synchronisation erfolgt bei dem Thread-Verfahren mittels des Konstrukts des wechselseitigen Ausschlusses (*engl.: mutual exclusion*).

   In allen Implementierungen von Thread-Verfahren wird ein binärer Semaphor verwendet, um kritische Abschnitte zu schützen. Synchronisation muss bei Thread-Verfahren explizit bei jedem kritischen Abschnitt angegeben werden.

   Bei objektorientierten Sprachen wie JAVA, die ein Thread-Verfahren anbieten, ist ein analoges Konzept der *Monitore* implementiert. Dies bietet dem Programmierer einen besser handhabbaren Weg, Threads zu synchronisieren.

Technische Details zur Programmierung mit Threads finden sich z.B. in [36].

### Message-Passing

Message-Passing (*engl. für:* Nachrichtenaustausch) wird vor allem auf MIMD-Computern mit nachrichtenorientiertem Programmiermodell eingesetzt. Message-Passing-Systeme arbeiten dabei mit den Primitiven *Send* und *Receive* — dem Senden und Empfangen von Nachrichten.

Die drei grundsätzlichen Punkte für Parallelisierungverfahren werden beim Message-Passing-System realisiert durch:

1. Parallele Abläufe starten und beenden:

   Zum Programmstart wird eine feste Anzahl von parallelen Ausführungseinheiten gestartet. Dies geschieht über die implementierungsabhängige Laufzeitumgebung des Message-Passing-Systems.

   Ausgehend von Rechnern mit verteiltem Hauptspeicher startet dasselbe Programm auf allen Knoten des Rechners gleichzeitig. Damit folgt auch dieses Verfahren dem SPMD-Paradigma.

   Nach Programmstart steht diese einmalig erzeugte Parallelität für den restlichen Programmablauf zur Verfügung und wird erst bei Programmende wieder beendet.

2. Die Kommunikation zwischen den zum Programmstart erzeugten nebenläufigen Teilen erfolgt durch explizites Zusenden von *Nachrichten* über die Kommunikationsprimitiven *Send* und *Receive*.

3. Synchronisation erfolgt — im Gegensatz zu der expliziten Synchronisation bei Thread-Verfahren mittels Semaphor — bei Message-Passing-Systemen implizit durch die *Send*- und *Receive*-Konstrukte der Kommunikationsprimitiven.

Der Standard für Message-Passing-Systeme ist das *Message-Passing-Interface* MPI. Für technische Details wird auf die MPI-Handbücher verwiesen.

Abschließend soll noch bemerkt werden, dass die vorgestellten Parallelisierungsverfahren zur Programmierung von MIMD-Rechner mit globalem Adressraum (Threads) oder für MIMD-Rechner mit nachrichtenorientiertem Programmiermodell in ihrer Semantik äquivalent sind (siehe [56]).

Das bedeutet auch, dass die Parallelisierung eines Problems unabhängig von dem später gewählten Parallelisierungsverfahren gemacht werden kann.

Bis auf Fragen der Effizienz sind parallele Programme damit auch auf allen Rechnern der MIMD-Klasse ausführbar. Dies wird z.B. in [6] gezeigt.

### 2.2.3 Speedup und Effizienz von parallelen Programmen

Zur Messung von parallelen Programmen werden außer der Programmlaufzeit noch die aus Knotenanzahl und Laufzeit errechneten Größen der *parallelen Effizienz* und des *Speedup* verwendet. Diese Begriffe sollen kurz erläutert werden (siehe auch [48]).

**Terminus 2.8 (Laufzeit $\mathcal{L}$)** *Unter der Laufzeit $\mathcal{L}$ eines parallelen Programms versteht man die Zeit, die während der Programmausführung verstreicht.*

Die Laufzeit $\mathcal{L}$ wird mit der Systemzeit des Rechners gemessen, auf dem das Programm ausgeführt wird.

Über Laufzeitmessungen bei unterschiedlicher Anzahl von Prozessoren wird die Güte der Parallelisierbarkeit eines Programms gemessen. Der Quotient der seriellen Laufzeit (Laufzeit auf einer CPU) mit der parallelen Laufzeit auf $N$ Prozessoren heißt Speedup des Programms bei $N$ Prozessoren. Je größer der Speedup für eine Anzahl von Prozessoren, desto besser parallelisiert das Programm.

**Terminus 2.9 (Speedup $\mathcal{S}$)**

$$\mathcal{S}(N) = \frac{\mathcal{L}_0}{\mathcal{L}(N)}$$

wobei $\mathcal{L}_0$ die serielle Laufzeit, d.h. die Laufzeit des Algorithmus auf einer CPU bedeutet. $\mathcal{L}(N)$ ist die gemessene Laufzeit bei $N$ CPU und $\mathcal{S}$ der daraus resultierende Speedup des Programms bei $N$ CPU.

Ein Maß für die Güte der Parallelisierung stellt die parallele Effizienz dar. Dabei wird angenommen, dass für $N$ CPU der maximale Speedup gleich $N$ betragen kann. Die parallele Effizienz ist damit definiert als der Quotient der eingesetzten Prozessoren und dem gemessenen Speedup.

**Terminus 2.10 (Parallele Effizienz $\mathcal{E}$)**

$$\mathcal{E}(N) = \frac{\mathcal{S}(N)}{N}$$

Mit $N$ der Anzahl der Prozessoren und dem Speedup $\mathcal{S}$.

## 2.3 Objektorientierte Design-Pattern

Nachdem die Grundlagen des Teilchensimulationsverfahrens SPH und des parallelen Rechnens besprochen wurden, wird jetzt noch ein Überblick über die Methode des objektorientierten Programmierens gegeben. Im Kapitel 1 wurde bereits der Einsatz von objektorientierter Programmierung begründet. Die Frage, ob und wie objektorientierte Programmierung sich für die Problemstellungen der vorliegenden Arbeit eignet, wird in Kapitel 5 diskutiert. Zunächst soll ein Überblick der Begriffe aus der objektorientierten Programmierung gegeben werden.

### 2.3.1 Grundbegriffe

Objektorientierte Programmierung ist, nicht zuletzt durch Java [23], zwar ein aktuelles Thema, aber keineswegs ein neues Konzept. Die Anfänge der objektorientierten Programmierung gehen bis in die 1960er-Jahre zurück — zur Sprache Simula67 [39].

Die am Xerox Palo Alto Research Center (Xerox PARC) in den 1970er-Jahren entwickelte Sprache Smalltalk kann als die erste vollständige, robuste, objektorientierte Sprache bezeichnet werden [60]. In Smalltalk sind alle Sprachelemente *Objekte* [22].

**Terminus 2.11 (Objekt)** *Ein Objekt hat einen Status, ein Verhalten und eine Identität; die Struktur und das Verhalten ähnlicher Objekte sind in ihrer gemeinsamen Klasse definiert; die Begriffe Instanz und Objekt sind austauschbar* (siehe [5]).

Und damit zusammenhängend:

**Terminus 2.12 (Klasse)** *Eine Klasse ist eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen* (siehe [5]).

Dabei wird unter *Status* und *Verhalten* mit [5] Folgendes verstanden:

**Terminus 2.13 (Status)** *Der Status eines Objekts umfasst alle (normalerweise statischen) Eigenschaften des Objektes, zusammen mit den aktuellen (normalerweise dynamischen) Werten dieser Eigenschaften.*

**Terminus 2.14 (Verhalten)** *Verhalten ist die Art und Weise, wie ein Objekt agiert und reagiert, in Form von Statusänderungen und der Übergabe von Nachrichten.*

Die folgende Definition von *Identität* aus [10] stellt vor allem klar, dass zwischen dem Namen eines Objekts — d.h. dessen Adressierbarkeit — und dem Objekt selber ein Unterschied besteht.

**Terminus 2.15 (Identität)** *Identität ist die Eigenschaft eines Objekts, die es von allen anderen Objekten unterscheidet.*

Auch heute noch wird Smalltalk als die reinste objektorientierte Sprache bezeichnet und sie war der Ausgangspunkt der heute in der Industrie laufenden Arbeiten. Simula67 und Smalltalk demonstrierten, dass es möglich ist, Programmieraufwände einzusparen, indem *Objekte* und *Klassen* wiederverwendet werden.

Dass die objektorientierte Programmierung erst relativ spät in den 1980er Jahren eine weitere Verbreitung gefunden hat, mag damit zusammenhängen, dass die Entwicklung der Smalltalk-Umgebung ausserhalb von Xerox PARC wenig bekannt wurde. Zum anderen war die Entwicklergemeinde auch mehr damit beschäftigt, Computer leichter bedienbar zu machen, und hat das objektorientierte Programmierkonzept von Smalltalk mehr als eine graphische Benutzeroberfläche mit Windows[14]

[14] Anm.: Der deutsche Begriff *Fenster* ist unüblich; deshalb wird im Folgenden das Computerfachwort *Windows* verwendet

missverstanden. Dies erklärt vielleicht auch, warum einige Hersteller von graphischen Benutzeroberflächen es ihren Benutzern anfänglich nicht ermöglichten, die Eigenschaften der Icons und Windows zu verändern, sondern diese lediglich als Bilder zu verwenden [60].

Anfang der 1980er-Jahre wurde die weit verbreitete Sprache C (siehe [31]) durch das Konzept der Klassen erweitert [54] und in den folgenden Jahren zu der objektorientierten Sprache C++ ausgebaut [55], die Anfang der 1990er-Jahre zum Industriestandard wurde[15]. Mit C++ wurde die objektorientierte Programmierung einer schon großen C-Entwicklergemeinde nahe gebracht, was auch die weite Verbreitung der Sprache C++ erklärt. Aus der großen Vielzahl der seit der Mitte der 1980er-Jahre entstandenen objektorientierten Sprachen und Systeme soll hier noch auf den Macintosh (1984) und den NeXT-Computer (1988) sowie auf das Oberon-Programmiersystem [49] hingewiesen werden.

Die 1995 als Programmiersprache für das WWW[16] entwickelte Sprache Java hat der objektorientierten Programmierung zu ihrem bisherigen Höhepunkt verholfen. Mit der Syntax angelehnt an die Sprache C ist Java — wie schon zuvor C++ — für eine große Entwicklergemeinde leicht erlernbar. Java bietet durch einfachere und klarere objektorientierte Konzepte gegenüber C++ auch Vorteile für Neueinsteiger. Durch seine umfangreichen Klassenbibliotheken und Eigenschaften wie den Bytecodeinterpreter bietet Java auch Möglichkeiten, wie sie von Smalltalk her bekannt sind.

Außer den technischen Merkmalen hat sicherlich die Popularität des WWW als die *graphische Oberfläche* des Internets zu der rasanten Verbreitung von Java beigetragen. Man kann sagen, dass Java die objektorientierte Programmierung sprichwörtlich in jeden Haushalt getragen hat. Mittlerweile ist mehr Literatur zu Java als zu Sprachen wie C++ oder Smalltalk erhältlich. Es ist auch erstaunlich zu beobachten, dass viele Entwickler und Programmierprojekte erst Ende der 1990er-Jahre durch Java erstmals mit der *objektorientierten Programmierung* in Kontakt kamen.

**Terminus 2.16 (Objektorientierte Programmierung)** *Objektorientierte Programmierung ist eine Implementierungsmethode, bei der Programme als kooperierende Ansammlungen von Objekten angeordnet sind. Jedes dieser Objekte stellt eine Instanz einer Klasse dar, und alle Klassen sind Elemente einer Klassenhierarchie, die durch Vererbungsbeziehungen gekennzeichnet ist. (Siehe: [5])*

*Die objektorientierte Programmierung wird mit* OOP *abgekürzt.*

In der Einleitung dieses Abschnitts wurde von einer *objektorientierten Methode*

[15] C++ ist keine rein objektorientierte Sprache, sondern wird als Multiparadigmen-Sprache bezeichnet. Es ist mit C++ möglich, auch objektorientierte Konzepte zu implementieren

[16] World Wide Web. Siehe http://www.w3.org

gesprochen. Ein neues Programmierparadigma alleine stellt aber noch keine Methode dar. Auch die oben angesprochenen Vorteile, wie die Wiederverwendbarkeit von Lösungen und die Ersparnis im Aufwand bei der Programmierung, sind nicht allein durch die Verwendung einer objektorientierten Programmiersprache zu bekommen.

**Die objektorientierte Methode**

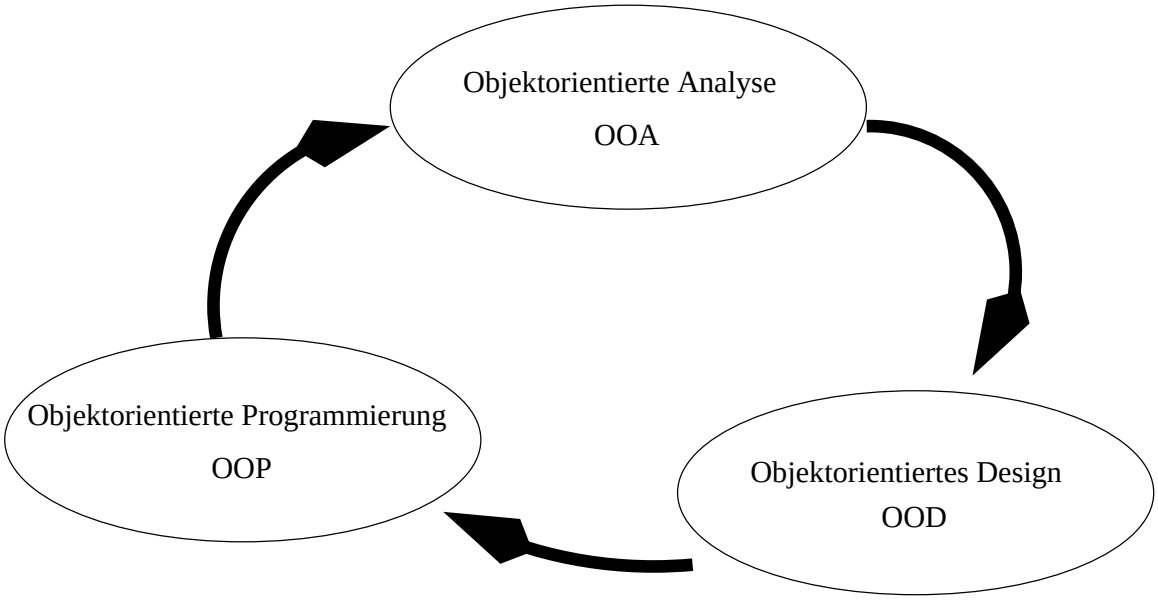



Abbildung 2.8: Schema eines objektorientierten Entwicklungsprozesses nach [5].

In Abbildung 2.8 ist das Schema eines objektorientierten Entwicklungsprozesses gezeigt. Zum einen ist zu sehen, dass die objektorientierte Programmierung nur einen Teil der Methode ausmacht. Zum anderen ist zu erkennen, dass die Teile OOA, OOD und OOP zyklisch ineinander greifen. Der objektorientierte Entwicklungsprozess hat sich zu Eigen gemacht, dass ein *iteratives* Vorgehen der bessere Weg vom Problem zum Programm ist.

Damit ergeben sich zwei weitere Begriffe, die erklärt werden müssen (siehe [5]):

**Terminus 2.17 (Objektorientiertes Design)** *Objektorientiertes Design ist eine Designmethode, die den Prozess der objektorientierten Zerlegung beinhaltet sowie eine Notation für die Beschreibung der logischen und physikalischen wie auch statischen und dynamischen Modelle des betrachteten Systems.*

*Für objektorientiertes Design wird auch das Akronym* OOD *verwendet.*

**Terminus 2.18 (Objektorientierte Analyse)** *Die objektorientierte Analyse ist eine Analysemethode, die die Anforderungen aus der Perspektive der Klassen und Objekte, die sich im Vokabular des Problembereichs finden, betrachtet.*

*Objektorientierte Analyse wird auch* OOA *genannt.*

Die Skizze in Abbildung 2.8 beschreibt den objektorientierten Entwicklungsprozess zwar nur sehr grob, aber ausreichend zum Verständnis für die vorliegende Arbeit. Zurzeit bildet sich der *Rational Unified Process (RUP)*[17] zu einem Industriestandard für den objektorientierten Entwicklungsprozess heraus. Ein Überblick über die objektorientierte Methode gibt [18].

Im Folgenden sollen noch die für die vorliegende Arbeit wichtigen Punkte der objektorientierte Methode dargestellt werden. Es wird zunächst die Möglichkeit, objektorientiertes Design und objektorientierte Analyse in einer normierten Symbolsprache (UML) notieren zu können, besprochen und im Folgenden auf die objektorientierten Design-Pattern eingegangen.

### 2.3.2 Unified Modeling Language

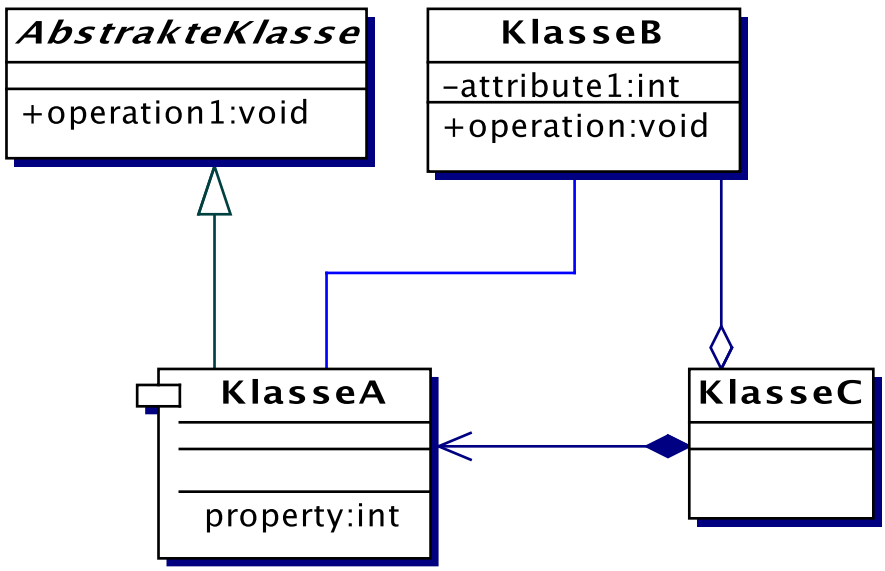



Abbildung 2.9: Klassendiagramme zeigen die statische Sicht auf das System.

Die *Unified Modeling Language* ist der aktuelle Industriestandard zur Notation von objektorientierten Systemen. UML stellt dabei in der Version 1.3 insgesamt 11

[17] siehe: http://www.rational.com

Diagrammtypen zur Verfügung. Hier soll nur auf die für die vorliegende Arbeit wichtigen Diagrammtypen eingegangen werden. Dabei werden nur die wesentlichen Merkmale der Diagramme besprochen. UML bietet darüber hinaus aber Diagramme für alle im objektorientierten Entwicklungsprozess auftretenden Problemstellungen an. Dies geht vom Anwendungsfalldiagramm (*engl.: use-case*) für eine erste Analyse auf der Ebene der Sprache der gegebenen Problemstellung bis hin zu speziellen Diagrammen für Enterprise-Java-Beans (EJB) Server und umfasst die Möglichkeit der Darstellung von Geschäftsprozessen und Netzwerktopologien.

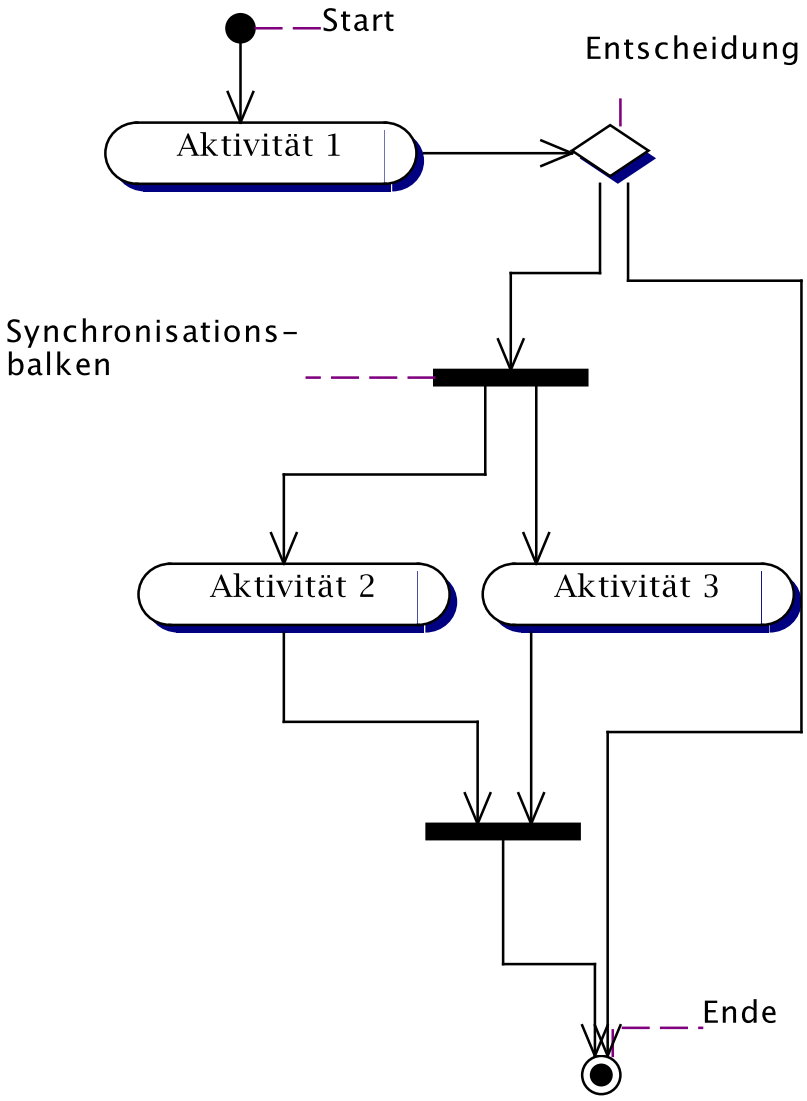



Abbildung 2.10: Aktivitätsdiagramme zeigen den fachlichen Ablauf eines Programms.

In Abbildung 2.9 ist ein UML-Klassendiagramm zu sehen. Ein Klassendiagramm

bietet eine statische Sicht auf die Struktur des Systems. Dabei werden Klassen als Rechtecke mit vordefinierten Unterteilungen gezeichnet. Oben steht der Klassenname, bei abstrakten Klassen *kursiv* geschrieben. Darunter reihen sich die Attribute, Operationen und Eigenschaften (*engl.: properties*) an. Dabei können diese Elemente mit einem + für öffentlich (*engl.: public)* oder einem − für privat (*engl.: private*) gekennzeichnet werden. Beziehungen zwischen Klassen werden durch Verbindungslinien mit verschiedenen Farben und Formen dokumentiert. Eine Verbindungslinie mit einem offenen Pfeil zeigt eine Generalisierungsbeziehung an. Generalisierungen werden in der objektorientierten Programmierung in der Regel als *Vererbung* implementiert, d.h., im UML-Klassendiagramm wird vom speziellen Objekt zum allgemeinen vorgegangen, was der tatsächlichen Vorgehensweise während der Analyse und Design-Phase entspricht. Während der Programmierung ist das Design bekannt und es wird vom allgemeinen auf das spezielle Objekt *vererbt*. Verbindungslinien zwischen Klassen markieren Assoziationen. Eine offene Raute bedeutet eine Aggregation, eine geschlossene Raute eine Komposition. Gerichtete Assoziationen haben eine Pfeilspitze. Generalisierungen werden auch als *ist-ein*-Beziehungen, Assoziationen als *hat-ein*-Beziehungen bezeichnet. Das Diagramm in Abbildung 2.9 ist bei weitem nicht vollständig, zum Verständnis der Darstellung der in der vorliegenden Arbeit entwickelten Design-Pattern (siehe Anhang A) aber ausreichend. Für eine ausführliche Darstellung der UML-Diagramme und Symbolik muss an dieser Stelle auf Standardliteratur wie [5] oder [43] verwiesen werden.

Es soll an dieser Stelle noch kurz auf das Aktivitätsdiagramm eingegangen werden, da dieses in Kapitel 4 eingesetzt wird. Das Aktivitätsdiagramm stellt einen fachlichen Ablauf dar. Es ist kein Datenflussdiagramm! Um den Programmablauf, der sich durch Implementierung des fachlichen Ablaufs ergibt, darzustellen, müssen weitere Diagrammtypen, wie beispielsweise Objektdiagramme, erstellt werden. In Abbildung 2.10 sind die wesentlichen Elemente eines Aktivitätsdiagramms zu sehen. Die Ovale kennzeichnen einen Arbeitsschritt (Aktivität). Die einzelnen Aktivitäten sind durch Pfeile miteinander verbunden, um eine Reihenfolge zu definieren. Entscheidungen können als spezielle Aktivität mit einem Diamantsymbol gekennzeichnet werden. Der Start und das Ende von nebenläufigen Aktivitäten wird durch *Synchronisationsbalken* dargestellt.

### 2.3.3 Design-Pattern

Zusammenfassend lässt sich sagen, dass die objektorientierte Methode aus den Teilen der objektorientierten Analyse (OOA), dem objektorientierten Design (OOD) und der objektorientierten Programmierung (OOP) besteht. Dabei wird iterativ vom Pro-

blem zur Implementierung der Lösung als Programm vorgegangen. Zur Dokumentation der einzelnen Phasen eines objektorientierten Entwicklungsprozesses wird die Symbolsprache UML verwendet.

Als Zielsetzung der objektorientierten Methode werden die Schlagworte *Wiederverwendbarkeit, Erweiterbarkeit* und *Wartbarkeit* angegeben. Diese Eigenschaften sind natürlich nicht nur von objektorientierter Software gewünscht, sondern beziehen sich allgemein auf die Softwareentwicklung. Genaugenommen sind diese Ziele auch in anderen Bereichen außerhalb der Softwareentwicklung zu finden.

Aus dem bisher Dargestellten kann aber nicht entnommen werden, wie diese Ziele erreicht werden können. Allein die Verwendung einer objektorientierten Programmiersprache wie Smalltalk, JAVA oder C++ wird keinesfalls dazu dienen können, diese Ziele zu erreichen. Wie schon dargestellt, setzt die objektorientierte Methode in der Problemanalyse an. Die Implementierung in einer objektorientierten Programmiersprache ist nur ein kleiner Teil des Vorgehens.

Die Frage, die noch offen bleibt, ist: Wie muss in der objektorientierten Methode vorgegangen werden, damit die genannten Ziele der Wiederverwendbarkeit, Erweiterbarkeit und Wartbarkeit erreicht werden?

Die Antwort darauf geben Design-Vorschläge in Form so genannter *Design-Pattern*.

Der Begriff Design-Pattern (*Entwurfsmuster*)[18] ist im Bereich der objektorientierten Softwareentwicklung noch kein standardisierter Begriff ([46]). Der Begriff Design-Pattern wurde in Anlehnung an ein Buch zum Haus- und Städtebau ([1]) auf der Konferenz für *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)* 1991 in den Bereich der Softwareentwicklung eingeführt.

Die Anfänge der objektorientierten Design-Pattern gehen zurück in die späten 70er- und frühen 80er-Jahre. Die ersten erhältlichen Frameworks wie z.B. das Model-View-Controller Framework von Smalltalk [32] waren durch ihre große Komplexität eine Belastung für die Programmierer. Die für die Frameworks erhältlichen sog. Kochbücher (*engl.: cook-book*) stellen durch ihre Rezeptsammlung eine Art von Design-Pattern zur Verwendung der Frameworks zur Verfügung.

Heute gibt es eine Vielzahl von Bücher die eine Auswahl von Design-Pattern — meist in Form eines Design-Pattern-Katalogs — anbieten. Design-Pattern sind dabei nicht ausschließlich Vorschläge, die sich in Form von Klassendiagrammen aufschreiben lassen. In [24] sind beispielsweise eine Reihe von Design-Pattern angegeben, die eine allgemeine Vorschrift darstellen können, wie ein objektorientiertes System gebaut werden sollte. Darunter fallen Design-Pattern wie:

[18] Design-Pattern: Es wird in der vorliegenden Arbeit der englische Fachbegriff *Design-Pattern* anstelle von Entwurfsmuster verwendet.

**Low Coupling — High Cohesion**

> *Koppelung zwischen Klassen soll möglichst gering sein. Zusammengehörigkeit innerhalb einer Klasse muss möglichst groß sein.*

**Polymorphism**

> *Wenn eine Auswahl getroffen werden muss, ist es besser, Polymorphie anstatt* `if`*- oder* `switch`*-Anweisungen zu verwenden.*

**Pure Fabrication**

> *Wenn es für eine Verantwortlichkeit keine geeignete Klasse im System gibt, so ist es besser, eine eigene Klasse dafür zu entwerfen, auch wenn diese Klasse ein rein künstliches Produkt ist und mit dem Problemfeld nichts zu tun hat.*

Im Standardwerk über Design-Pattern [19] werden Design-Pattern in Form eines Kataloges dargestellt. Dabei sind zu jedem Design-Pattern das Anwendungsumfeld, Konsequenzen der Anwendung und Beispiele gegeben.

Design-Pattern spielen für die vorliegende Arbeit eine wichtige Rolle und werden unter *Stand der Forschung* in Abschnitt 3.3 ausführlicher besprochen. Im Anhang A.4 auf Seite 167 sind die für die vorliegende Arbeit relevanten Design-Pattern aus dem Standardwerk [19] aufgelistet.

# Kapitel 3

# Stand der Forschung

Der Überblick über aktuelle Forschungsarbeiten auf dem Gebiet paralleler SPH-Verfahren beschränkt sich auf die Bereiche, die die vorliegende Arbeit direkt behandelt. Angrenzende Bereiche wie Konzepte für objektorientiertes Message-Passing oder Neuentwicklungen von speziellen SPH-Methoden werden hier als Grundlagen verwendet.

Im folgenden Abschnitt 3.1 wird ein Überblick über parallele SPH-Verfahren gegeben. Schwerpunkt ist dabei nicht das eigentliche SPH-Verfahren, sondern die verwendete Parallelisierungsmethode. Im Anschluss werden aktuelle Methoden und Frameworks für objektorientiertes wissenschaftliches Rechnen aus dem Bereich der Teilchensimulationsverfahren untersucht (Abschnitt 3.2). Abschließend werden im Abschnitt 3.3 aktuelle Arbeiten im Bereich der objektorientierten Design-Pattern vorgestellt.

Eine Bewertung dieser Arbeiten und die daraus folgenden Fragestellungen finden sich im Abschnitt 3.4 auf Seite 59.

## 3.1 Paralleles SPH

Auf dem Gebiet der parallelen SPH-Codes gibt es eine Reihe von Implementierungen, die von einem portablen auf MPI basierenden Code (PTreeSPH) bis zur Verwendung von Spezialhardware (GRAPE) variieren. Ein direkter Vergleich von SPH-Codes ist aufgrund der verschiedenen SPH-Verfahren schwer möglich. Die vorliegende Arbeit orientiert sich an einer allgemeinen SPH-Formulierung der Navier-Stokes-Gleichung, wie sie in [52] ausführlich erläutert wird.

Die untersuchten SPH-Codes wurden zum besseren Vergleich der im Kapitel 4 dargestellten eigenen Arbeiten auf einem Vergleichsrechner wie der Cray T3E ausgeführt. Dazu mussten einige Implementierungen angepasst werden.

### 3.1.1 PTreeSPH

Bei PTreeSPH (siehe [13]) handelt es sich um eine auch im MPI Garching[1] verwendete Implementierung eines SPH-Verfahrens mit Eigengravitation zwischen den SPH-Teilchen und variabler Smoothing-Length.

Die Implementierung ist eine Weiterentwicklung eines SPH-Codes aus dem Jahr 1989 (siehe [26]), bei dem die Berechnung der Eigengraviation über eine Baumstruktur als Erweiterung hinzugefügt wurde.

Als Kommunikationsbibliothek wurde MPI zur Parallelisierung verwendet. Die Messungen auf einer Cray T3D werden von den Autoren in [13] dargestellt. Dabei verwenden die Autoren die in Gleichung 3.1 angegebene Formel zur Berechnung der Lastverteilung auf die Prozessoren:

$$L = \frac{1}{N_p} \sum_{i=1}^{N_p} 1 - \frac{t_{\max} - t_i}{t_{\max}} = \frac{1}{N_p} \sum_{i=1}^{N_p} \frac{t_i}{t_{\max}} \tag{3.1}$$

Dabei ist $N_p$ die Anzahl der Prozessoren, $t_i$ die Laufzeit auf Prozessor $i$ und $t_{\max}$ die Laufzeit des Prozessors, der am längsten rechnet.

Bei 8 Prozessoren und $64^3$ Teilchen ermitteln die Autoren eine Lastverteilung von 89,5%. Dabei wird die Gravitationsberechnung über die Baumsuche mit 96,0% besser lastverteilt, der parallele Mehraufwand schneidet aber mit 62,6% schlechter ab.

In der vorliegenden Arbeit werden zum Vergleich von parallelen Programmläufen Speedup-Kurven bevorzugt. Eine Angabe der Effizienz von parallelen Programmen über den Begriff der Lastverteilung ist schwer greifbar.

Abbildung 3.1 zeigt eine im Rahmen der vorliegenden Arbeit gemachte Messung des PTreeSPH-Programms auf der Cray T3E mit den oben genannten Werten von $64^3$ Teilchen, wobei die Anzahl der Prozessoren variiert wurde. Es muss eine Mindestanzahl von 8 Knoten der Cray T3E verwendet werden, um die Teilchenzahl auf dem Hauptspeicher verteilen zu können.

Es ist zu sehen, dass die parallele Effizienz des PTreeSPH-Programms zwischen 32 und 64 Knoten auf unter 50% abfällt. Diese Messungen können mit den eigenen Arbeiten (siehe Kapitel 4) verglichen werden. Aus diesen Messungen kann entnommen werden, dass die Berechnung der Gravitation sehr gut skaliert. Die SPH-Berechnung, die den größten Teil der Rechenzeit in Anspruch nimmt, skaliert aber deutlich schlechter.

[1] Siehe `http://www.mpa-garching.mpg.de`

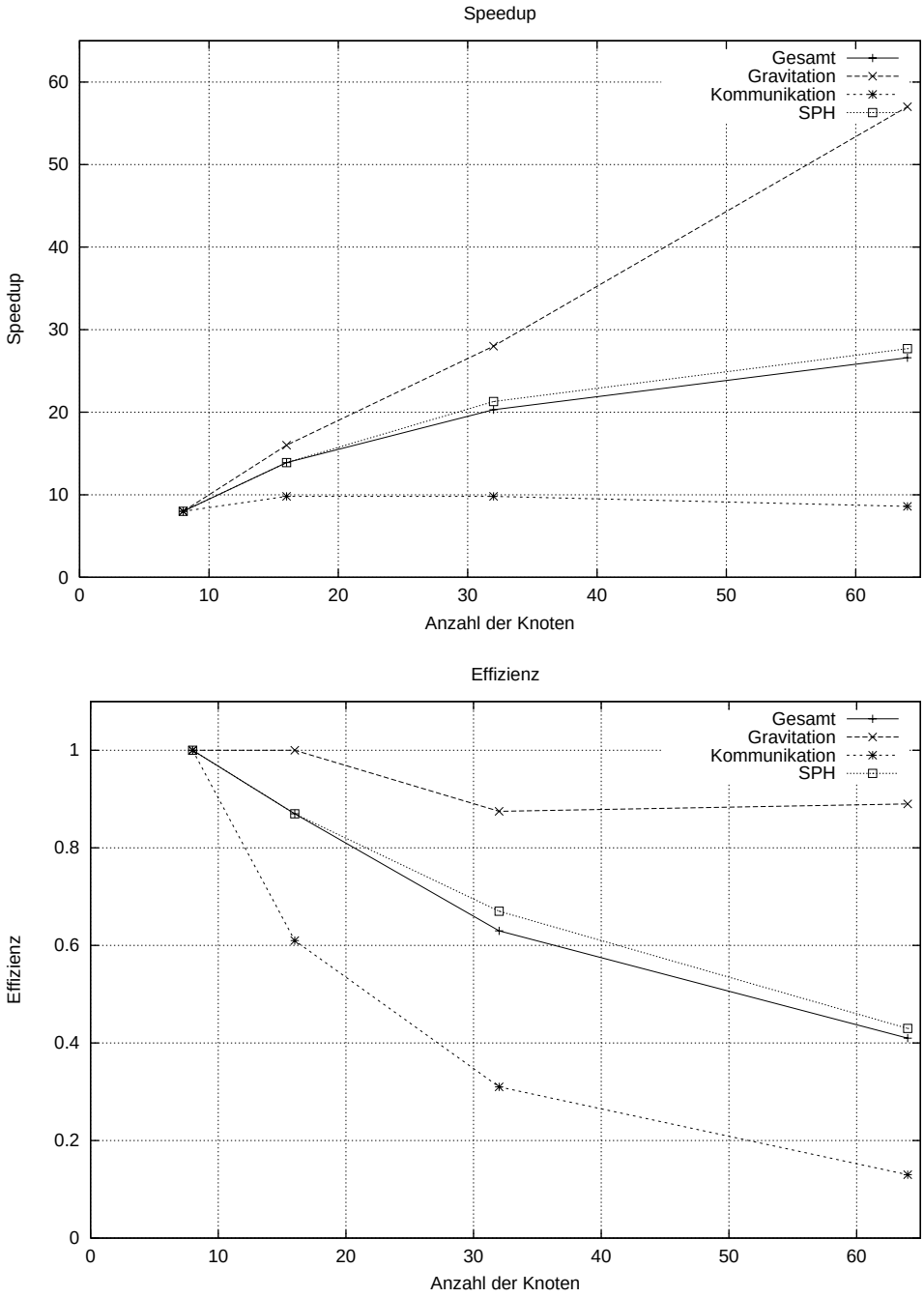



Abbildung 3.1: Gemessener Speedup und Effizienz des PTreeSPH-Codes auf Cray T3E mit $64^3$ Partikeln. (Bei 8 Knoten wird 100% Effizienz angenommen.)

### 3.1.2 GRAPE

Für SPH-Verfahren stellt die Nachbarschaftssuche ein zentrales Problem dar. Zu einem gegebenen SPH-Teilchen müssen alle mit diesem Teilchen in Wechselwirkung stehenden Teilchen der Umgebung gefunden werden. Die Nachbarschaftssuche ist

im ungünstigsten Fall ein $O(N^2)$-Problem. Unter der Voraussetzung, dass nur kurzreichweitige Kräfte von Interesse sind, kann die Nachbarschaftssuche durch einen $O(N)$-Algorithmus berechnet werden (siehe [2]).

Bei der Berechnung gravitativer Kräfte zwischen Teilchen liegt grundsätzlich ein $O(N^2)$-Problem vor (außer bei dem Fall der näherungsweisen Berechnung mittels eines Tree-Codes, das ein $O(N \log N)$-Problem darstellt). Zu diesem Zweck wurde von Fukushige et al. (1991) die Spezialhardware GRAPE (GRAvity PipE) entwickelt. Umemura et al. zeigen in [58], dass es mittels dieser Spezialhardware möglich ist, auch SPH-Probleme zu beschleunigen, wobei die von der Hardware berechnete Liste der wechselwirkenden Teilchen verwendet wird. Ein offensichtlicher Nachteil dieser Methode ist eben die Verwendung der Spezialhardware GRAPE, die den Code nicht portabel macht und die häufig nicht zur Verfügung steht.

GRAPE wird im MPI in Garching in Zusammenhang mit dem oben beschriebenen PTreeSPH-Programm verwendet. Vergleichsmessungen sind hier nicht direkt möglich.

### 3.1.3 Parallel Tree SPH (Lia et al.)

In [37] gibt Lia et al. eine Variante des PTreeSPH-Verfahrens an, das direkt auf der SHMEM-Bibliothek der Cray T3E aufsetzt. Zur Lastverteilung wird das gleiche Verfahren verwendet wie in Formel 3.1 angegeben. Die Lastverteilung liegt bei bis zu 128 Knoten auch über 90%. Die Autoren von [37] ermitteln die Laufzeiten ihres SPH-Programms mit bis zu 32 Knoten. Die Diagramme zeigen eine lineare Abnahme der Rechenzeit, wobei allerdings der Speedup bei einer Steigerung der Knoten von 8 auf 32 nur knapp den Faktor 2 beträgt.

Da nicht klar hervorgeht, wie dieses Laufzeitverhalten zustande kommt — es kann nur vermutet werden, dass die Lastverteilung mehr Rechenaufwand benötigt —, wird hier auf eine Vergleichsgraphik verzichtet und auf [37] verwiesen.

### 3.1.4 MEMSY-SPH

Ein SPH-Verfahren, das mit dem in der vorliegenden Arbeit verwendeten Verfahren verglichen werden kann, wurde von der Universität Erlangen auf der MEMSY-Architektur parallelisiert.

MEMSY steht für *Modular Expandable Multiprocessor System* und ist ein von der Universität Erlangen entwickelter Parallelrechner mit verteiltem Speicher. Zu den Hardwarebesonderheiten des MEMSY-Computers verweisen wir auf [57]. Die MEMSY-Knoten können bis zu 4 Prozessoren aufnehmen. Die 20 Knoten des

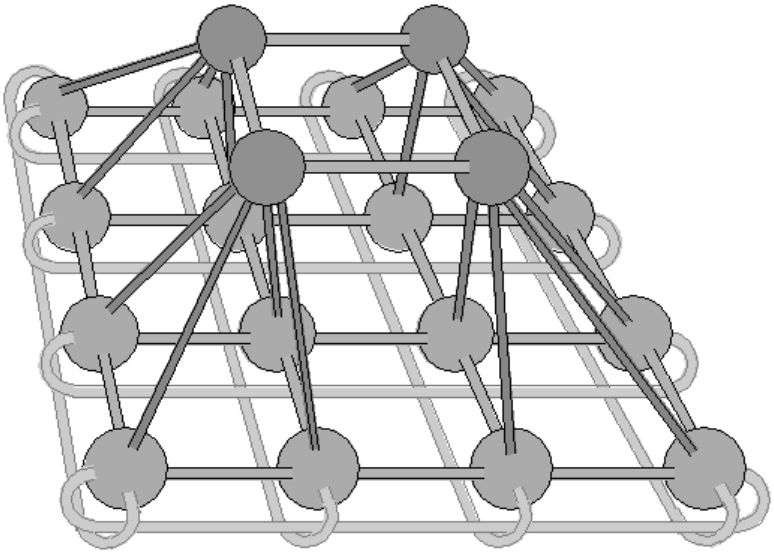


Abbildung 3.2: Schema der Knotenanordnung der MEMSY-Architektur.

MEMSY-Computers sind in zwei Ebenen aufgeteilt, wie in Abbildung 3.2 skizziert. Die Knoten sind über *Coupling Units*, ein vereinfachter Kreuzschienenverteiler mit mehreren benachbarten Speichersegmenten, miteinander verbunden. Jeder Knoten hat über zwei *Coupling Units* Zugriff auf 8 Speichersegmente. Die Knoten sind an den Kanten mit den gegenüberliegenden Knoten zu einem Torus verbunden.

Der SPH-Code für den MEMSY-Computer muss die spezielle Hardware des MEMSY-Computers berücksichtigen, ist aber gut auf andere Systeme, wie z.B. die Cray T3E, zu portieren. Zur Portierung auf die Cray T3E müssen lediglich die bidirektionalen Warteschlangen zum Zugriff auf die Speichersegmente des MEMSY-Computers in eine *Barrier*-Synchronisation umgewandelt werden (siehe [28]).

**Laufzeitmessung auf dem MEMSY-Computer** Aus den auf der MEMSY-Architektur durchgeführten Testreihen konnte eine Formel für die Laufzeitentwicklung mit den beiden Anteilen *Kommunikation* und *SPH-Berechnung* aufgestellt werden.

$$T(N,k) = c_1\frac{N}{k} + c_2(\frac{N}{k})^2 \tag{3.2}$$

In Gleichung 3.2 ist $N$ die Teilchenzahl und $k$ die Kantenlänge der Prozessormatrix. Damit ist $k^2$ die Anzahl der beteiligten Knoten des MEMSY-Computers. Der erste Term in Gleichung 3.2 beinhaltet dabei die Kommunikationsanteile und der zweite Term die SPH-Berechnungen.

Für die Laufzeiten wird für $k = 4$ (entspricht 16 Knoten) für 10.000 Teilchen eine parallele Effizienz von 85% angegeben. Für $k = 16$ (entspricht 256 Knoten) würde

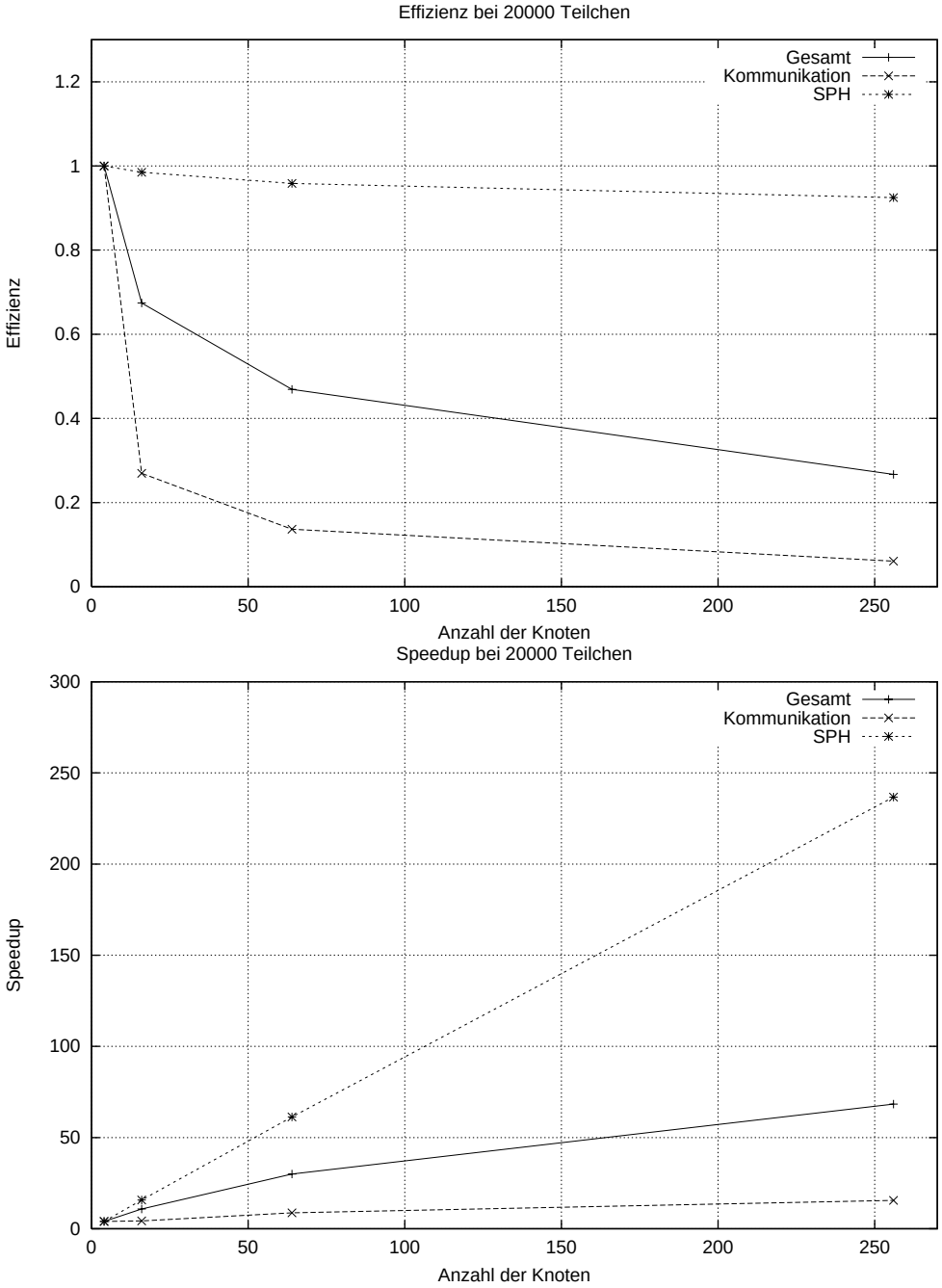



Abbildung 3.3: Vergleichsmessung des MEMSY-Verfahrens auf der Cray T3E mit 20.000 Teilchen.

daraus eine errechnete parallele Effizienz von 57% folgen. Da der MEMSY-Computer aber nur über 20 Knoten verfügt, ist eine Messung dieser Vorhersage nicht möglich.

**Vergleichsmessung auf Cray T3E** Um einen Vergleich mit der vorliegenden Arbeit möglich zu machen, wurde durch das oben beschriebene Vorgehen (siehe auch [28]) eine Portierung des MEMSY-SPH-Codes auf der Cray T3E implementiert und ver-

messen.

In Abbildung 3.3 ist die parallele Effizienz und der Speedup mit 20.000 Teilchen bei bis zu 256 Prozessoren aufgezeigt. Es ist zu erkennen, dass der Kommunikationsanteil dazu beiträgt, die Effizienz bei 256 Knoten auf unter 30% zu drücken. Bei 32 Knoten liegt die parallele Effizient noch deutlich über 60%.

## 3.2 Paralleles objektorientiertes wissenschaftliches Rechnen

Außer den speziellen Implementierungen von SPH-Verfahren existieren verschiedene Programmpakete mit Sammlungen von Algorithmen und Verfahren für wissenschaftliches Rechnen. Die vorliegende Arbeit beschäftigt sich mit parallelen, objektorientierten Methoden für Teilchensimulationsverfahren. Demnach werden CFD-Simulationspakete[2], die mit Gittermethoden arbeiten, Simulationspakete aus anderen Bereichen der Physik und Programmpakete aus Bereichen der Mathematik, die zwar interessant sind, aber nicht direkt mit der Fragestellung der vorliegenden Arbeit in Berührung stehen, keine Beachtung finden.

Die zurzeit wichtigsten Entwicklungen im Bereich der parallelen, objektorientierten Teilchensimulationsverfahren kommen aus dem Los Alamos National Laboratory (POOMA) oder sind in bekannten Paketen wie NETLIB zusammengefasst. Das jüngste Projekt, eine Bibliothek von Simulationsverfahren zu schaffen, stellt der Cactus Code Server[3] dar. Diese Programmpakete und Frameworks werden in den nächsten Abschnitten untersucht.[4]

### 3.2.1 POOMA

POOMA steht für *Parallel Object-Oriented Methods and Applications* und wird seit 1995 vom Los Alamos National Laboratory (LANL[5]) entwickelt.

POOMA ist ein objektorientiertes Framework für wissenschaftliches Rechnen auf Höchstleistungs-Parallelrechnern. POOMA ist eine C++-Klassenbibliothek, von der der Anwendungsprogrammierer durch Erweiterung und Abstraktion (Ableitung) die eigenen Anwendungen für Parallelrechner implementieren kann.

[2] CFD: Computational Fluid Dynamics

[3] Siehe: `http://www.cactuscode.org`

[4] Kommerzielle Software ist im Bereich von parallelen SPH-Verfahren, im Gegensatz zu gitterbasierten CFD-Codes, noch nicht erhältlich. SPH ist ein noch sehr junges Simulationsverfahren und steht selbst noch im Blickpunkt der Forschung.

[5] Siehe: `http://www.acl.lanl.gov/pooma`

LANL gibt als Zielsetzung für POOMA folgende Punkte an:

- Portabler Code für serielle und parallele Computerarchitekturen ohne Code-Anpassungen.
- Wiederverwendbare Komponenten für beschleunigte Anwendungsentwicklung (*engl.: rapid prototyping*).
- Problemfeld- und anwendungsorientiertes Framework.
- Schnellere Entwicklungszeit vom Problem zum parallelen Programm.

Zu Anfang[6] bestand POOMA aus einer Methodensammlung für Simulationen auch aus dem Bereich der Hydrodynamik und der Teilchensimulationsmethoden. Die Anwendungsgebiete von POOMA sind partielle und gewöhnliche Differentialgleichungen mit einer Vielzahl von verschiedenen Anfangsbedingungen und Raumdimensionen. Teilchen-Feld-Wechselwirkungen werden dabei durch Methoden wie Particle-In-Cell (PIC) und Particle-Particle-Particle-Mesh (PPPM) als Kombination von molekulardynamischen Methoden und PIC dargestellt (siehe [4]). Die Version 2 aus dem Jahr 1999 ergänzt die Funktionalität um eine datenparallele Multithread-Umgebung für Maschinen mit gemeinsamem Hauptspeicher. Im August 2000 wurde POOMA in der Version 2.3.0 auch für Maschinen mit verteiltem Hauptspeicher, über eine auf MPI basierende Message-Passing-Bibliothek (CHEETA-Layer), ausgebaut (siehe [35]).

POOMA macht extensiven Gebrauch der *Standard Template Library* (STL) [41] von C++. STL verwendet Templates (parametrisierbare Klassen), um Container — wie Vektoren und Listen — von zugehörigen Algorithmen — Suchen, Finden, Sortieren — zu trennen. Verbunden werden Container und Algorithmen über Iteratoren, die die Container lesen können, ohne den expliziten Typ des Containers an die Algorithmen preiszugeben.

Um auch als C++-Klassenbibliothek hohen Effizienzanforderungen zu genügen, verwendet POOMA auch weniger bekannte Konzepte von C++. Dazu zählen spezifische Implementierungen von überladenen Operatoren, Kopier-Konstruktoren und Trait-Templates (siehe [42]).

**Parallelisierungskonzepte von POOMA** Das Grundelement der Parallelisierung von POOMA heißt *Context*. Ein *Context* besteht dabei aus einem lokalen Adressraum im Speicher des Rechners. In einem *Context* können mehrere Threads laufen, die alle auf dem gleichen Rechner (oder Knoten ab Version 2.3.0) laufen müssen. Ein *Context*

[6] Erste POOMA-Implementierungen werden als R1 bezeichnet.

```
1   template<int D, class T>
2   T accumulate(
3       const ConstArray<D, T, MultiPatch<UniformTag,Brick>> & x
4   ){
5       // Get the GridLayout from the array.
6       const GridLayout<2>& layout = x.message(GetGridLayoutTag<2>());
7       // Find the number of patches.  We'll have one thread per patch.
8       int patches = layout.size();
9
10      // An array of thread ids.
11      pthread_t *ids = new pthread_t[patches];
12
13      [ ... Code Iteriert über die patches ...]
14      // Wait for all the threads to finish.
15      // Get the sum from each, and accumulate that
16      // in this thread.
17      T sum = 0;
18      for (int j=0; j<c; ++j)
19      {
20          // Wait for a given thread to finish.
21          // cout << "join" << endl;
22          void * v;
23          pthread_join(ids[j], &v);
24
25          [ ... Ergebnisse werden eingesammelt und akkumuliert ... ]
26      }
27      // Return the full sum.
28      return sum;
29  }
```

Abbildung 3.4: Kontrollfluss-Parallelität mit POOMA.

kann nicht über mehrere Knoten verteilt werden, wohl können aber mehrere *Context* auf einem Rechner (Knoten) leben.

In der Version 2.0 unterstützt POOMA ein auf Threads basierendes Parallelisierungskonzept. Dabei laufen die Threads innerhalb eines *Context*. POOMA folgt dabei dem SPMD-Paradigma (siehe Abschnitt 2.2.2 auf Seite 28). Eine Version des Programmcodes läuft in einem *Parse*-Thread, und datenparallele Ausdrücke können mittels Iteratoren nebenläufig durch andere Threads ausgeführt werden. In der Multithread-Version sehen alle Threads die selben (globalen) Datenstrukturen, d.h. es wird ein datenparalleles Programmiermodell mit globalem Adressraum verwendet (vgl. Abschnitt 2.2.2 auf Seite 25).

Seit August 2000 unterstützt POOMA in der Version 2.3.0 auch Maschinen mit verteiltem Speicher. Das Konzept der *Context* mit mehreren Threads wird ausge-

weitet auf ein Modell, bei dem mehrfache *Context* (*engl.: multiple context*) auf die Knoten eines Parallelrechners verteilt werden. Auch hier folgt POOMA dem SPMD-Paradigma und startet dasselbe Programm in den *Context* der einzelnen Knoten. Die verteilten *Context* kommunizieren nun über ein darunter liegendes Message-Passing-Layer (CHEETAH). Dadurch können die verteilten *Context* auf Teile einer Datenstruktur — POOMA sprechen von Array — zugreifen, die sich nicht im lokalen Speicher des eigenen Knoten befinden.

Eine Einschränkung der Version 2.3.0 ist noch, dass innerhalb von verteilten *Context* nur jeweils ein Thread laufen kann. Die Möglichkeit, auch Multithread-*Context* zu verteilen soll in der Folgeversion 2.4 nachgebessert werden.[7]

Beiden Parallelisierungskonzepten von POOMA ist gemeinsam, dass eine datenparallele Programmierung angenommen wird. Kontrollflussorientierte Parallelisierung kann nur über explizites Programmieren der POSIX-Threads-Schnittstelle durch den Anwendungsentwickler implementiert werden. Abb. 3.4 auf der vorherigen Seite zeigt einen Programmausschnitt eines C++-Programms mit POOMA und POSIX-Threads. Diese Art der Kontrollflussparallelität ist nur innerhalb eines *Context* möglich und kann nicht auf die Knoten von nachrichtenorientierten Maschinen verteilt werden.

### 3.2.2 Cactus Code Server

Cactus [7] wird seit 1999 als allgemeines Framework für Computational Physics zur Lösung von partiellen Differentialgleichungen aus der numerischen Astrophysik entwickelt. Als prominentes Beispiel sei der in der Dezemberausgabe von 1999 des IEEE-Magazins *Computer* erschienene Artikel über die Lösung der Einstein-Gleichungen erwähnt.

Cactus hat den Namen von der Idee, in ein *Fleisch* (*engl.: flesh*) von Basisfunktionalität *Stacheln* (*engl.: thorns*) von Speziallösungen zu stecken. Catus ist damit modular und verspricht leichtere Wartbarkeit und Benutzerfreundlichkeit. Dabei verwendet Cactus aus der objektorientierten Programmierung entlehnte Konzepte (Cactus spricht von: [... *object-oriented inspired features*], siehe `http://www.cactuscode.org`).

Es können über Cactus *Thorns* in verschiedenen Sprachen wie `C++`, `C`, `FORTRAN77` miteinander verbunden werden. Parallelität erreicht Cactus über einen Parallelitäts-*Thorn*, der über eine einheitliche Schnittstelle im *Flesh* erreichbar ist.

[7] Anm.: Die Einführung von verteilten *Context* und Message-Passing hatte LANL für 1999 angekündigt, aber erst Ende 2000 veröffentlicht.

Unter den Zielsetzungen versteht Cactus die Lehren der *Grand-Challange*-Rechnungen der letzten Jahre:

- Es gibt keine Infrastruktur für eine Zusammenarbeit bei High-Performance-Computing-Problemen.
- In der Regel sind Programmierer keine Informatiker (*engl.: computer scientists*).
- Sprachbarrieren: Mathematiker, Physiker und Informatiker verwenden unterschiedliche Ausdrücke, Symbole und Vokabeln.
- Code-Fragmente, Codierungs-Stile und Funktionseinheiten wie Prozeduren passen häufig nicht zusammen.
- Gewachsener Code kann oft nicht gut auf neue Technologie (wie z.B. MPI) portiert werden.

Cactus verspricht mit dem modularen *Flesh*-*Thorn*-Konzept, für diese Probleme eine Abhilfe zu schaffen.

Die einzelnen *Thorns* bieten eine Reihe von Problemlösern im Bereich der partiellen Differentialgleichungen an. Dabei werden gitterbasierte Lösungsmethoden verwendet, auf die an dieser Stelle nicht weiter eingegangen wird (siehe: `http://www.cactuscode.org`).

**Parallelisierungskonzepte von Cactus** Cactus geht von einem nachrichtenorientierten Programmiermodell aus. Es wird eine Maschine mit verteiltem Hauptspeicher angenommen. Die Simulationsdaten werden nach einem Verfahren in ein globales Gitter (*engl.: grid*) unterteilt. Jeder Simulations-*Thorn* bekommt von der ParallelsSchnittstelle im *Flesh* einen Teil des globalen Gitters zur Berechnung zugewiesen.

Das Verfahren zur Aufteilung der Daten in ein Gitter (*engl.: domain-decomposition*) ist wiederum in einem *Thorn* implementiert. Dadurch kann für jede Problemstellung ein spezifischer Verteilungs-*Thorn* implementiert werden, der über eine einheitliche Schnittstelle im *Flesh* angesprochen wird.

In einem Application-*Thorn* können Synchronisations- und Kommunikationsprimitiven von Cactus verwendet werden. Diese sind den Kommunikationsprimitiven von MPI entlehnt.

- `CCTK_MyProc(cctkGH)` — die eigene Prozessornummer.
- `CCTK_nProcs(cctkGH)` — Gesamtanzahl aller Prozessoren.
- `CCTK_Barrier(cctkGH)` — Barrier-Synchronisation über alle Prozessoren.

- `CCTK_SyncGroup(cctkGH, groupname)` — Gruppenkommunikation zum Austausch von Randgebieten der verteilten Daten-Gitter.

Die aktuelle Version 4 beta 6 von Cactus unterstützt keine Parallelisierung durch das Threads-Konzept auf Maschinen mit gemeinsamen Speicher.

### 3.2.3 NETLIB

Das Netlib-Verzeichnis enthält frei verfügbare Software, Dokumente und Datenbanken aus dem Bereich der numerischen Mathematik und des wissenschaftlichen Rechnens. Netlib wird von den AT&T Bell Laboratories, der University of Tennessee (UTK) und dem Oak Ridge National Laboratory (ORNL) gepflegt. Zugriff auf das Netlib-Verzeichnis ist über `http://www.netlib.org` zu erhalten.

Zu den bekanntesten Softwarepaketen unter Netlib zählen die *Basic Linear Algebra Subprogramms* (BLAS). BLAS bietet Vektor- und Matrix-Operationen und ist auch in objektorientierter Version erhältlich. Da die vorliegende Arbeit nicht auf linearer Algebra aufbaut, wird hier nicht weiter auf BLAS eingegangen.

**PETSc** Unter Netlib ist eine Vielzahl von Problemlösern für partielle Differentialgleichungen zu finden, die in der Regel auf Gittermethoden aufbauen. Zu nennen ist hier das *Portable, Extensible Toolkit for Scientific Computation* (PETSc), das eine Sammlung von verschiedenen Codes zur Lösung von partiellen Differentialgleichungen anbietet. PETSc setzt auf MPI auf und nimmt ein nachrichtenorientiertes Parallelisierungskonzept an. Die Lösungsmethoden sind matrixorientierte Gleichungslöser für partielle Differentialgleichungen. PETSc ist auf viele Rechnerarchitekturen portiert, hat aber bei z.B. der Cray T3E noch Probleme mit der MPI-Implementierung.

**Overture** Als weiteres Beispiel für aktuelle Softwarepakete im Bereich des objektorientierten wissenschaftlichen Rechnens sei Overture genannt. Overture wird vom Lawrence Livermore National Laboratory (LLNL)[8] entwickelt.

Overture verwendet finite Differenzen und finite Volumenmethoden, kombiniert mit adaptiven Gitterverfeinerungen (*engl.: block-structured adaptive mesh refinement (AMR)*). Diese Methoden dienen ebenfalls der Lösung von partiellen Differentialgleichungen im Bereich der Hydrodynamik.

Overture verwendet objektorientierte Konzepte zur Darstellung der Simulationsdaten. Dabei werden die Datenstrukturen (Overture spricht von *Arrays*) mit den dazugehörigen Operationen als eine Klasse verstanden. Overture bietet nun serielle wie

[8] Siehe `http://www.llnl.gov`

auch parallele Versionen der Operationen auf einer Datenstruktur an. Die parallelen Versionen der Operationen können nun unter einem datenparallelen Paradigma ausgeführt werden.

Systemgrundlage zur Ausführung der datenparallelen Anweisungen ist MPI oder PVM.

## 3.3 Design-Pattern

Alle im Abschnitt 3.2 vorgestellten Methoden für wissenschaftliches Rechnen haben einen objektorientierten Ansatz. Auffallend sind dabei die Unterschiede zwischen den Konzepten. Die Vorschläge, wie objektorientiert zu programmieren sei, variieren von den sehr speziellen, technischen Vorschlägen der Trait-Templates von POOMA über die *vom Objektorientierten inspirierten* modularen Ansätze des Cactus-Projekts bis hin zu den Konzepten, serielle und parallele Ausführung eines Programms durch Überladen von Operationen zu realisieren (Overture).

Nach dem Überblick über aktuelle Konzepte des objektorientierten wissenschaftlichen Rechnens bleibt die Frage: Wie soll für wissenschaftliche Anwendungen objektorientiert programmiert werden?

Im vorliegenden Abschnitt werden aktuelle Konzepte aus dem Bereich der objektorientierten Programmierung vorgestellt. In den letzen Jahren hat sich auf die Frage nach dem *Wie* der objektorientierten Programmierung eine Antwort durchgesetzt: *Design-Pattern.*

### 3.3.1 Überblick

Die Herkunft des Begriffs *Design-Pattern* wurde bereits in Abschnitt 2.3.3 auf Seite 38 erläutert. Im Folgenden wird ein kurzer Abriss über die Entwicklung der Design-Pattern gegeben.

**Formal Contracts** Da vorhandene Analyse- und Design-Methoden Anfang der 1990er Jahre zur Entwicklung von wiederverwendbaren Softwarearchitekturen nicht ausreichten, bildeten sich detailliertere Design-Pattern heraus. Dazu zählen die Untersuchungen von Helm [25], der durch *Formal Contracts* eine formale Beschreibung der Wechselwirkungen zwischen Objekten von verschiedenen Klassen entwarf.

**Object-Oriented Patterns** In dem Artikel über objektorientierte Entwurfsmuster (*engl.: Object-Oriented Patterns*) von Peter Coad in [8] aus dem Jahre 1992 werden

Basismuster zur Vererbung und Interaktion, Muster zur Strukturierung von Softwaresystemen sowie MVC-verwandte[9] Muster vorgestellt.

**Coding Patterns** Coplien schreibt in [11] *[„. . . ] most of what guides the structure of programs, and therefore of the systems we build, is the* styles *and* idioms *we adopt to express design concepts*“. Coplien vertritt die Einstellung, dass eine Vereinheitlichung in den Codierungsstilen, d.h. der Quellcodestruktur und Namen, sowie eine Bereinigung von Mängeln der objektorientierten Sprachen eine Weiterentwicklung darstellen. Im weiteren Sinne zählen also auch die Codierungsmuster von Coplien zu den Design-Pattern.

**Framework Adaption Cookbook** Die Tradition der schon in Abschnitt 2.3.3 genannten *Cookbooks* wird durch das *Framework Adaption Cookbook* von Ralph Johnson [30] fortgeführt. In *Cookbooks* findet der Programmierer eine Rezeptsammlung von wiederverwendbaren Lösungen zu häufigen Problemen. Durch Dokumente im Hypertext-Format werden die Lösungen der *Cookbooks* leicht wiederfindbar dargestellt.

### 3.3.2 Design-Pattern-Kataloge

Mit dem Buch *Design Pattern* von Erich Gamma et al. (siehe [19]) aus dem Jahr 1994 bekamen die Design-Pattern mit dem Design-Pattern-Katalog eine von da an immer wieder verwendete Form.

Als Definition von Design-Pattern wird auch in [19] auf Alexander verwiesen: *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice* (siehe [1]).

In [19] werden zur Beschreibung von Design-Pattern vier Schritte angegeben:

1. Name des Design-Pattern:

   Der Name sollte die Problemstellung, das Problemumfeld und die Auswirkungen des Design-Pattern in einem Wort zusammenfassen. Anders ausgedrückt: Der Name soll das Design-Pattern sinnvoll beschreiben.

   Dadurch sind Fachleute in der Lage, Problembeschreibungen und Lösungsstrategien durch die Benennung von Design-Pattern einfach und gezielt zu besprechen.

[9] MVC steht für *Model-View-Controller*; ein Design-Pattern aus Smalltalk zum Entwurf von graphischen Benutzeroberflächen.

2. Problemumfeld:

   Zur Beschreibung eines Design-Pattern gehört auch die Angabe über den Kontext, in dem das Design-Pattern Verwendung findet. Hier wird eine typische Aufgabenstellung, ggf. mit einem Beispiel, wie es nicht gemacht wird, angegeben. Eine Liste von Vorbedingungen für die Anwendung für ein Design-Pattern findet hier ihren Platz.

3. Lösung:

   Der Lösungsweg, den das Design-Pattern vorschlägt, wird hier erklärt. Damit ist *nicht* die Angabe einer Implementierung, oft nicht einmal ein Klassendiagramm oder Vergleichbares gemeint.

   Für viele Lösungen mit Design-Pattern bietet es sich aber an, die Lösung in Form von UML-Diagrammen zu notieren. Weiter unten in diesem Abschnitt werden Beispiele für Design-Pattern mit und ohne konkrete UML-Diagramme gegeben.

4. Konsequenzen:

   Den Abschluss der Beschreibung eines Design-Pattern bildet die Aufzählung der Konsequenzen, die sich aus der Verwendung dieses Design-Pattern ergeben. Ein Design-Pattern ist keine Universallösung, sondern kann auch Nachteile für andere Problembereiche mit sich ziehen. Diese Vor- und Nachteile müssen erläutert werden.

In [19] werden zur Beschreibung eines Design-Pattern noch eine Reihe weiterer Unterpunkte verwendet. In der vorliegenden Arbeit werden Design-Pattern mit den oben genannten vier Punkten beschrieben.

Die in [19] beschriebenen Design-Pattern werden oft als *GoF*-Pattern bezeichnet. *GoF* steht dabei für *Gang of Four*; die vier Autoren dieses Buches: Gamma, Helm, Johnson, Vlissides.

Eine extensive Sammlung von Design-Pattern findet sich in den Bänden des Design-Pattern-Katalogs: *Pattern Languages of Program Design* (siehe [12] und [59]).

### 3.3.3 Auswahl einiger Design-Pattern

Im Folgenden werden die für die vorliegende Arbeit relevanten Design-Pattern vorgestellt. In [24] werden unter der Bezeichung *GRASP* (*engl.: General Responsibility Assignment Software Patterns*) einige grundlegende Design-Pattern zusammengefasst. Darunter fallen:

- **Low Coupling/High Cohesion**

  *Problem:* Klassen, die so eng an andere Klassen gekoppelt sind oder keinen sinnvollen inneren Zusammenhang haben, machen ein Design verworren und schwer erweiterbar.

  *Lösung:* Einige GRASP-Pattern anwenden, sodass eine lose Kopplung und ein enger innerer Zusammenhalt erreicht wird.

  *Konsequenz:* Es sind Erfahrungswerte, die entscheiden, ob eine Klasse eine zu große Kopplung hat. Es ist nicht immer eine zu große Kopplung das eigentliche Problem. Wenn das Design-Pattern greift, wird das System leichter erweiterbar und modifizierbar.

- **Pure Fabrication**

  *Problem:* Eine Verantwortlichkeit muss einer Klasse zugewiesen werden. Diese Zuweisung würde aber das Design-Pattern *Low Coupling/High Cohesion* verletzen.

  *Lösung:* Ein neue Klasse wird eingeführt, die keinen direkten Bezug zum realen Problemraum hat. Diese Klasse ist *rein künstlich* (*engl.: pure fabrication*) und hat nur die Funktion, bei Übernahme der Verantwortlichkeit das Design-Pattern der *Low Coupling/High Cohesion* nicht zu verletzen.

  *Konsequenz:*

  – Die Wiederverwendbarkeit der Klassen im Design wird erhöht, da durch *Pure Fabrication* eine feinkörnige Zuweisung von Verantwortlichkeiten begünstigt wird.

  – *Pure Fabrication* erhält die lose Kopplung und den starken inneren Zusammenhalt der Klassen im Design.

- **Controller**

  *Problem:* Ein System empfängt Ereignisse (*engl.: events*) und stellt Objekte zu deren Verarbeitung zur Verfügung. Die Ereignis-Quelle soll nicht direkt mit den Ereignis-Verarbeitern gekoppelt werden.

  *Lösung:* Es wird eine Controller-Klasse eingeführt, die die Ereignisse an die verarbeitenden Objekte weiterleitet.

  *Konsequenz:* Die Ereignis-Quellen und Ereignis-Verarbeiter-Klassen sind entkoppelt und wiederverwendbar. Die Controller-Klassen haben eine starke Kopplung zu Quellen und Verarbeitern und sind schlecht wiederverwendbar.

- **Law of Demeter**

  *Problem:* Zwei Klassen, die keinen Grund haben, direkt voneinander Kenntnis zu haben, oder in irgendeiner Weise gekoppelt sind, sollten auch nicht miteinander interagieren. (*Sprich nicht mit Fremden.*)

  *Lösung:* Objekte sollten Operationen nur auf Objekten aufrufen, die zueinander in einer der folgenden Beziehung stehen:

  – dasselbe Objekt (`this`)
  – ein Objekt, das als Parameter einer Operation übergeben wurde (Nachricht)
  – ein Objekt, das aggregiert oder assoziiert ist
  – ein Objekt, das erzeugt wurde

  *Konsequenz:* Dieses Design-Pattern hält die Kopplung zwischen Klassen niedrig. Es wird Overhead beim Aufruf von Operationen zwischen Objekten eingefügt.

In [19] werden die als *GoF*-Pattern bekannten Design-Pattern in drei Kategorien unterteilt: *Erzeuger* (*engl.: creational patterns*), *Struktur* (*engl.: structural patterns*) und *Verhalten* (*engl.: behavioral patterns*).

- **Singleton** — Erzeuger

  Ein UML-Diagramm des Singleton-Patterns findet sich im Anhang A.14 auf Seite 169.

  *Problem:* Von einer Klasse darf es nur eine einzige Instanz (Objekt) im System geben. (Beispiel: exklusiver Zugriff auf Rechenressource.)

  *Lösung:* Ein Klasse stellt sicher, dass nur eine Instanz existiert. Die Möglichkeiten der Implementierung sind stark sprachabhängig.

  *Konsequenz:*

  – Kontrollierter Zugriff auf Instanz.
  – Es kann leicht eine bestimmte Anzahl $N$ von Instanzen zugelassen werden.
  – Flexibler als statische Klassenoperationen.

- **Composite** — Struktur

  *Problem:* Ein Problem lässt sich in eine baumartige Struktur untergliedern, bei dem ein Teil als Summe seiner Einzelteile dargestellt werden kann. Beispiel: Eine Linie ist ein Bildelement, ein Bildelement kann aus mehreren Bildelementen aufgebaut sein.

*Lösung:* Ein UML-Diagramm des Composite-Patterns findet sich im Anhang A.13 auf Seite 168.

*Konsequenz:*

- Überall dort, wo im Programm ein Objekt des Composite-Patterns erwartet wird, kann entweder ein zusammengesetztes oder ein elementares Objekt eingesetzt werden.
- Die Verwendung von zusammengesetzten Strukturen wird vereinfacht.
- Neue elementare Teile einer Struktur können leicht hinzugefügt werden.

- **Strategy** — Verhalten

  *Problem:* Aus einer Familie von Algorithmen zur Lösung eines Problems muss einer ausgewählt werden können.

  *Lösung:* Ein UML-Diagramm des Strategy-Patterns findet sich im Anhang A.12 auf Seite 167.

  *Konsequenz:*

  - Algorithmen werden hierarchisch in Familien strukturiert.
  - Eine Alternative zum Vererbungsmechanismus. Vererbung stellt eine starke Kopplung zwischen Klassen (Super- und Sub-Klasse) dar, die in manchen Situationen nicht erwünscht ist.
  - Das Strategie-Pattern vermeidet aufwendige `if-then-else`-Konstrukte.

- **Builder** — Erzeuger

  *Problem:* Die Erzeugung eines Objekts ist ein komplexer Vorgang, der von der Darstellung des Objekts getrennt werden sollte.

  *Lösung:* Ein UML-Diagramm des Builder-Patterns findet sich im Anhang A.16 auf Seite 171.

  *Konsequenz:*

  - Die interne Darstellung eines Produkts (Objekts) kann variiert werden.
  - Programmcode für Erzeugung und Repräsentation eines Objekts werden getrennt.
  - Der Konstruktionsprozess für ein Objekt kann besser strukturiert und kontrolliert werden.

Design-Pattern im Bereich des parallelen Rechnens werden in den Bänden [12], [59] und [36] angegeben. Diese Design-Pattern beschäftigen sich mit grundlegenden Problemen des parallelen Rechnens wie Synchronisation, Deadlock, Datentransfer über Netzwerke, verteilte Anwendungen.

Design-Pattern, die direkt für numerische, wissenschaftliche Simulationen gedacht sind, sind nicht explizit dokumentiert.

## 3.4 Bewertung

Das Thema der vorliegenden Arbeit lässt sich in die drei oben behandelten Gebiete unterteilen:

1. Paralleles SPH
2. Objektorientiertes wissenschaftliches Rechnen
3. Objektorientierte Design-Pattern

#### Paralleles SPH

Der Überblick aus Abschnitt 3.1 zeigt, dass es eine Reihe von unterschiedlichen parallelen SPH-Codes gibt. Alle Codes machen spezielle Annahmen, entweder bezüglich des SPH-Verfahrens — PTreeSPH mit Eigengravitation — oder bezüglich der Hardware — MEMSY und GRAPE.

Weiterhin sind die SPH-Codes als spezielle Implementierungen, jedoch nicht als echte Bibliothekslösungen zu verwenden. Der Aspekt der Portabilität von parallelen Programmen wird in den besprochenen Implementierungen durch die Verwendung von MPI angegangen. MPI ist ein Standard für nachrichtenorientierte Rechner. Ein mit MPI parallelisiertes Programm kann aber nicht als allgemein portabel angesehen werden, auch wenn es technisch möglich ist, ein MPI-basiertes Programm auf verschiedenen Rechnerplattformen auszuführen. Eine auf Message-Passing-Primitiven beruhende Parallelisierung wird immer maschinenspezifische Annahmen machen müssen, um effizient zu bleiben.

Die Vergleichsmessungen und Effizienzangaben aus dem Abschnitt 3.1 zeigen außerdem ein nicht zufrieden stellendes Ergebnis bei der Ausführung auf massivparallelen Systemen wie der Cray T3E . Ein gutes Skalierungsverhalten auf einem solchen System darf nicht schon unter 100 Knoten einbrechen. Für Programme mit skalierbarer Parallelität von bis zu 32 oder 40 Knoten stehen Maschinen mit gemeinsamem Hauptspeicher, wie die NEC SX-4 , zur Verfügung. Untersuchungen auf diesen

Architekturen liegen nicht vor. Aus eigenen Voruntersuchungen ([29]) muss auch angenommen werden, dass SPH auch auf Systemen wie der Cray T3E mit bis zu 512 Knoten gut skaliert.

#### Paralleles objektorientiertes wissenschaftliches Rechnen

Im Bereich des parallelen objektorientierten wissenschaftlichen Rechnens sind keine der aktuellen Programmpakete und Bibliotheken auf SPH-Verfahren ausgelegt.

Der Überblick in Abschnitt 3.2 zeigt außerdem, dass es keine einheitliche Vorgehensweise bei objektorientierten wissenschaftlichen Programmen gibt. Der Begriff *objektorientiert* bedeutet bei POOMA die Verwendung von C++-Templates in einer speziellen Ausprägung. Cactus verwendet ein vom *Objektorientierten inspiriertes* Modul-Konzept, um die einzelnen Teile in ein einheitliches Konzept zu bringen. Es gibt keinen offensichtlichen Grund, der es verbietet, ein echtes objektorientiertes System zu schaffen. Ein „Eigenbau“, wie das Konzept von Cactus, schafft wieder eine Insellösung eines Systems, bei dem sich der Wissenschaftler auf eine Speziallösung konzentriert. Ein Dialog mit Experten aus anderen Bereichen (Informatik, Mathematik) ist nur dann möglich, wenn diese sich ebenfalls auf die Cactus-Terminologie einlassen.

Bei allen Paketen für objektorientiertes wissenschaftliches Rechnen fehlt eine Auseinandersetzung mit den allgemeinen Erkenntnissen der OO-Technologien wie z.B. den Design-Pattern.

#### Design-Pattern

Objektorientierte Design-Pattern geben die Erfahrungen aus dem Bereich der objektorientierten Programmierung der letzten Jahrzehnte wieder. Dabei werden allgemeine *Meta*-Pattern bis hin zu speziellen Pattern wie den im Anhang dargestellten GoF-Pattern angegeben. Im Bereich des parallelen Rechnens werden grundsätzliche Problemstellungen behandelt. Eine Auseinandersetzung mit speziellen Problemen aus der numerischen, wissenschaftlichen Simulation findet in den Design-Pattern nicht statt.

### 3.4.1 Zu lösende Probleme

- SPH-Verfahren sind eine ganze Klasse von Simulationsverfahren. Je nach Zusammensetzung der Differentialgleichungen für das jeweilige Problem wird ein SPH-Verfahren andere Anforderungen an das Programm stellen. Es muss ge-

zeigt werden, dass ein nicht triviales SPH-Verfahren auf den gängigen Hochleistungsrechnern sinnvoll parallel skaliert.

- Es existieren keine allgemeinen, portablen Lösungsbibliotheken für paralleles SPH, wie sie beispielsweise durch POOMA, Cactus oder Netlib für andere, gitterbasierte Gleichungslöser für partielle Differentialgleichungen zur Verfügung gestellt werden.

- Die parallelen Bibliotheken nehmen alle ein datenparalleles Parallelisierungskonzept an. Die allgemeinere und flexiblere Form der Kontrollflussparallelität wird nicht als vorgefertigte Bibliothekslösung angeboten.

- Im Bereich des objektorientierten wissenschaftlichen Rechnens gibt es keinen Konsens darüber, wie das Design eines objektorientierten Simulationspaketes auszusehen hat. Eine Auseinandersetzung mit den umfangreichen Design-Pattern-Katalogen hat nicht stattgefunden.

- Im Bereich des objektorientierten Programmierens sind durch die Design-Pattern-Kataloge die Erfahrungen der letzten Jahre in eine leicht verständliche und anwendbare Form gebracht. Eine Anwendung dieser Erfahrungen auf wissenschaftliche Programme ist in den Design-Pattern-Katalogen nicht zu finden.

Daraus ergeben sich die bereits in Kapitel 1 beschriebenen Zielsetzungen für die vorliegende Arbeit. Zusammenfassend sind dies:

- Entkopplung von SPH-Verfahren und Parallelisierungsmethoden

- Wiederverwendbare Klassenbibliothek für paralleles SPH

- Anpassung an verschiedene Parallelrechner

- Rahmenprogramm für parallele SPH-Programme

- Anwendung der Methoden: effiziente Parallelisierung von SPH-Verfahren

Im Kapitel 4 werden dazu SPH-Verfahren auf ihre Parallelisierbarkeit für verschiedene Parallelrechnerarchitekturen hin untersucht. Im Kapitel 5 werden Design-Pattern für eine Klassenbibliothek und ein Rahmenprogramm für SPH-Programme dargestellt. Anwendungen und Messungen der in der vorliegenden Arbeit entwickelten Verfahren finden sich im Kapitel 6.

# Kapitel 4

# Parallelisierung von SPH

In diesem und den folgenden Kapiteln werden die eigenen Arbeiten des Autors dargestellt und diskutiert. Die zum Verständnis wichtigen Grundlagen finden sich im Kapitel 2 und Arbeiten anderer Autoren wurden bereits im Kapitel 3 erläutert. Abschnitt 3.4 gibt eine Zusammenfassung und Bewertung der Arbeiten anderer Autoren. Die aus der Bewertung des Stands der Technik folgenden Zielsetzungen wurden bereits in Kapitel 1 motiviert und in Abschnitt 3.4.1 auf Seite 60 zusammengefasst.

Bevor im Kapitel 5 untersucht werden kann, wie Design-Pattern zum Erstellen eines Rahmenprogramms für SPH-Verfahren in Form einer objektorientierten Klassenbibliothek eingesetzt werden können, müssen die Grundlagen der Parallelisierung von SPH-Programmen dargestellt werden.

Das laufende Kapitel beschäftigt sich mit der Struktur von SPH-Programmen. Es werden die parallelisierbaren Teile eines SPH-Programms herausgestellt und Algorithmen angegeben, wie SPH-Verfahren parallelisiert werden können.

Konkrete Umsetzungen und Implementierungen auf die verschiedenen Hardware-Typen finden sich in den Abschnitten 4.4 und 4.5. Dabei wird auf die Implementierung von SPH-Programmen für Maschinen mit gemeinsamem Speicher eingegangen und Implementierungen angegeben, wie SPH auf diesen Architekturen mit einer parallelen Effizienz von bis zu 90% parallelisiert werden kann. Für Architekturen mit verteiltem Speicher werden Implementierungen auf der Cray T3E und dem Kepler-Cluster vorgestellt. Diese Implementierungen können als Referenzimplementierungen für die im Kapitel 6 beschriebenen Anwendungen der in Kapitel 5 beschriebenen objektorientierten Parallelisierungsmethoden verwendet werden.

## 4.1 Vorbemerkungen zur Parallelisierung von SPH

Um eine Parallelisierung eines SPH-Programms durchführen zu können, müssen die verwendeten Algorithmen im Prinzip verstanden werden, damit die Semantik

des SPH-Programms durch die Parallelisierung nicht verändert wird. Allerdings lassen sich Veränderungen von Simulationsergebnissen durch die Parallelisierung nicht ganz vermeiden. Bei den meisten technisch-wissenschaftlichen Simulationen werden sowohl die numerischen Verfahren der Mathematik als auch die Rechnerhardware selbst bis an die Grenze der numerischen Genauigkeit belastet. Schon wenn wenige Codezeilen umgestellt oder wenn nur wenige Operationen vertauscht werden, kann die Simulation im Detail völlig andere Ergebnisse liefern. Würde beispielsweise bei einer kommutativen Operation wie einer Additionsoperation nur die Reihenfolge der Operanden vertauscht, so bliebe im mathematischen Sinne die Semantik der Operation erhalten. Vertauscht man aber in einem Programm die Reihenfolge der Operanden, so sind die Ergebnisse durchaus nicht gleich, da unterschiedliche Rundungsfehler auftreten können. Einzelne Rundungsfehler von Zwischenrechnungen können sich über den gesamten Programmverlauf zudem noch in einer nichtlinearen Form aufschaukeln. Ein System, bei dem kleine Änderungen im Anfangszustand große Auswirkungen auf unterschiedliche Endzustände haben, nennt man auch *chaotisch*.

SPH-Simulationen fallen unter diese Kategorie von Simulationen. Im Fall von SPH-Simulationen bewirken kleine Veränderungen an z.B. den Anfangszuständen der einzelnen SPH-Teilchenkoordinaten große Unterschiede auf den Ort und die Geschwindigkeit — die zwei wesentlichen Größen der klassischen Mechanik — eines SPH-Teilchens am Ende einer Simulation.

Die Parallelisierung eines SPH-Programms ist nun eine sehr drastische Veränderung des Programmcodes und man muss mit Auswirkungen auf die Numerik rechnen. Die Forderung, dass eine Parallelisierung dieselben numerischen Ergebnisse wie eine serielle SPH-Version liefern soll, wäre eine zu starke Einschränkung. Solche Forderungen sind gerechtfertigt bei Produktionscodes im Bereich des Bauingenieurwesens oder der Luft- und Raumfahrttechnik, bei denen langjährige Heuristiken in die Simulationen mit einfließen und direkte Auswirkungen auf z.B. die Gestaltung einer Flugzeugtragfläche hätten.

Ziel dieser Arbeit ist es aber nicht, eine spezielle SPH-Version an einen speziellen Parallelrechner ohne semantische Veränderungen anzupassen, sondern Verfahren zur Parallelisierung von SPH-Verfahren zu untersuchen. Wir nehmen also Veränderungen der Semantik von SPH-Programmen durch die Parallelisierung in Kauf. Wir müssen dann aber die prinzipielle Korrektheit des parallelen SPH-Verfahrens gesondert feststellen.

Für SPH-Verfahren wird die Situation dadurch etwas entschärft, dass die genauen Orts- und Impulskoordinaten der einzelnen SPH-Teilchen nicht von direktem Interesse für den Astrophysiker sind. Die Simulation soll mit Messergebnissen aus

der Astronomie verglichen werden. Diese Messgrößen sind abgeleitete Größen wie Temperatur, Frequenzspektrum des Lichts oder zeitlich veränderliche Helligkeit des Sterns. Zur Berechnung dieser Größen werden immer Mittelwerte über große SPH-Teilchenzahlen gebildet, sodass die einzelne Teilchenposition nicht relevant ist. Um eine Korrektheit einer Parallelisierung für ein SPH-Programm zu zeigen, müssen also diese Mittelwerte übereinstimmen. Eine bis auf die letzte Kommastelle genaue Übereinstimmung kann auch hier nicht erwartet werden, wodurch ein einfacher, maschineller Vergleich der Ergebnisse im Allgemeinen nicht möglich ist. Die Korrektheit der Ergebnisse muss durch einen Experten bestätigt werden. Der Korrektheitsnachweis, d.h. eine Methode, mit der die Ergebnisse von SPH-Programmen verglichen werden können, ist aber nicht Bestandteil dieser Arbeit. An entsprechenden Stellen im Text, wie z.B. bei den im Kapitel 6 vorgestellten Anwendungen, wird deshalb nur kurz auf den Korrektheitsnachweis eingegangen und auf weiterführende Literatur über SPH wie [52] verwiesen.

Wird nun ein SPH-Programm von Beginn an parallel entwickelt, z.B. mit den im Folgenden diskutierten Methoden, dann muss der betreffende Programmierer (Astrophysiker) die Korrektheit sowieso erst gesondert nachweisen, da es sich hier dann i.d.R. um ein neues Simulationsproblem handelt.[1]

## 4.2 Die Programmstruktur von SPH

Die Struktur eines SPH-Programms richtet sich an dem numerischen Verfahren zur Lösung der partiellen Differentialgleichung (siehe Gleichung 2.9) aus. Je nach gewähltem Modell — d.h., je nachdem, welche Terme in der Differentialgleichung vorkommen — wird der Algorithmus etwas anders aussehen. Es lässt sich aber, unabhängig davon, welche Terme die Gleichung — in Gleichung 2.2 mit $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ bezeichnet — tatsächlich hat, eine Grundstruktur für alle SPH-Programme herausarbeiten.

Die Struktur eines SPH-Programms lässt sich in 5 Hauptblöcke aufteilen. Abbildung 4.1 auf Seite 67 stellt die Struktur eines SPH-Programms als UML-Aktivitäts-

[1] Interessant ist zu beobachten, dass besonders bei chaotischen Problemen sehr wohl zwei Programmversionen unterschiedliche Ergebnisse liefern können und trotzdem beide als korrekt angesehen werden — i.d.R. von zwei verschiedenen Forschergruppen. Die geschilderte Problematik tritt in der Diskussion mit den Fachwissenschaftlern (hier: Astrophysiker) nur dann auf, wenn es schon ein bewährtes Simulationsprogramm für das Problem gibt. Dann ist es allerdings nur unter der Mitarbeit des Fachwissenschaftlers möglich, oder sogar unmöglich, der neuen, parallelisierten Version Anerkennung zu verschaffen!

Diagramm[2] dar. Die Strukturelemente bedeuten dabei im Einzelnen:

**A:** Initialisierung

Hier werden die Daten der Simulation eingelesen, die Randbedingungen des physikalischen Systems festgelegt und die Parameter des numerischen Verfahrens initialisiert.

**B:** Zeitintegration mit Fehlerkorrektur

Ein ausgewählter Zeitintegrator bestimmt den genauen Verlauf dieses Programmabschnitts. Das Grundprinzip aller Zeitintegrationsverfahren ist aber das gleiche: eine Iteration über die Terme einer Differentialgleichung von einem Zeitpunkt $T_{Beginn}$ bis zu einem Zeitpunkt $T_{Ende}$.

Zeitintegratoren haben i.d.R. eine dynamische Fehlerkorrektur, um eine Genauigkeitsgrenze für das Ergebnis der Integration angeben zu können (siehe [47]).

**C:** Initialisierung der SPH-Größen

Durch das SPH-Verfahren werden in die Simulation zusätzliche Parameter eingeführt. Diese SPH-Größen müssen gegebenenfalls in jedem Zeitschritt der Zeitintegration neu berechnet werden.

**D:** Lösung der partiellen Differentialgleichung durch SPH

Das SPH-Verfahren berechnet die Terme der der Simulation zugrundeliegenden Differentialgleichung (siehe Gleichung 2.9 auf Seite 17).

**D.1:** Druckterme — Term $\mathcal{B}$ in Gleichung 2.2

**D.2:** Spannungstensor — Term $\mathcal{C}$ in Gleichung 2.2

**D.3:** Volumenkräfte — Term $\mathcal{D}$ in Gleichung 2.2

**D.4:** Beschleunigungen — Term $\mathcal{A}$ in Gleichung 2.2

**E:** Datenausgabe

Ausgaben der Berechnung auf Datenträger. Dieser scheinbar triviale Punkt kann bei den Datenmengen der Simulationen der Astrophysik zu einem schwerwiegenden Problem werden.

Blöcke **A**, **B** und **E** treten im Allgemeinen bei Simulationsverfahren mit einer zeitlichen Entwicklung auf. Die vorliegende Arbeit konzentriert sich an dieser Stelle auf die für eine SPH-Simulation besonderen Blöcke **C** und **D**.

[2] Erläuterungen zu UML-Aktivitäts-Diagrammen siehe Abb. 2.10 auf Seite 37

Die algorithmische Grundstruktur einer SPH-Simulation wird im Algorithmus 1 auf der nächsten Seite weiter ausgeführt. Die unter *SPH-Initialisierung* gestellten Algorithmen *Gittererstellung*, *Nachbarschaftssuche* und *Wechselwirkungslisten* sind vorbereitende Berechnungen für jeden Simulationsschritt eines SPH-Verfahrens. Die Aufgabe der SPH-Initialisierungen ist es, Datenstrukturen für die darauf folgende Be-

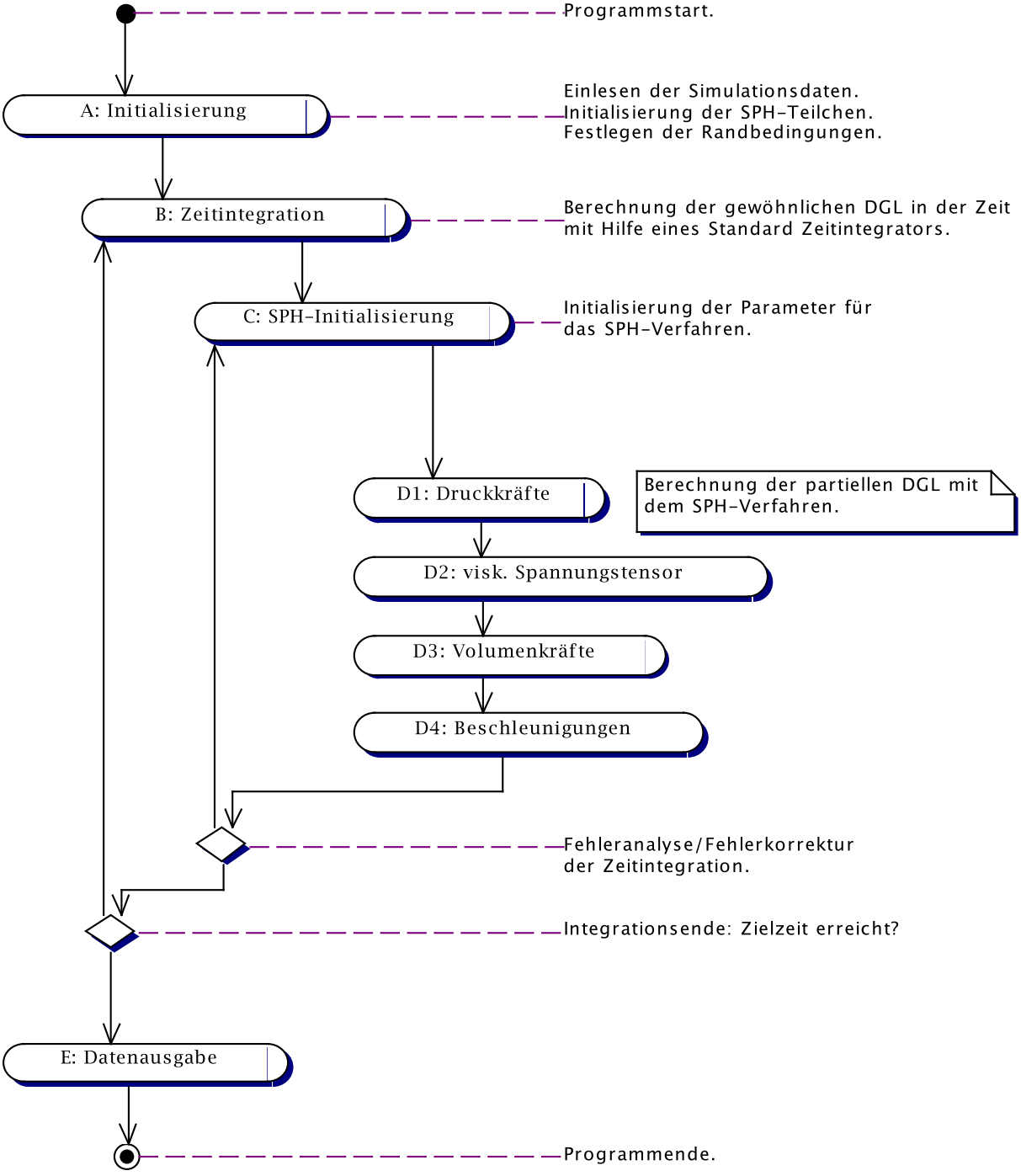



Abbildung 4.1: Die Struktur einer SPH-Simulation.

**Algorithmus 1** *Die Struktur eines SPH-Verfahrens*

**C — SPH-Initialisierung**
Gittererstellung
Nachbarschaftssuche
Wechselwirkungslisten
**D — SPH-Terme berechnen**
Dichteberechnung
Volumenkräfte
Druckkräfte
Spannungstensor

rechnung der SPH-Terme zu erstellen. Es wird zusätzlich zu der vorhandenen Liste von SPH-Teilchen noch eine Liste von wechselwirkenden SPH-Teilchen gebraucht. Da die Gleichung 2.9 auf Seite 17 ein System gekoppelter Differentialgleichungen beschreibt, sind die Berechnungen für die SPH-Teilchen nicht unabhängig voneinander. Diese Abhängigkeit wird in einer *Wechselwirkungsliste* beschrieben. Diese Wechselwirkungsliste wird durch einen *Nachbarschaftssuche*-Algorithmus erstellt, der wiederum ein Hilfsgitter verwendet. Dieses Gitter muss mittels eines *Gittererstellungs*-Algorithmus mit den Teilchendaten initialisiert werden.

Die Algorithmen zur Berechnung der SPH-Terme sind die direkte Implementierung der Gleichung 2.9 auf Seite 17. Die Implementierungsdetails sind für alle SPH-Verfahren ähnlich. In der vorliegenden Arbeit interessieren nur die Auswirkungen der Algorithmen auf die Parallelisierbarkeit von SPH-Programmen. Eine vertiefende Diskussion der in den Kapiteln 2 und 3 dargestellten Implementierungsdetails von SPH-Verfahren findet sich z.B. in [52].

## 4.3 Die parallelisierbaren Teile eines SPH-Programms

Die Blöcke **C** und **D** der Programmstruktur eines SPH-Programms aus dem Abschnitt 4.2 sind die Programmteile, die in jedem Zeitintegrationsschritt abgearbeitet werden. Die Zeitintegrationsscheife selbst kann nicht parallelisiert werden. Die im Algorithmus 1 angegebenen Programmteile sind die Teile eines SPH-Programms, die parallelisiert werden können. Im Folgenden werden dieses Programmteile im Hinblick auf eine Parallelisierung genauer dargestellt. In den darauf folgenden Abschnitten 4.4 und 4.5 werden parallele Algorithmen für die parallelisierbaren SPH-Programmteile für Maschinen mit gemeinsamem Speicher und für Maschinen mit verteiltem Speicher angegeben.

### 4.3.1 Wechselwirkungslisten — Nachbarschaftssuche

**Algorithmus 2** *Nachbarschaftssuche*

Für jedes SPH-Teilchen müssen alle Nachbarn gefunden werden. Nachbarn sind dabei alle Teilchen, die einen kleineren Abstand als den Wechselwirkungsradius $h$ haben:

```
Teilchen[n]                                   {Liste aller SPH-Teilchen}
Gitter[i][j]                                  {2D-Gitter mit Teilchenlisten (siehe Alg. 3)}
for n = 0 to N (N = Teilchenanzahl) do
    (i, j) ← Gitterzelle (i, j) des Teilchen[n] berechnen.
    (ΔI, ΔJ) ← die Umgebung von Zelle (i, j) berechnen.
    for i − ΔI < i < i + ΔI do
        for j − ΔJ < j < j + ΔJ do
            alle Teilchen m aus (ΔI, ΔJ) mit Teilchen n vergleichen.
            if Abstand von Teilchen[m] zu Teilchen[n] < h then
                WW[k] ← Teilchen m und Teilchen n der Wechselwirkungsliste
                zuordnen.
            end if
        end for
    end for                                   {Die Umgebung von Teilchen n ist durchsucht.}
end for                                       {Alle Teilchen n sind durchsucht.}
WW[k]                                         {Liste von Wechselwirkungsteilchen ist erstellt.}
```

Wie im Abschnitt 2.1.2 dargestellt, stehen die SPH-Teilchen miteinander in Wechselwirkung. Für die Lösung eines SPH-Terms sind nur die SPH-Teilchen relevant, die miteinander in Wechselwirkung stehen. Die dafür verantwortliche Größe ist der Wechselwirkungsradius der SPH-Teilchen. Vor der Berechnung eines SPH-Terms müssen alle Wechselwirkungspaare gefunden werden. Algorithmisch ist dies die Suche nach den k-nächsten Nachbarn (*engl.: k-nearest neigbours*) eines SPH-Teilchens.

Die Nachbarschaftssuche zum Erstellen der Wechselwirkungslisten ist eines der aufwendigsten Teile eines SPH-Programms. Ohne Optimierungen wäre dies ein Problem der Komplexität $O(N^2)$, da jedes SPH-Teilchen mit jedem anderen SPH-Teilchen verglichen werden müsste. Die in den Algorithmen 2 und 3 dargestellten Verfahren bieten dabei eine Lösung und reduzieren die Komplexität auf $O(N')$. $N'$ ändert sich in jedem Integrationsschritt und ist stark von dem jeweiligen SPH-Problem abhängig.

Die Lösung besteht in der Einführung einer Hilfsgröße *Gitter* (siehe Algorithmus 3). Mit Hilfe dieses Gitters wird durch eine Vorsortierung der Teilchen die Suche nach Nachbarn eines Teilchens $n$ auf die nähere Umgebung dieses Teilchens $n$ eingeschränkt.

Da die Nachbarschaftssuche für große Teilchenzahlen einen erheblichen Teil der

Programmlaufzeit ausmacht, müssen diese Algorithmen parallelisiert werden. In den Abschnitten 4.4 und 4.5 werden die parallelen Algorithmen zur Nachbarschaftssuche für Maschinen mit gemeinsamem Speicher und für Maschinen mit verteiltem Speicher dargestellt.

**Algorithmus 3** *Gittererstellung (zwei dimensional)*

Das Gitter ist eine Hilfsgröße zur Nachbarschaftssuche (Algorithmus 2) und muss vor jedem Integrationsschritt neu erstellt werden. Jeder Gitterzelle wird eine Teilchenliste zugeordnet (hier im zwei dimensionalen Fall):

```
Gitter[i][j]                                   {2D-Gitter mit (leeren) Teilchenlisten}
Teilchen[n]                                    {Liste aller SPH-Teilchen}
for n = 0 to N (N = Teilchenanzahl) do
    i ← Projektion der x-Koordinate von Teilchen[n] auf das Gitter
    j ← Projektion der y-Koordinate von Teilchen[n] auf das Gitter
    Gitter[i][j] ← Teilchen[n]                 {an die Teilchenliste anhaengen}
end for
```

### 4.3.2 Dichteberechnung — skalare Größen

**Algorithmus 4** *SPH-Massendichte*

Aus der Nachbarliste eines Teilchens kann die SPH-Massendichte an der Stelle des Teilchens $n$ berechnet werden. Die Nachbarn eines Teilchens gehen mit dem *Kernel* (siehe Gleichung 2.4 auf Seite 15) gewichtet in die Massendichte ein.

```
T(n) ≡ Teilchen[n]                             {Liste aller SPH-Teilchen}
WW(i,n) ≡ Nachbar(i,n)                         {Wechselwirkungspartner i von Teilchen n}
ρ(n)                                           {Massendichte an der Stelle von Teilchen n}
for n = 0 to N (N = Teilchenanzahl) do
    ρ(n) = 0.0                                 {Massendichte von Teilchen n initialisieren}
    for all Nachbar i von Teilchen n do
        ρ(n) = ρ(n) + D_N × Kernel(T(n), WW(i,n))
    end for
end for
```

$\mathcal{D}_\mathcal{N}$ ist eine physikalische Normierungsgröße, die aus der Anzahldichte die Massendichte an der Stelle des Teilchens $n$ normiert. In die Berechnung geht sonst nur der über den *Kernel* gewichtete Abstand zwischen dem Teilchen $n$ und seinen Nachbarteilchen ein.

Die Berechnung der SPH-Terme bildet den zweiten Teil der zu parallelisierenden Programmteile eines SPH-Programms. Anhand der Dichteberechnung — in Gleichung 2.9 auf Seite 17 mit $\rho$ bezeichnet — wird die Berechnung eines SPH-Terms in Algorithmus 4 genauer erläutert.

Um eine physikalische Größe wie die Massendichte an der Stelle eines SPH-Teilchens zu berechnen, wird für jedes SPH-Teilchen die Liste seiner Nachbarteilchen abgearbeitet und der SPH-Term berechnet. Es kommt also für jedes Teilchen die in 4.3.1 beschriebene Liste der Wechselwirkungspartner eines Teilchens zum Einsatz.

Jede andere skalare, physikalische Größe wird in einem SPH-Verfahren analog behandelt.

### 4.3.3 Druckkräfte — vektorielle Größen

Druckkräfte sind vektorielle, d.h. gerichtete Größen. Die Berechnung ist analog zu der Berechnung der skalaren Größen, wie sie in Algorithmus 4 dargestellt ist. Auch hier ist die Druckkraft an der Stelle eines Teilchens $n$ eine Funktion über alle Nachbarteilchen gewichtet mit dem *Kernel* aus Gleichung 2.4 auf Seite 15.

Die Druckkraft steht hier als Beispiel für die Berechnung von vektoriellen Größen in einem SPH-Verfahren.

### 4.3.4 Spannungstensor — tensorielle Größen

Tensorielle Größen wie der in Gleichung 2.9 auf Seite 17 dargestellte viskose Spannungstensor stellen eine Besonderheit für die Berechnung dar. Tensorielle Größen erfordern es, auch die Wechselwirkungspartner der Nachbarn eines Teilchens mit in die Berechnung einzubeziehen. Es sind also auch die Nachbarn der Nachbarn gefragt. Abb. 4.2 stellt diesen Sachverhalt graphisch dar.

Durch Einführung einer Zwischengröße in der Gleichung 2.9 ist es möglich, die Berechnung des viskosen Spannungstensors durch ein 2-maliges Durchlaufen der Nachbarlisten für jedes Teilchen zu berechnen. Damit wird aus einem Verfahren der Komplexität $O(N * M)$ ein Verfahren der Komplexität $O(N)$. Dieses mathematische Verfahren wird in [16] vorgestellt. Auswirkungen für die Parallelisierung von SPH-Verfahren werden im Abschnitt 4.4 diskutiert.

### 4.3.5 Volumenkräfte — äußere Kräfte

Volumenkräfte sind auf ein Teilchen von außen einwirkende Kräfte. Diese Kräfte können unabhängig von den Nachbarn eines Teilchens für jedes Teilchen getrennt berechnet werden. Die Parallelisierung solcher Kräfte ist trivial und auf Maschinen mit gemeinsamem Speicher sowie auf Maschinen mit verteiltem Speicher gleichermaßen effizient implementierbar.

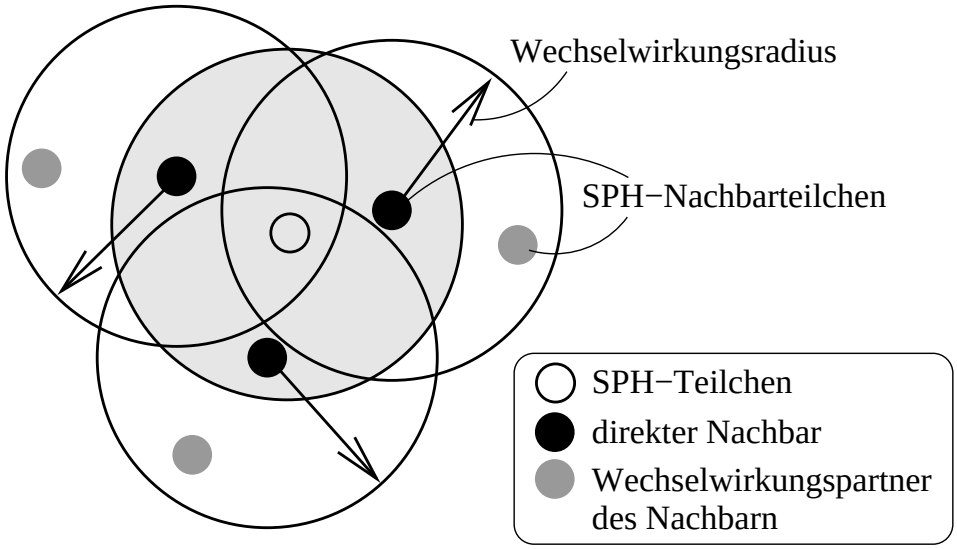



Abbildung 4.2: Zur Berechnung tensorieller Größen an der Stelle eines SPH-Teilchens werden die direkten Nachbarteilchen (schwarz) und deren Wechselwirkungspartner (grau) benötigt. Die Kreise um die SPH-Teilchen stellen den Wechselwirkungsradius eines Teilchens dar.

### 4.3.6 Zusammenfassung

Die in diesem Abschnitt dargestellten Programmteile eines SPH-Verfahrens werden im Folgenden in den Abschnitten 4.4 und 4.5 auf Maschinen mit gemeinsamem und auf Maschinen mit verteiltem Speicher parallelisiert. Die Wechselwirkung zwischen den SPH-Teilchen bringt dabei eine Abhängigkeit zwischen den möglichen parallelen Abläufen ein, die in den parallelen Algorithmen berücksichtigt werden muss. Da sich die Wechselwirkungsbeziehungen zwischen den SPH-Teilchen während des Programmablaufs ständig ändern, stellt diese Abhängigkeit ein nichttriviales Problem für die Parallelisierung dar.

## 4.4 SPH auf Maschinen mit gemeinsamem Speicher

In den Abschnitten 4.3.1 bis 4.3.5 wurden die parallelisierbaren Teile eines SPH-Programms dargestellt. Im Folgenden wird eine konkrete Implementierung eines parallelen SPH-Programms untersucht. Dieser Abschnitt bezieht sich auf Implementierungen für Maschinen mit gemeinsamem Speicher. Implementierungen für Architekturen mit verteiltem Speicher werden im Abschnitt 4.5 untersucht.

Zuerst werden die Besonderheiten einer Implementierung auf Maschinen mit

gemeinsamem Speicher erläutert. Dann wird eine Implementierung eines SPH-Programms auf der NEC SX-4 auf Laufzeitverhalten hin näher untersucht. Dabei wird die parallele Effizienz und der Speedup der Implementierung diskutiert. Die Darstellung von SPH auf Maschinen mit gemeinsamem Speicher schließt mit einem Vergleich von parallelem SPH mit seriellen und vektorisierten SPH-Versionen im Abschnitt 4.4.3.

### 4.4.1 Algorithmische Besonderheiten

Vor der Erklärung der zur Parallelisierung auf Maschinen mit gemeinsamem Speicher notwendigen algorithmischen Anpassungen muss kurz auf den Zusammenhang zwischen mathematischem Modell eines SPH-Verfahrens und dem seriellen SPH-Programm eingegangen werden.

Wie im Abschnitt 4.3.4 auf Seite 71 dargestellt wurde, sind zur Berechnung von tensoriellen Größen in SPH-Verfahren zusätzlich zu der Information über die Nachbarn eines Teilchens auch noch die Informationen über die Wechselwirkungspartner der Nachbarn notwendig.

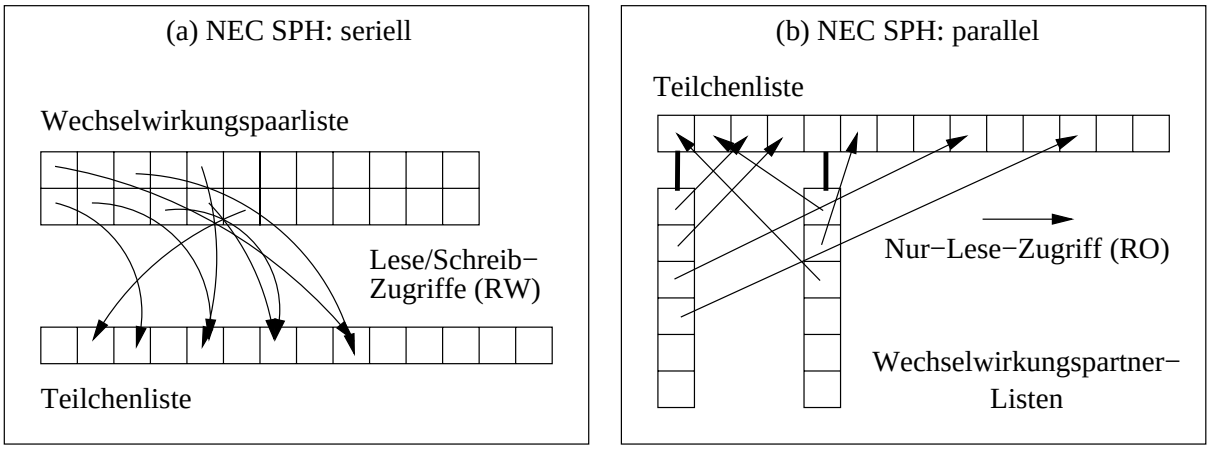



Abbildung 4.3: (a) zeigt die Datenstrukturen in einem seriellen Programm. In (b) sind die zur parallelen Ausführung geeigneten Datenstrukturen dargestellt.

Um die numerische Berechnung zu vereinfachen, wurden im mathematischen Modell (siehe Gleichung 2.9 auf Seite 17) physikalische Eigenschaften berücksichtigt — für eine fachliche Diskussion dieses Vorgehens verweisen wir auf [16] und [52]. Eine dieser physikalischen Eigenschaften ist die Symmetrie der auf die Teilchen wirkenden Kräfte (*Actio = Reactio*). Dies bedeutet, dass Kräfte sowohl auf das Teilchen wie auch auf dessen Nachbarn (Wechselwirkungspartner) wirken. Die Kräfte auf bei-

de Teilchen sind gleich groß, aber wirken in die entgegengesetzte Richtung. Mathematisch bedeutet das, dass dabei nur ein Vorzeichen $(+/-)$ beachtet werden muss. In einem Programm muss eine Kraft auf beide beteiligten Teilchen also nur einmal berechnet werden und kann dann mit unterschiedlichem Vorzeichen auf beide Teilchen addiert werden.

Die serielle Implementierung dieses numerischen Verfahrens hat dann eine Liste von Wechselwirkungspaaren, wie sie durch den Algorithmus 2 berechnet werden. Diese Liste von Wechselwirkungspaaren wird zur zentralen Datenstruktur bei der Berechnung aller SPH-Terme wie in Abbildung 4.3 (a) dargestellt. Ausgehend von der Liste der Wechselwirkungspaare wird auf die Liste der SPH-Teilchen *schreibend* zugegriffen. Der vom mathematischen Modell vorgegebene Algorithmus sieht vor, dass für alle SPH-Terme höchstens ein zweimaliges Durchlaufen der Liste von Wechselwirkungspaaren (im Falle von tensoriellen Größen) zu deren Berechnung ausreicht. Im Algorithmus 5 ist dies schematisch dargestellt.

**Algorithmus 5** *Wechselwirkungspaare (seriell)*

$\mathcal{WWPL}$ sei die durch Algorithmus 2 berechnete Liste von wechselwirkenden Teilchen. In der Liste sind jeweils Paare von Teilchen (bzw. Teilchennummern) gespeichert, sodass $\mathcal{WWPL} \rightarrow T_1$ und $\mathcal{WWPL} \rightarrow T_2$ Teilchen aus der Liste der Teilchen $\mathcal{T}$ bezeichnen.

**for all** $\mathcal{WW}_i \in \mathcal{WWPL}$ **do**

{*fuer alle Wechselwirkungspaare in der Liste*}

$\mathcal{T}_1 \equiv \mathcal{T}(\mathcal{WW}_i \rightarrow T_1)$ {*Wechselwirkungspartner 1 lesen*}

$\mathcal{T}_2 \equiv \mathcal{T}(\mathcal{WW}_i \rightarrow T_2)$ {*Wechselwirkungspartner 2 lesen*}

$\mathcal{K} = \mathcal{K}(\mathcal{T}_1, \mathcal{T}_2)$ {*Berechne Kraefte auf Teilchen* $\mathcal{T}_1$ *und* $\mathcal{T}_2$ }

$\mathcal{T}_1 \leftarrow \mathcal{K}$ {*Schreibe Ergebnis fuer Teilchen 1*}

$\mathcal{T}_2 \leftarrow -\mathcal{K}$ {*Schreibe Ergebnis fuer Teilchen 2*}

**end for**

Alle Kräftetypen (siehe Abschnitte 4.3.2 bis 4.3.5) — hier mit $\mathcal{K}(\mathcal{T}_1, \mathcal{T}_2)$ bezeichnet — , können durch ein zweimaliges Durchlaufen der Liste der Wechselwirkungspaare $\mathcal{WWPL}$ berechnet werden.

Was auf den ersten Blick wie getrennte Überlegungen aussieht, nämlich das mathematische Modell und die serielle Implementierung, stellt sich bei näherem Hinsehen als ein durchgängiger Gedankengang heraus. Bei den Überlegungen zum mathematische Modell wurde schon an die Optimierung im Falle einer Implementierung als Software gedacht. Die Darstellung des SPH-Verfahrens mit einer Liste von Wechselwirkungspaaren als zentrale Informationsquelle ist für eine serielle (prozedurale) Implementierung hilfreich und führt bei der Umsetzung zu einem optimierten Programm.

Bei einer Implementierung auf einem Parallelrechner hingegen führt die Überle-

gung, eine Liste von Wechselwirkungspaaren zu verwenden, — in Algorithmus 5 mit $\mathcal{WWPL}$ bezeichnet — zu einem zentralen, kritischen Abschnitt. Da bei dem Durchlaufen der Liste der Wechselwirkungspaare bei jedem Paar schreibend auf die Liste der Teilchen zugegriffen werden muss und die Teilchen in verschiedenen Wechselwirkungspaaren auftreten, stellt der Schreibzugriff auf die Teilchenliste einen kritischen Abschnitt dar und muss durch wechselseitigen Ausschluss geschützt werden.

Durch diesen kritischen Abschnitt wird zum einen der parallele Programmablauf unterbrochen, da alle Zugriffe auf die Teilchenliste sequentialisiert werden. Zum anderen muss, um einen wechselseitigen Ausschluss zu garantieren, ein Semaphor (siehe 3) verwendet werden. Durch die Systemaufrufe für die Semaphorvariable wird die Programmlaufzeit beträchtlich erhöht, da diese Aufrufe für jeden Teilchenzugriff in allen Zeititerationen vorkommen. Bei 100.000 Teilchen und 20.000 Iterationsschritten sind dies schon $2 * 10^9$ zusätzliche Systemaufrufe für Semaphore.

Ein Ansatz zur Lösung wäre es, die Liste der Wechselwirkungspaare so zu sortieren, dass die Liste in Teillisten unterteilt werden könnte, die unabhängig voneinander stehen; d.h., dass in den verschiedenen Teillisten wechselseitig nicht auf Teilchen der anderen Teillisten zugegriffen wird. Diese Teillisten könnten dann nebenläufig berechnet werden. Der zur Sortierung der Liste der Wechselwirkungspaare notwendige Algorithmus erhöht wiederum die Laufzeit des SPH-Programms. Da die SPH-Teilchen in den meisten Problemen stark miteinander wechselwirken — d.h., jedes SPH-Teilchen hat viele Nachbarn — und außerdem die Wechselwirkungspartner sich ständig ändern, würde es nur wenige unabhängige Teillisten geben. D.h., die mit einem solchen Ansatz gewonnene Parallelität wäre für ein chaotisches Problem, wie dies bei den hier untersuchten SPH-Verfahren vorliegt, zu gering; das Programm würde nicht sinnvoll auf Hochleistungsrechnern skalieren.

In Abbildung 4.3 auf Seite 73 ist die hier gefundene Lösung des Problems schematisch dargestellt. Die Lösung setzt bei den Überlegungen im mathematischen Modell an. Wie oben beschrieben, wird im Modell und in der seriellen Implementierung die physikalische Eigenschaft von *Actio = Reactio* ausgenützt, um die numerische Berechnung zu beschleunigen. In der parallelen Implementierung des SPH-Verfahrens wird diese Eigenschaft ignoriert. Die Konsequenz ist, dass Kräfte zwischen zwei Teilchen für jedes Teilchen gesondert berechnet werden müssen.

Auch diese Programmänderung hat eine Laufzeiterhöhung zur Folge, die aber nach oben abgeschätzt werden kann. Durch die doppelte Berechnung aller Kräfte auf die Teilchen wird sich die Programmlaufzeit gegenüber einem optimalen seriellen Programm höchstens verdoppeln. Da außer den Kräfteberechnungen auch noch andere Programmteile zeitlich relevant sind, ist der Faktor 2 sogar eine obere Grenze der Laufzeiterhöhung für diese Art der Parallelisierung.

**Algorithmus 6** *Wechselwirkungspartner*

In der Liste der Teilchen $\mathcal{T}$ ist jedem Teilchen $\mathcal{T}_i$ eine Liste von Wechselwirkungspartnern $\mathcal{WWPARTNER}$ zugeordnet. Diese Listen werden durch den Algorithmus 2 berechnet.

**for all** $\mathcal{T}_i \in \mathcal{T}$ **do**
{*fuer alle Teilchen in der Teilchenliste*}
  **for all** $\mathcal{WW}_j \in \mathcal{WWPARTNER}$ **do**
{*fuer alle Wechselwirkungspartner dieses Teilchens*}
    $\mathcal{K} = \mathcal{K}(\mathcal{T}_i, \mathcal{WW}_j)$ {*Berechne Kraefte auf* $\mathcal{T}_i$}
    $\mathcal{T}_i \leftarrow \mathcal{K}$ {*Schreibe Ergebnis fuer Teilchen* $\mathcal{T}_i$}
  **end for**
**end for**

Im Algorithmus 6 ist die veränderte algorithmische Struktur dargestellt. Hier ist die Teilchenliste die einzige und zentrale Datenstruktur, auf die lesend und schreibend zugegriffen wird. Durch die jedem Teilchen eigens zugeordnete Liste von Wechselwirkungspartnern ist jedes Teilchen für die Berechnung seiner neuen Koordinaten selbst verantwortlich. Die Informationen der Nachbarteilchen aus der Liste der Wechselwirkungspartner dient als Nur-Lese-Information und stellt somit *keinen* kritischen Abschnitt für den parallelen Programmablauf dar.

Durch die Neuformulierung der Problemstellung im mathematischen Modell und die Umstrukturierung der Daten bei der Implementierung wurden die kritischen Abschnitte bei der Kräfteberechung der SPH-Teilchen vermieden.

Die Schleifendurchläufe der Schleife über alle Teilchen $\mathcal{T}_i$ im Algorithmus 6 können jetzt nebenläufig ausgeführt werden. Da in den Simulationen große Teilchenzahlen verwendet werden, ist dieses Verfahren effizient für die Ausführung auf Hochleistungsrechnern, da es gute Skalierungseigenschaften besitzt.

### 4.4.2 Implementierung auf NEC SX-4

Der für die Implementierung des parallelen SPH-Verfahrens verwendete Rechner ist die 1996 im Höchstleistungsrechenzentrum (HLRS) in Stuttgart installierte NEC SX-4. Die Leistungsdaten der NEC SX-4 sind in Abschnitt 2.2.1 auf Seite 24 zusammengefasst.

#### Parallelisierung nach dem Threads-Paradigma

Die in Abschnitt 4.3 auf Seite 68 dargestellten parallelisierbaren Teile eines SPH-Programms werden auf der NEC SX-4 nach dem Threads-Paradigma implementiert.

Als Schnittstelle zum System der NEC SX-4 wird dabei die am WSI[3] entwickelte Parallelisierungsumgebung DTS [6] verwendet.

Jedes der in Abschnitt 4.3 erläuterten Teile einer SPH-Kräfteberechnung wird in einem prozeduralen Programm als eine Funktion (Subroutine) abgearbeitet. Das Threads-Paradigma zur Parallelisierung setzt genau auf dieser Ebene eines Programms an. Funktionsaufrufe zur Berechnung einer SPH-Kraft, z.B. der Druckkraft — Term $\mathcal{B}$ in Gleichung 2.2 —, werden in einem eigenem Thread nebenläufig abgearbeitet.

Jedem Funktionsaufruf wird dabei ein Teil der Teilchenliste zur Bearbeitung übergeben. Da die Kräfte auf die Teilchen in der Teilchenliste unabhängig voneinander berechnet werden können, können je nach Größe der Teil-Teilchenlisten mehr oder weniger Threads zur Kräfteberechnung eingesetzt werden.

Nach einer Kraftberechnung muss der Programmablauf synchronisiert werden. Diese Synchronisation geschieht beim Threads-Paradigma implizit durch das Zusammenführen der Threads mittels `join`.

```
1   /* die Länge der Teillisten bestimmen */
2   length = num_of_particles / num_of_threads;
3
4   /* die Teilchenlisten (particleList) auf die Threads
5      verteilen */
6   for(threadNo = 0; threadNo < num_of_threads; threadNo++)
7     dts_id[threadNo] =
8         fork_Pressure(particleList(num_of_particles),
9                       parameters);
10
11  /* die berechneten Ergebnisse synchronisieren/einsammeln */
12  for(threadNo = 0;  threadNo < num_of_threads; threadNo++)
13    join_Pressure(dts_id[threadNo]);
```

Abbildung 4.4: Parallelisierung von SPH mit dem Thread-Paradigma.

In Abbildung 4.4 ist ein vereinfachter Programmtext angegeben, der die Teilchen in einer Teilchenliste `particleList` auf Threads verteilt. Die Funktion `Pressure` arbeitet nach dem im Algorithmus 6 dargestellten Prinzip und berechnet für jedes Teilchen aus der übergebenen Teil-Teilchenliste mit Hilfe der Wechselwirkungspartner die Kraft auf das Teilchen.

Da die Teilchen über die Liste der Wechselwirkungspartner auf Daten anderer Teilchen lesend zugreifen müssen, ist diese Art der Parallelisierung nur auf Rechnern mit gemeinsamem Hauptspeicher (*engl.: shared memory*) wie der NEC SX-4 möglich.

[3] Wilhelm-Schickard-Institut der Universität Tübingen

Hier wird auch davon ausgegangen, dass dieser Hauptspeicherzugriff für alle Teilchen gleich teuer ist (UMA).

**Effizienz und Speedup der Implementierung**

| # Threads | # CPU | Laufzeit in sec. | Speedup | Effizienz |
|---|---|---|---|---|
| 1 | 1 | 4020 | `Referenz` | `Referenz` |
| 2 | 2 | 2018 | 1,99 | 0,99 |
| 4 | 4 | 1013 | 3,96 | 0,99 |
| 5 | 5 | 812 | 4,95 | 0,99 |
| 8 | 8 | 510 | 7,88 | 0,98 |
| 10 | 10 | 406 | 9,90 | 0,99 |
| 20 | 20 | 210 | 19,14 | 0,95 |

Tabelle 4.1: Speedup und Effizienz von SPH auf NEC SX-4 mit einer SPH-Simulation mit 100.000 Teilchen.

Zur Messung der parallelen Effizienz der SPH-Implementierung auf der NEC SX-4 wurden die Profiling-Informationen von Programmläufen auf der NEC SX-4 ausgewertet.

In Abbildung 4.5 sind die wesentlichen Teile der Profiling-Informationen eines Programmlaufs auf der NEC SX-4 abgebildet. Dabei interessieren vor allem die nebenläufigen Prozesse und deren Laufzeiten. In Abbildung 4.5 sind diese Zeiten mit `Conc. Time` bezeichnet. Die Anzahl der nebenläufigen Prozesse ist mit `Max Concurrent Proc.` angegeben.

In Tabelle 4.1 sind Programmläufe mit SPH-Programmen zusammengefasst. Das ausgemessene SPH-Programm berechnete die Bewegung einer zerfließenden Akkretionsscheibe um eine Zentralmasse. Die Messdaten wurden für Thread-Zahlen bis 20 Threads erfasst. Dabei konnten aus programmtechnischen Gründen nicht alle 20 Messstufen ausgemessen werden.

In der 3. Spalte der Tabelle 4.1 ist die Gesamtprogrammlaufzeit in Sekunden angegeben. Durch Vergleich mit der Gesamtprogrammlaufzeit des Programms mit einem Thread auf einer CPU können die parallele Beschleunigung (*engl.: speedup*) und die parallele Effizienz für die Programmläufe mit mehreren Threads auf mehreren CPUs berechnet werden. Die Ergebnisse sind dabei auf zwei Nachkommastellen gerundet.

Der Laufzeitverlust gegenüber einer optimalen seriellen Programmversion blieb bei diesen Messungen unberücksichtigt und wird in Abschnitt 4.4.3 auf Seite 81 diskutiert.

In der Abbildung 4.6 sind die Messungen als Säulendiagramm dargestellt. Auf der Abszisse ist die Zahl der verwendeten CPUs (Threads) und auf der Ordinate ist die

```
     ******  Program Information  ******
Real Time (sec)       :          1070.654886
User Time (sec)       :          4092.188967
Sys  Time (sec)       :             2.346253
Inst. Count           :           3250542085
FLOP Count            :             58901462
MOPS                  :             0.794328
MFLOPS                :             0.014394
MOPS   (concurrent)   :             3.137786
MFLOPS (concurrent)   :             0.056858
Memory Size (MB)      :          1584.000000
Max Concurrent Proc.  :                   5
 Conc. Time(>= 1)(sec):          1035.935046
 Conc. Time(>= 2)(sec):          1007.439705
 Conc. Time(>= 3)(sec):          1007.204944
 Conc. Time(>= 4)(sec):          1003.518409
 Conc. Time(>= 5)(sec):             4.958729
MIPS                  :             0.794328
MIPS (concurrent)     :             3.137786
I-Cache (sec)         :             1.322210
O-Cache (sec)         :            97.181292
Bank (sec)            :             4.327705
```

Abbildung 4.5: Profiling Information auf NEC SX-4.

parallele Effizienz aufgetragen.

Grau dargestellt sind die Simulationsläufe mit 10.000 SPH-Teilchen. Bei diesen Messungen ist erkennbar, dass die parallele Effizienz mit zunehmender Thread-Anzahl abnimmt. D.h., mit zunehmender Zerstückelung der Teilchenliste von 10.000 Teilchen in Teillisten bekommt der serielle Overhead mehr Gewicht und die parallele Laufzeit wird größer.

Die schwarz dargestellten Simulationsläufe wurden mit 100.000 SPH-Teilchen gemacht. Dort ist der Effizienzverlust bei 20 CPU deutlich geringer als bei den Simulationsläufen mit 10.000 Teilchen.

Mit 100.000 Teilchen kann für 20 CPU eine parallele Effizienz von 90% erreicht werden, bei 10.000 Teilchen immer noch parallele Effizienzen von bis zu 69%. In Abbildung 4.7 sind die Ergebnisse in Form eines Speedup-Graphen aufgezeigt. An dieser Darstellung kann der zu erreichende Laufzeitgewinn direkt abgelesen werden. Die

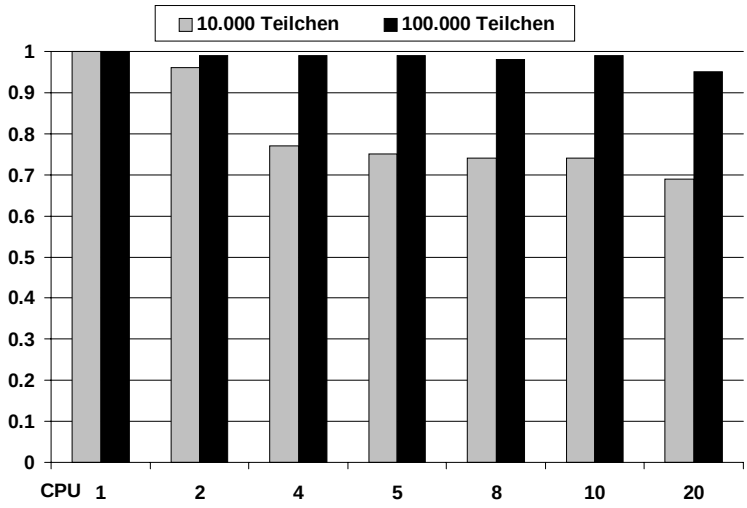



Abbildung 4.6: Effizienz der Parallelisierung von SPH auf NEC SX-4.

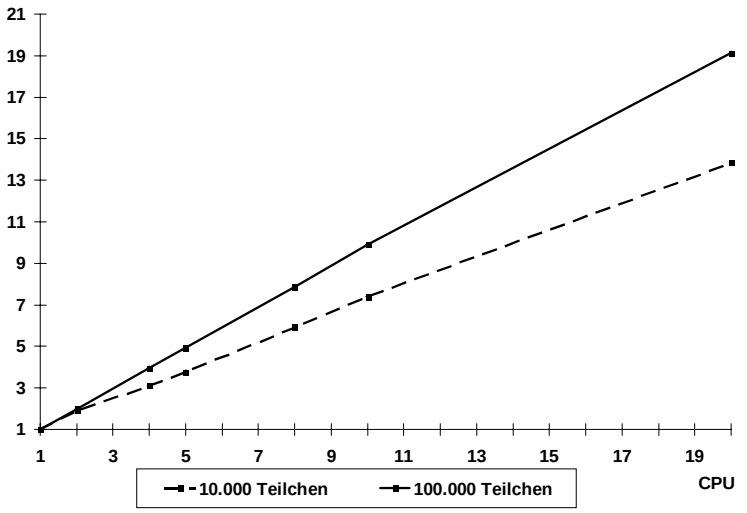



Abbildung 4.7: Speedup der Parallelisierung von SPH auf NEC SX-4.

gestrichelte Linie zeigt die Simulationsläufe mit 10.000 SPH-Teilchen, die durchgezogene Linie zeigt die Simulationsläufe mit 100.000 SPH-Teilchen. Für 20 CPU erreicht man mit 100.000 Teilchen einen 19fachen Laufzeitgewinn.[4]

Das Ergebnis zeigt deutlich, dass das hier gefundene Verfahren zur Parallelisierung von SPH-Simulationen gut skaliert und eine fast optimale parallele Effizienz erreicht.

[4]Die NEC SX-4 läuft in einem Batch-Modus und wird von mehreren Forschergruppen aus verschiedenen Universitäten und Industrieunternehmen verwendet. Leider standen dadurch für einen Programmlauf nur höchstens 20 Knoten zur Verfügung.

### 4.4.3 Vergleich mit einem vektorisierten SPH-Programm

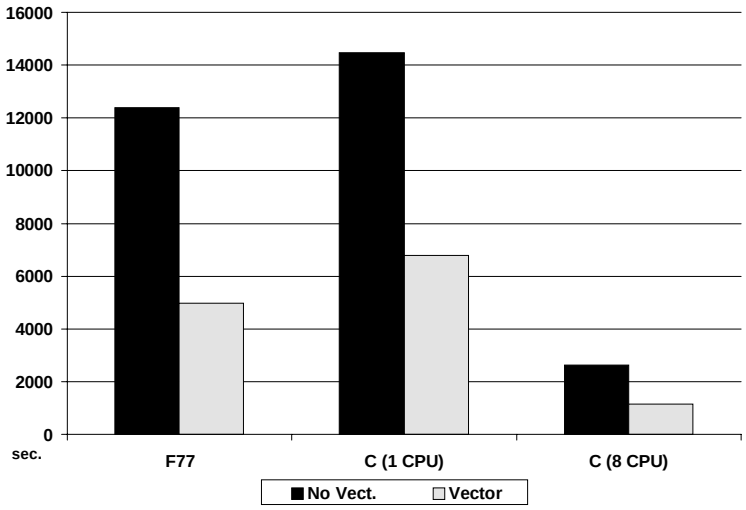



Abbildung 4.8: Untersuchungen zur Vektorisierung eines parallelen SPH-Programms auf der NEC SX-4.

Abschließend muss noch diskutiert werden, wie das hier parallelisierte SPH-Programm im Vergleich mit einem optimierten seriellen SPH-Programm zu bewerten ist.

Dazu wurden Vergleichsmessungen auf der NEC SX-4 mit einem bereits vorhandenen SPH-Programm in `FORTRAN77` gemacht. In Abbildung 4.8 sind die Ergebnisse dargestellt. Die schwarzen Säulen geben die Programmlaufzeiten in Sekunden für eine `FORTRAN77`-Implementierung im Vergleich mit einer parallelisierten Implementierung in `C` mit einer CPU und mit 8 CPU an.

Es ist zu erkennen, dass die parallelisierte Version auf einer CPU langsamer ist als das serielle Programm in `FORTRAN77`. Der in Abschnitt 4.4.1 angebene Faktor 2 als obere Grenze für die Laufzeiterhöhung wird dabei nicht erreicht. Die Schwankungen in der Laufzeiterhöhung sind abhängig von der Teilchenverteilung jedes Simulationslaufes.

Schon die parallelisierte Version mit 8 CPU ist aber einen Faktor 6 schneller als die serielle Version in `FORTRAN77`.

Die grau dargestellten Säulen zeigen die gleichen Messungen mit auf der NEC SX-4 vektorisierten Versionen. Auf die Vektorisierung soll hier nicht eingegangen werden. Auch die vektorisierten Versionen zeigen ein ähnliches Verhalten; auch hier ist eine parallelisierte Version schon bei 8 CPU deutlich schneller als ein serielles, vektorisiertes `FORTRAN77`-Programm. Hier wird natürlich noch ein weiterer Laufzeitgewinn

erzielt, indem die Vektorregister der NEC SX-4 ausgenutzt werden.

## 4.5 SPH auf Maschinen mit verteiltem Speicher

Anhand der Untersuchungen in Abschnitt 4.4 konnte gezeigt werden, dass sich ein SPH-Verfahren effizient parallelisieren lässt. Auf Maschinen mit gemeinsamem Hauptspeicher wie der NEC SX-4 werden dabei parallele Effizienzen von bis zu 90% erreicht.

Maschinen mit gemeinsamem Speicher skalieren aber nur für relativ wenig Prozessoren; die NEC SX-4 hat in voller Ausbaustufe 32 CPU. Auch das Nachfolgemodell, die NEC SX-5, hat mit 32 CPU ihre skalierbare Leistungsgrenze der Hardware erreicht.

Wie in Abschnitt 4.4.2 dargestellt, skaliert das in der vorliegenden Arbeit beschriebene parallele SPH-Verfahren bis 20 CPU mit nahezu optimaler Effizienz. Es ist also anzunehmen, dass auch Maschinen mit wesentlich mehr Knoten (CPU) noch effizient ausgelastet werden können. Maschinen mit mehr als 100 CPU sind ausnahmslos Maschinen mit verteiltem Hauptspeicher, wie z.B. die Cray T3E mit 512 Knoten.

Eine Maschine mit verteiltem Hauptspeicher verlangt allerdings ein anderes Programmiermodell als das in Abschnitt 4.4 verwendete Thread-Paradigma. Üblicherweise wird auf Maschinen mit verteiltem Hauptspeicher mit einem Message-Passing-Modell gearbeitet. Die Cray T3E bietet mit der SHMEM-Bibliothek auch noch ein Programmiermodell mit globalem Adressraum an und fällt damit in die Klasse der *Distributed-Shared-Memory*-Architekturen (siehe 2.2.1 auf Seite 21).

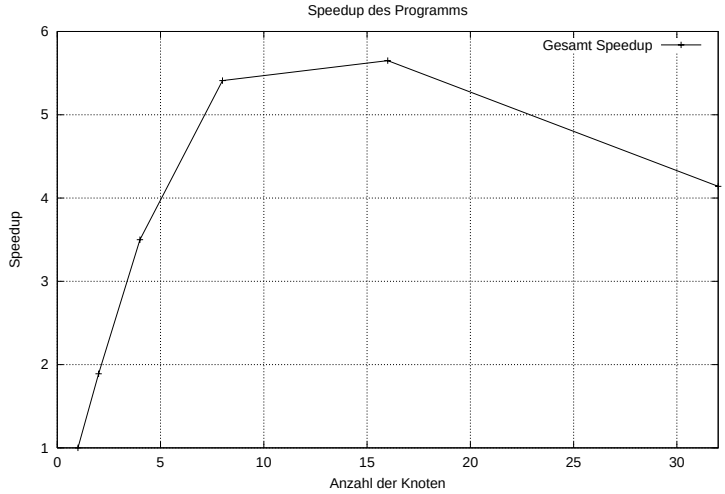
Unabhängig von der gewählten Hardware und dem jeweiligen Programmiermodell ist die generelle Parallelisierbarkeit des SPH-Verfahrens. Auch die in Abschnitt 4.4.1 ausgearbeiteten parallelisierbaren Teile eines SPH-Verfahrens sind für ein Message-Passing-Modell die gleichen. Es kann gezeigt werden, dass die parallelen Programmierparadigmen des Threads-Paradigmas sowie des Message-Passing-Modells äquivalent sind [56]. D.h., die Bedeutung eines parallelen Programms wird durch den Tausch des parallelen Programmiermodells nicht verändert.

Die technische Realisierung einer Implementierung mit Message-Passing auf einer Maschine mit verteiltem Hauptspeicher stellt allerdings eine starke Veränderung gegenüber einer Implementierung auf Maschinen mit gemeinsamem Speicher mittels des Thread-Paradigmas dar.

### 4.5.1 Besonderheiten für verteilten Speicher

Bevor auf die spezielle Problematik der Programmierung auf Rechnern mit verteiltem Speicher eingegangen wird, soll zunächst eine direkte Portierung des in Ab-

**(a)**




**(b)**

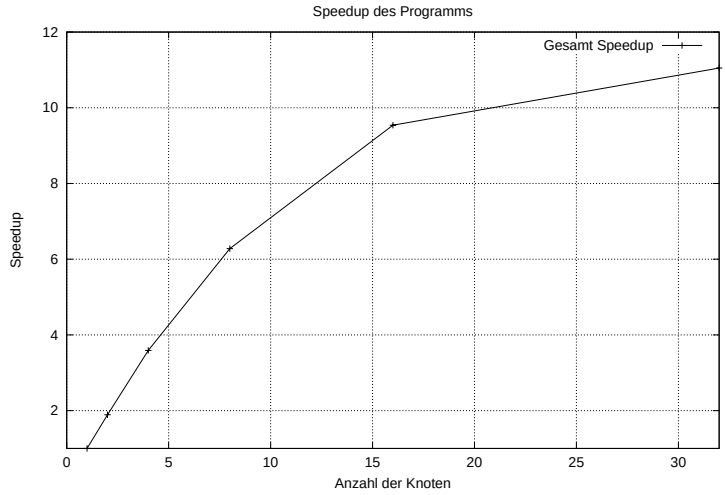



Abbildung 4.9: Speedup der direkten Portierung auf die Cray T3E des SPH-Programms von der NEC SX-4 **(a)** und Speedup der 1. Verfeinerung **(b)** (Programmlauf mit 5000 Teilchen).

schnitt 4.4.2 beschriebenen SPH-Programms auf die Cray T3E gezeigt werden. Da die Cray T3E außer mit MPI auch mit der Cray-eigenen SHMEM-Bibliothek, die ein Programmiermodell mit globalem Adressraum zur Verfügung stellt, programmiert werden kann, ist eine direkte Portierung der Implementierung von der NEC SX-4 auf die Cray T3E möglich.

Die zu parallelisierenden Programmteile bleiben auch bei der Implementierung auf der Cray T3E so wie in Abschnitt 4.2 beschrieben. Die Simulationsdaten müssen bei der Cray-Implementierung aber mittels der SHMEM-Bibliothek auf die beteiligten Knoten verteilt werden. Diese Verteilung geschieht mittels *PUT*- und *GET*-Operationen mit den Simulationsdaten als Übergabeparameter. Die *PUT*- und

*GET*-Operationen dienen außerdem noch als Synchronisationspunkt der parallelen Abläufe. Die NEC-Implementierung mittels des Thread-Paradigmas kann so leicht auf die Cray T3E portiert werden.

In Abbildung 4.9 **(a)** ist der Speedup-Graph von Programmläufen mit 5000 SPH-Teilchen aufgezeigt. Es ist zu erkennen, dass bei 16 Knoten ein Speedup von nur 5,5 erreicht wird. Bei höherer Knotenanzahl bricht die Effizienz weiter ein.

Dieses Ergebnis liegt deutlich unter der erreichten Effizienz der Implementierung auf der NEC SX-4. Eine naive, direkte Übernahme von Programmcode einer Implementierung für Maschinen mit gemeinsamem Speicher auf eine *Distributed-Shared-Memory*-Maschine führt erwartungsgemäß zu einem unbefriedigenden Ergebnis.

In Abbildung 4.9**(b)** ist der Speedup-Graph einer ersten Verfeinerung der Portierung auf die Cray T3E aufgezeigt. Hier ist zu sehen, dass die parallele Effizienz bis 32 Knoten nicht einbricht. Bei 16 Knoten ist immerhin noch eine parallele Effizienz von ca. 60% erreicht.

Die Veränderung besteht hier darin, dass die *PUT*- und *GET*-Operationen durch *COLLECT*- und *BROADCAST*-Operationen teilweise ersetzt wurden. Schon leichte Optimierungen können also das SPH-Programm an die neue Architektur besser anpassen. Es steht zu erwarten, dass eine spezielle Implementierung für die Cray T3E, die die besonderen Eigenheiten der Hardware mit einbezieht, wesentlich bessere Ergebnisse auch für hohe Knotenzahlen bringen wird.

In den folgenden Abschnitten wird auf die Details einer speziellen Implementierung für Maschinen der Bauart der Cray T3E eingegangen. In Abschnitt 4.5.3 wird eine Implementierung einer SPH-Simulation auf der Cray T3E untersucht. Für die Ausführung der Implementierungen und Messungen wurde von Michael Hipp eine Diplomarbeit angefertigt [28].

### 4.5.2 Gebietszerlegung für SPH-Verfahren

Das Programmiermodell mit globalem Adressraum der Cray, das durch die SHMEM-Bibliothek gegeben ist, unterscheidet sich von dem Programmiermodell einer Maschine mit gemeinsamem Hauptspeicher, wie der NEC SX-4, dadurch, dass der Aufwand für einen Speicherzugriff nicht für alle Speicheradressen gleich groß ist. Eine solche Art des Speicherzugriffs wird auch als NUMA (*engl.: Non-Uniform-Memory-Access*) bezeichnet (siehe [50]).

Den unterschiedlichen Zugriffszeiten auf lokalen und entfernten Speicher bei der Cray T3E muss bei der Parallelisierung Rechnung getragen werden. Operationen auf Daten, die im lokalen Speicher liegen, sind Operationen auf Daten auf entferntem Speicher vorzuziehen. Um während eines Programmablaufs möglichst viele lokale

Operationen vollziehen zu können, müssen die Daten geschickt auf die Knoten des Rechners verteilt werden.

Die gleiche Problematik tritt bei Maschinen mit verteiltem Speicher auf. Wird die Cray T3E mittels eines nachrichtenorientierten Programmiermodells, wie es beispielsweise durch MPI zur Verfügung gestellt wird, programmiert, so kann die Cray T3E als eine Maschine mit verteiltem Speicher angesehen werden. Bei diesem Programmiermodell ist ein entfernter Speicherzugriff nicht möglich und die Simulationsdaten müssen per Nachrichtenaustausch auf die Knoten des Rechners verteilt werden.

Eine solche Aufteilung der Simulationsdaten auf die verschiedenen Knoten eines Rechners nennt man Gebietszerlegung (*engl.: domain-decomposition*).

Gebietszerlegungsverfahren richten sich an dem jeweiligen Simulationsverfahren aus. Dabei ist zur effizienten Implementierung auch die konkrete Topologie des verwendeten Parallelrechners zu berücksichtigen. Im Folgenden soll das hier für SPH-Verfahren entwickelte und implementierte Gebietszerlegungsverfahren vorgestellt werden. Dieses Gebietszerlegungsverfahren hat sich aus den Vorarbeiten (siehe [29]) und den Analysen bestehender Verfahren als sinnvoll erwiesen. Im Anschluss an die Darstellung des Gebietszerlegungsverfahrens selbst folgt die Beschreibung des sich daraus ergebenden Programmablaufs für SPH-Simulationen. Im Abschnitt 4.5.3 wird die parallele Effizienz des Verfahrens quantitativ untersucht.

#### Die Zwei-Gebiete-Trennung

Um bei den SPH-Verfahren mit Gebietszerlegung Rechenzeit und Speicherverbrauch besser bestimmen zu können, wird hier im Gegensatz zu der im Abschnitt 4.4.2 gezeigten Implementierung ein SPH-Verfahren mit variablem Wechselwirkungsradius verwendet. Der Unterschied zwischen den SPH-Verfahren mit festem und mit variablem Wechselwirkungsradius besteht darin, dass bei dem Verfahren mit festem Wechselwirkungsradius die Größe, die den Wechselwirkungsbereich eines Teilchens angibt, festgehalten wird und für alle Teilchen gleich ist. Damit hat jedes SPH-Teilchen eine veränderliche und unterschiedliche Anzahl von Wechselwirkungspartnern.

Bei dem Verfahren mit variablem Wechselwirkungsradius wird die Anzahl der Wechselwirkungspartner für jedes Teilchen gleich gehalten und jedem SPH-Teilchen ein anderer und veränderlicher Wechselwirkungsradius zugeteilt. Dieser variable Wechselwirkungsradius muss nach jedem Zeitintegrationsschritt neu berechnet werden und richtet sich nach der zu erreichenden Anzahl von Wechselwirkungspartnern pro Teilchen.

Die physikalischen und mathematischen Implikationen des Verfahrens mit variablem Wechselwirkungsradius finden sich in [52].

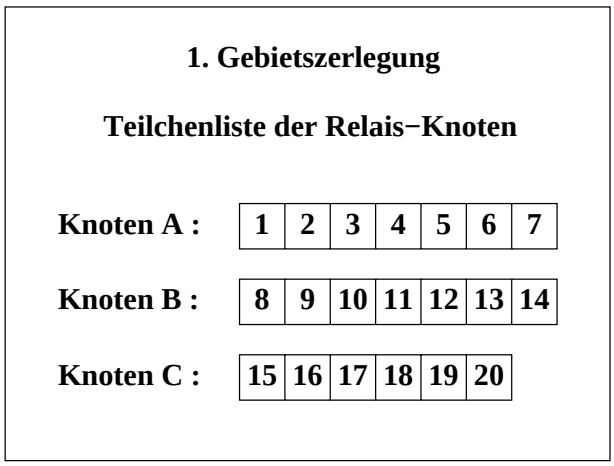



Abbildung 4.10: Die Teilchen werden ohne Berücksichtigung von Eigenschaften gleichmäßig auf ihre *Relais*-Knoten verteilt. Hier im Beispiel mit 20 Teilchen auf die 3 Knoten A, B und C.

Die SPH-Teilchen werden nun wie folgt auf die Knoten des Rechners verteilt. In der 1. Gebietszerlegung (siehe Abbildung 4.10) werden die Teilchen gleichmäßig auf die vorhandenen Knoten des Rechners verteilt. Dabei werden keine Randbedingungen berücksichtigt, d.h., die Teilchen werden einfach anhand ihrer (zufälligen) Teilchennummer auf die Knoten verteilt. Da wir hier mit variablem Wechselwirkungsradius und einer festen Anzahl von Wechselwirkungspartnern arbeiten, ist die Last dadurch gleichmäßig auf die Knoten verteilt. Jeder Knoten dient für seine so zugewiesenen SPH-Teilchen als *Relais*-Knoten, d.h., die Daten für ein bestimmtes SPH-Teilchen können über diesen allen bekannten Relais-Knoten für dieses Teilchen bezogen werden.

Die 2. Gebietszerlegung wird mit Hilfe einer Gitterstruktur ermittelt. Dabei wird die Geometrie des Simulationsraumes der SPH-Teilchen mit einbezogen. Die Gitterstruktur orientiert sich an den Wechselwirkungsbeziehungen der Teilchen untereinander und nützt dadurch die simulationseigene Topologie mit aus. Die Funktion dieser Gitterstruktur ist ähnlich der in Abschnitt 4.2 beschriebenen Gitterstruktur zur Nachbarschaftssuche. Über dieses Gitter werden Abhängigkeiten zwischen den SPH-Teilchen aufgelöst. Jede Gitterzelle wird im Folgenden einem Knoten zugewiesen. Die Wechselwirkungsbeziehungen zwischen den SPH-Teilchen, die in verschiedenen Gitterzellen zu liegen kommen, werden über *Import*- und *Export*-Listen für die einzelnen Knoten dargestellt. In Abbildung 4.11 ist die Verteilung der Gitterzel-

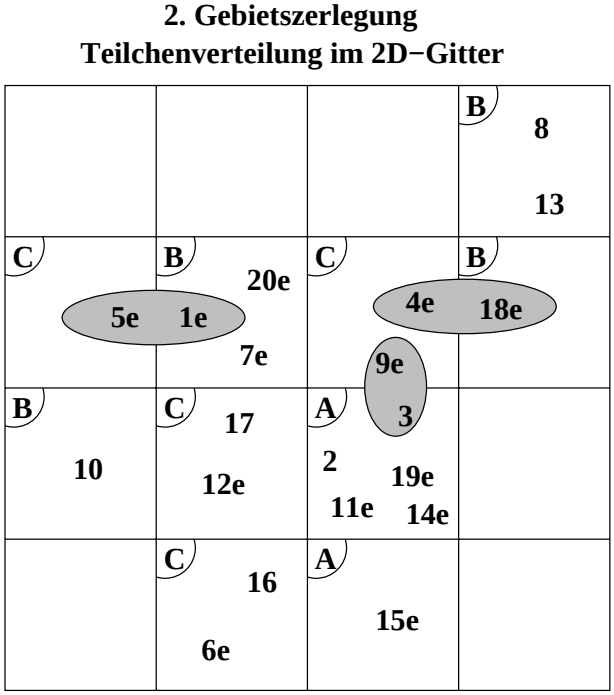



Abbildung 4.11: Die 2. Gebietszerlegung berücksichtigt die Topologie des Simulationsraumes. Hier die Teilchenverteilung auf die Knoten für den zweidimensionalen Fall. Die Knotennummer ist mit A, B oder C angegeben. Die mit **e** markierten Teilchen kommen in die Export-Liste des Knotens. Die grau hinterlegten Teilchen zeigen die Wechselwirkungsbeziehungen, die für die Import-Liste verantwortlich sind.

len mit den darin enthaltenen SPH-Teilchen der 2. Gebietszerlegung auf die Knoten (`A,B,C`) gezeigt.

Die Import- und Export-Listen sind dabei wie folgt aufgebaut (siehe Abbildung 4.12):

- ***Export*-Listen** In den *Export*-Listen werden die Teilchen verwaltet, die durch die 2. Gebietszerlegung diesem Knoten zugewiesen wurden und deshalb auf diesem Knoten berechnet werden, deren *Relais*-Knoten (1. Gebietszerlegung) aber ein anderer Knoten ist.

  **Beispiel** Dem Knoten `A` wurden durch die 1. Gebietszerlegung die Teilchen mit den Nummern `1,2,3,4,5,6,7` zugewiesen (Abb. 4.10). Für diese Teilchen ist der Knoten `A` der *Relais*-Knoten. Durch die 2. Gebietszerlegung werdem dem Knoten `A` außerdem noch die Teilchen `2,3,11,14,15,19` zugewiesen

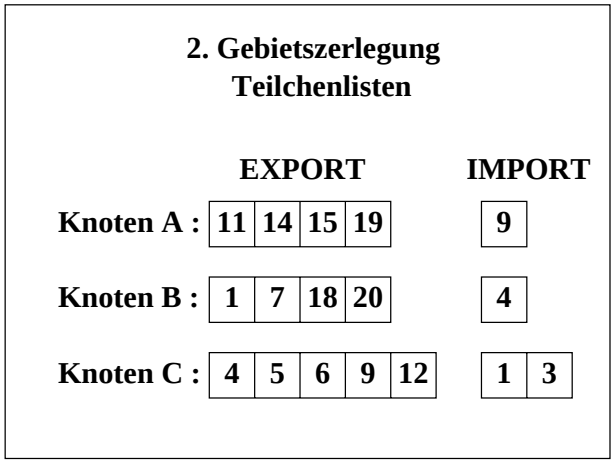



Abbildung 4.12: Aus der 2. Gebietszerlegung werden die Import- und Export-Listen der Knoten gebildet. Siehe dazu auch Abb. 4.11 und Abb. 4.10.

(Abb. 4.11). Damit sind die Teilchen `11,14,15,19` in der *Export*-Liste des Knotens `A` (Abb. 4.12).

- ***Import*-Listen** In den *Import*-Listen werden die Teilchen gehalten, für die der Knoten weder der *Relais*-Knoten ist und die auch nicht durch die 2. Gebietszerlegung dem Knoten zugewiesen wurden, die aber durch die Wechselwirkung von Teilchen des Knotens mit anderen Teilchen von anderen Knoten zur Berechnung benötigt werden.

  **Beispiel** Die *Import*-Liste des Knotens `A` besteht aus dem Teilchen `9`. Das Teilchen 9 steht mit dem Teilchen `3` in Wechselwirkung, ist aber weder durch die 1. Gebietszerlegung noch durch die 2. Gebietszerlegung dem Knoten `A` zugewiesen worden. Der Knoten `A` wird alle Teilchen aus seiner Import-Liste von deren jeweiligen Relais-Knoten anfordern. In diesem Fall wird das Teilchen 9 von dessen Relais-Knoten `B` angefordert.

### Programmablauf einer verteilten SPH-Simulation

In Abbildung 4.13 auf der nächsten Seite ist der Programmablauf einer SPH-Simulation mit dem in Abschnitt 4.5.2 beschriebenen Gebietszerlegungsverfahren als UML-Aktivitätsdiagramm dargestellt. Nach einer Initialisierungsroutine durchläuft das SPH-Programm die Simulationsschritte, bis die gewünschte numerische Genauigkeit erreicht ist. Die einzelnen Programmteile werden im Folgenden näher erklärt.

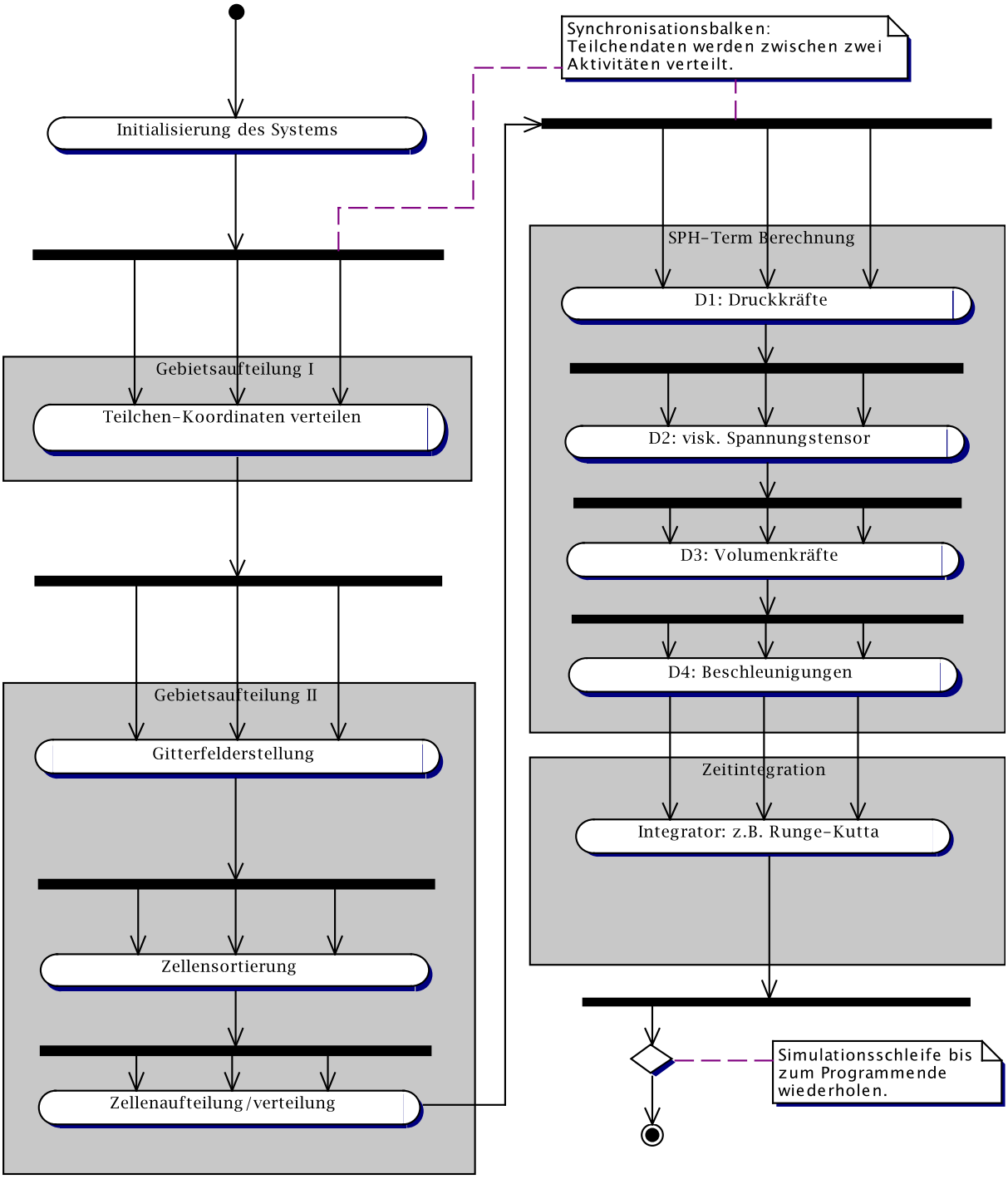



Abbildung 4.13: Ablaufschema des SPH-Programms auf der Cray T3E mit Kommunikations- und Synchronisationsstellen. (Vergleiche auch Abb. 4.1 auf Seite 67.)

**I: 1. Gebietszerlegung** Die Teilchen-Koordinaten werden gleichmäßig ohne Berücksichtigung von Randbedingungen auf die vorhandenen Knoten verteilt. Dies ist der erste Teil der unter dem Abschnitt 4.5.2 auf Seite 85 beschriebenen Zwei-Gebiete-Trennung. Die Teilchenverteilung auf die Knoten der Rechners ist als Beispiel in Abb. 4.10 auf Seite 86 dargestellt. Dadurch wird eine einfache Lastverteilung erreicht.

**II: 2. Gebietszerlegung** Die Realisierung der 2. Gebietszerlegung umfasst die in der Abbildung 4.13 dargestellten Schritte der *Gitterfelderstellung*, *Zellsortierung* und *Zellaufteilung*, auf die an dieser Stelle detaillierter eingegangen wird. Wichtig ist dabei, dass bis auf die Zellaufteilung alle Schritte vollständig parallelisiert werden konnten. Eine Darstellung der parallelen Effizienz der Programmteile der 2. Gebietszerlegung findet sich im Abschnitt 4.5.3 auf Seite 92.

**Gitterfelderstellung (Zellerstellung)** In der Gitterfelderstellung werden die Teilchen des Simulationsraumes anhand ihrer Ortskoordinaten in Zellen einsortiert. Das Einsortieren in die Zellen folgt dabei im Wesentlichen einem Linked-List-Algorithmus, wie er z.B. in [29] und in [2] beschrieben ist.

Dieser Algorithmus kann hier parallel ausgeführt werden, indem jeder Knoten seine eigenen Teilchen aus der 1. Gebietszerlegung bearbeitet. Teilchen, die logisch in eine lokale Zelle des Knotens gehören, die aber auf einem anderen Relais-Knoten liegen, werden über eine Referenz gekennzeichnet. Über diesen Algorithmus lässt sich auch gleichzeitig die *Export*-Liste für die Knoten erstellen.

Im Anhang B auf Seite 175 ist der Programmcode der Gittererstellung für die Implementierung auf der Cray T3E abgebildet.

**Zellsortierung / Zellaufteilung** Im Anschluss an die logische Verteilung der Teilchen erfolgt eine physikalische Verteilung der Zellen auf die vorhandenen Knoten. Im ersten Schritt werden die erstellten Zellen der Größe nach sortiert. Größe bedeutet hier die Anzahl der einer Zelle zugeordneten Teilchen.

Diese Sortierung kann durch einen halb parallelen Mergesort gemacht werden. Dabei reduziert sich die Anzahl der beteiligten parallelen Knoten mit fortschreitendem Algorithmus (deshalb: halb parallel). Eine Parallelisierung ist aber auch hier notwendig, da ein einzelner Knoten mit der Aufgabe überlastet wäre.

Die dann folgende Zellaufteilung wird von einem Masterknoten übernommen. Hier wird ein schneller Algorithmus gefordert, da diese Serialisierung keine Verzögerung nach sich ziehen darf.

Der halb parallele Mergesort ist im Anhang B auf Seite 176 abgebildet.

**Nachbarschaftssuche** Im Rahmen der Zellaufteilung kann auch die Nachbarschaftssuche und damit die Erstellung der Wechselwirkungpartner-Listen für die SPH-Teilchen abgearbeitet werden. Ohne die Zellaufteilung hätte dieser Algorithmus einen Aufwand der Ordnung $O(N^2)$ für $N$ Teilchen, da jedes SPH-Teilchen mit jedem anderen SPH-Teilchen auf eine mögliche Wechselwirkungsbeziehung hin untersucht werden müsste. Durch die vorherige Zellaufteilung kann die Suche auf die Zellen beschränkt werden, die innerhalb des Wechselwirkungsradius $h$ eines SPH-Teilchens liegen. Die Suche ist ein spiralähnliches Durchlaufen der eigenen und der umliegenden Zellen. Die Implementierung dieses Algorithmus ist im Anhang B auf Seite 173 dargestellt. Um eine bessere Lastverteilung zu erreichen, wird ein Knoten, der mit der Nachbarschaftssuche auf seinen Teilchen fertig ist, einem noch in Arbeit befindlichen Knoten helfen, indem er die noch nicht bearbeiteten Zellen dieses Knotens übernimmt.

| Teilchenzahl | Gitterdimension |
|---|---|
| 10.000 | $64 \times 64$ |
| 100.000 | $128 \times 128$ |
| 500.000 | $256 \times 256$ |

Tabelle 4.2: Erfahrungswerte der Gittergröße für verschiedene Teilchenzahlen (aus [28]).

Da die Zellen des Gitters zur Abschätzung der Wechselwirkungsbeziehungen verwendet werden, hängt die Effizienz dieser Nachbarschaftssuche stark von der Gittererstellung ab. Ein zu klein gewähltes Gitter erfordert zu viele Zugriffe auf Nachbarzellen. Bei einem zu groß gewählten Gitter ist die Nachbarschaftssuche zu ineffizient (im Extremfall eine $O(N^2)$-Suche).

Erfahrungswerte für die Gittergröße sind in Tabelle 4.2 aufgezeigt. Dabei sind die Teilchenzahlen gelistet, die in den im Abschnitt 4.5.3 diskutierten Messungen auf der Cray T3E verwendet wurden. Abweichungen von diesen Gittergrößen führen zu Laufzeitverlusten für den Programmteil mit der Nachbarschaftssuche. Für andere Teilchenzahlen und verschiedene Wechselwirkungsradien $h$ muss jeweils eine geeignete Gittergröße durch Ausmessen gefunden werden.

**III: SPH-Verfahren** Jetzt durchläuft das verteilte SPH-Programm die schon in Abschnitt 4.3 dargestellten Teile der eigentlichen, numerischen Berechnung der Daten nach dem SPH-Verfahren. Das Verfahren schließt einen Berechnungsschritt mit der Zeitintegration (z.B. Runge-Kutta-Zeitintegrator).

Die Synchronisationsbalken zwischen den Berechnungsschritten der einzelnen SPH-Terme in Abbildung 4.13 geben die Zeitpunkte im Programmablauf an, an denen die Teilchendaten der Teilchen aus den *Import*- und *Export*-Listen über die Knoten synchronisiert werden. Die Synchronisation erfolgt über *Barrier*-Aufrufe.

### 4.5.3 Implementierung auf Cray T3E

Für die Implementierung des SPH-Verfahrens mit Zwei-Gebiete-Trennung stand die im HLRS in Stuttgart installierte Cray T3E zur Verfügung. Für eine Zusammenfassung der Leistungsdaten der Cray T3E siehe Abschnitt 2.2.1 auf Seite 21.

#### Effizienz und Speedup von SPH auf der Cray T3E

Zur Messung der Programmlaufzeiten auf der Cray T3E wurde der Cray-eigene Profiler *Apprentice* verwendet. Der Profiler erlaubt eine übersichtliche Anzeige der Laufzeiten aller Programmfunktionen. In der graphischen Ansicht sind dabei mit einem grünen Balken die Zeiten, bei denen die CPU rechnet, gekennzeichnet; mit einem gelben Balken sind Wartezeiten auf I/O-Operationen gekennzeichnet; mit einem roten Balken sind Wartezeiten für Zugriffe auf entfernte Datenbereiche (Synchronisation) gekennzeichnet.

Aus den Profiling-Informationen des Cray-Profilers lassen sich nun die Laufzeiten der Implementierungen der SPH-Verfahren direkt ablesen, woraus sich Speedup und parallele Effizienz berechnen lassen. In den Abbildungen 4.14 bis 4.16 sind die Ergebnisse der Messungen graphisch als Speedup und Effizienzdiagramm dargestellt.

#### Diskussion der Messungen

In den Abbildungen 4.14 bis 4.16 ist im unteren Teil die parallele Effizienz bei 10.000, 100.000 und 500.000 SPH-Teilchen dargestellt. Die Anzahl der Knoten variiert dabei zwischen einer Mindestanzahl von Knoten, auf der das Problem noch berechnet werden kann, bis zur maximalen Knotenanzahl auf der Cray T3E von 512 Knoten. Die minimale Knotenanzahl richtet sich dabei nach der Problemgröße: bei 10.000 Teilchen kann das Problem noch auf einem Knoten berechnet werden. Bei 100.000 Teilchen braucht man schon mindestens 4 Knoten und bei 500.000 Teilchen lässt sich das

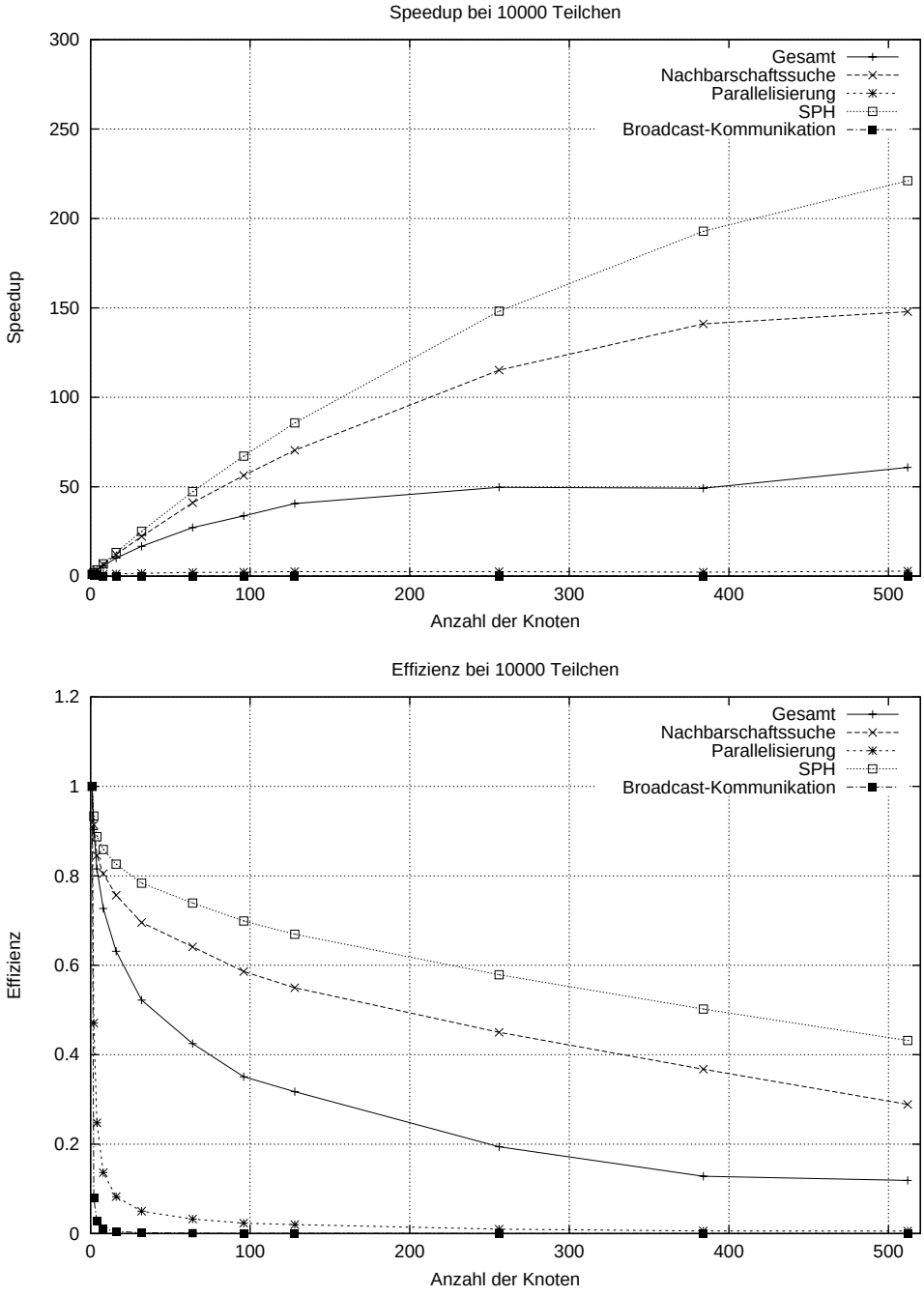



Abbildung 4.14: Messung des 10.000-Teilchenproblems.
Gittergröße: $64 \times 64$.

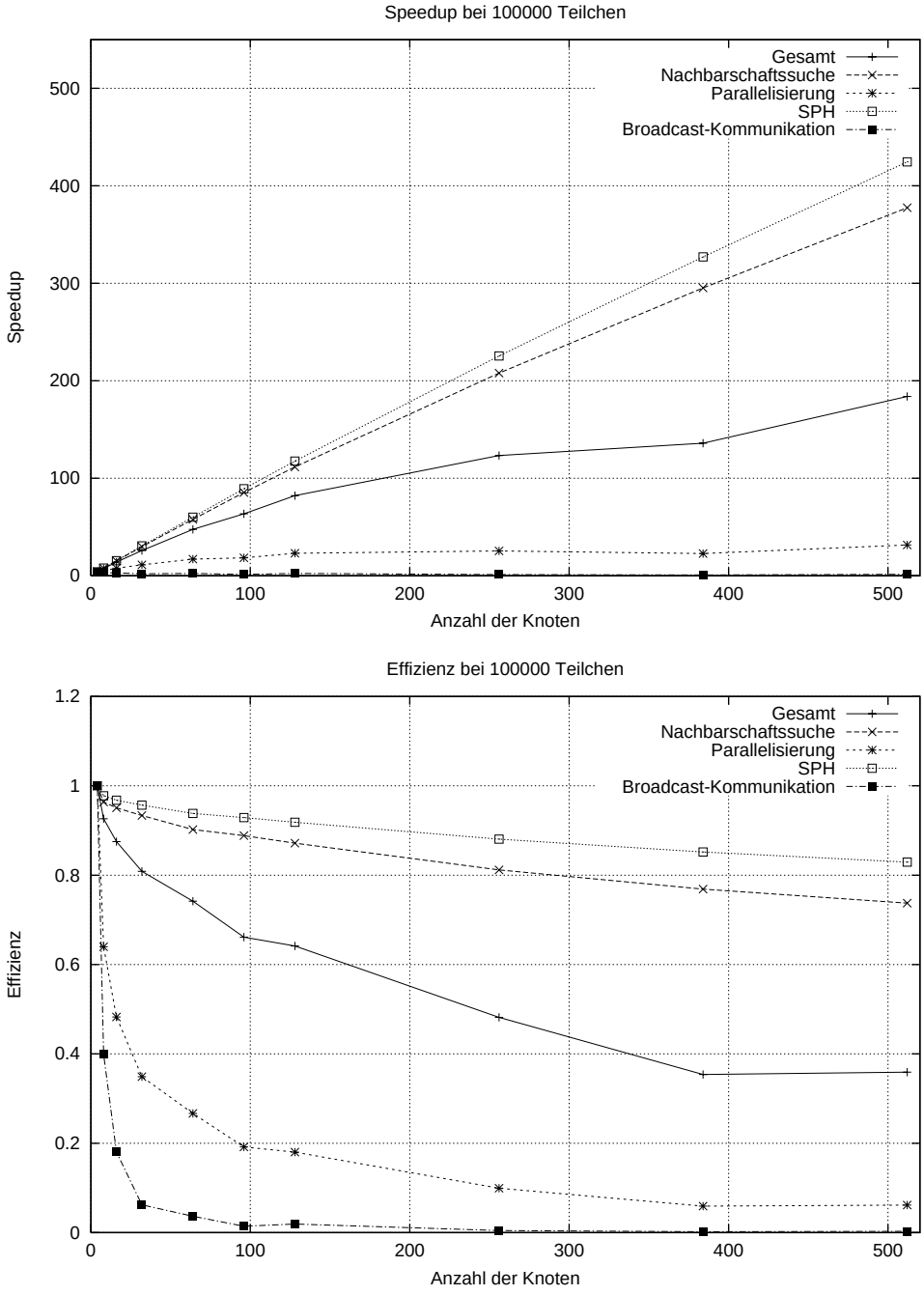



Abbildung 4.15: Messung des 100.000-Teilchenproblems.
Gittergröße: $128 \times 128$.

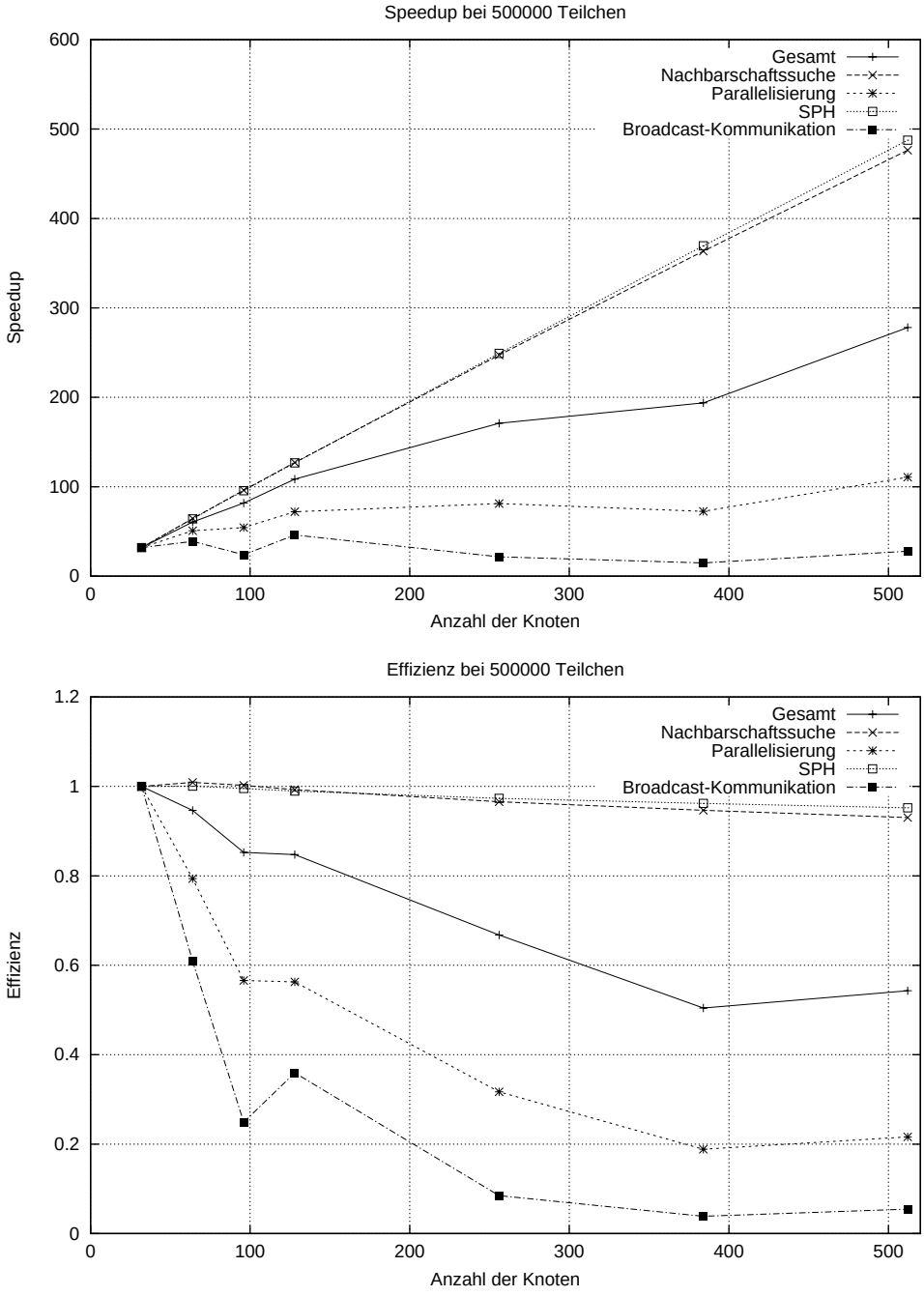



Abbildung 4.16: Messung des 500.000-Teilchenproblems.
Gittergröße: $256 \times 256$.

Problem erst ab 32 Knoten berechnen. Aus diesen Zahlen ist zu sehen, dass der Speicherverbrauch pro Teilchen nicht konstant ist. Dies liegt an der Liste der Wechselwirkungspartner, die für jedes Teilchen mit verwaltet wird. Bei größeren Teilchenzahlen wurden bei den in Abb. 4.14 bis 4.16 dargestellten Messungen auch mehr Wechselwirkungspartner pro Teilchen verwendet. Somit können bei 100.000 Teilchen noch 25.000 Teilchen pro Knoten gehalten werden, bei 500.000 Teilchen aber nur noch ca. 16.000 Teilchen.

Die Messungen der parallelen Laufzeit des SPH-Programms ist in den Abbildungen 4.14 bis 4.16 in jeweils fünf Graphen dargestellt. Der Graph der Gesamtmessung gibt den tatsächlichen zeitlichen Verlauf eines SPH-Programms auf der Cray T3E wieder. Die anderen vier Graphen stellen die wichtigsten Teile eines SPH-Programms gesondert dar (siehe dazu Abb. 4.13). Diese Teile sind wie folgt in den Diagrammen bezeichnet:

1. *Nachbarschaftssuche*: die Laufzeit des parallelisierten Nachbarschaftssuche-Algorithmus. Die Bestimmung der Wechselwirkungspartner ist nach jedem Integrationsschritt notwendig und stellt einen der Hauptteile eines parallelen SPH-Programms dar (siehe Abschnitt. 4.5.2 Nr. 2 auf Seite 91).

2. *SPH*: die Laufzeit der eigentlichen SPH-Berechnungen. Die parallele Berechnung der SPH-Terme ist in Abb. 4.13 mit *SPH-Term Berechnung* gekennzeichnet. In den Diagrammen der Messungen ist zu erkennen, dass die SPH-Terme von allen Teilen eines SPH-Programms am besten skalieren.

3. *Parallelisierung*: die Kommunikations- und Wartezeiten der parallelen Bearbeitungsschritte. Darunter fallen auch die Programmteile der 2. Gebietszerlegung. Die Programmlaufzeiten der nur für die Parallelisierung eingebrachten Programmteile sind in diesem Graph gesondert aufgeführt. In dieser Messung ist auch der Broadcast-Kommunikationsanteil enthalten.

4. *Broadcast-Kommunikation*: Gesondert dargestellt ist noch die Broadcast-Kommunikation, die auch schon in der Messung *Parallelisierung* enthalten ist.

Aus den Diagrammen ist erkennbar, dass sich die Effizienz stark mit der Problemgröße verändert. Bei 10.000 Teilchen sinkt die Effizienz bereits bei 64 Knoten unter 50%. Bei 100.000 Teilchen wird die 50%-Marke erst bei 256 Knoten unterschritten und bei 500.000 Teilchen bleibt die parallele Effizienz bis zur maximalen Knotenanzahl von 512 Knoten deutlich über 50%.

Das Verhältnis von Parallelisierungs- und Kommunikationsanteil zur eigentlichen Rechenzeit ist bei großen Teilchen deutlich besser. Dies liegt mitunter daran, dass der Aufwand einiger Parallelanteile unabhängig von der Teilchenanzahl ist.

Die Nichtmonotonien in den Graphen der Messungen, besonders deutlich bei 500.000 Teilchen zu erkennen, müssen noch erklärt werden. Diese Nichtmonotonien treten bei den Parallelisierungs- und Kommunikationsteilen auf und übertragen sich damit auch auf die Gesamtmessung. Die Nichtmonotonien treten bei Knotenzahlen auf, die keine Zweierpotenz sind, und sind durch die gewählten parallelen Algorithmen zu erklären.

Im oberen Teil der Abbildungen 4.14 bis 4.16 sind die Messungen nochmals als Speedup-Diagramm dargestellt, an denen die zu erreichende Beschleunigung abgelesen werden kann. Für 500.000 Teilchen ergibt sich eine Beschleunigung von 278 bei 512 Knoten.

Die Cray T3E sollte also aus Effizienzgründen nur für Simulationen mit sehr großen Teilchenzahlen eingesetzt werden.

## 4.6 Zwischenergebnis

Es konnte gezeigt werden, dass die SPH-Methode grundsätzlich gut zu parallelisieren ist, sofern nicht in der mathematischen Formulierung des physikalischen Problems schon Annahmen über einen seriellen Programmablauf einbezogen werden (siehe Abschnitt 4.4).

Auf Maschinen mit gemeinsamem Hauptspeicher (*engl.: shared memory*) können kritische Abschnitte (*engl.: critical section*) im parallelen Programmablauf durch die Wahl geeigneter Datenstrukturen für die SPH-Teilchen und die Wechselwirkungslisten vollständig vermieden werden (siehe Abschnitt 4.4.1). Eine Implementierung eines SPH-Verfahrens unter den hier entwickelten Gesichtspunkten auf der NEC SX-4 konnte eine parallele Beschleunigung (*engl.: speedup*) von 19 auf einer Anzahl von 20 CPU erreichen, was einer parallelen Effizienz von 90% entspricht (siehe: Abschnitt 4.4.2).

Es konnte gezeigt werden, dass es einen 2-stufigen Gebietszerlegungs-Algorithmus (*engl.: domain decomposition*) gibt, mit dem ein SPH-Verfahren auf der Cray T3E mit einer parallelen Effizienz von 65% implementiert werden kann. Die im Abschnitt 4.5 diskutierte Vorgehensweise zur Implementierung eines SPH-Verfahrens auf der Cray T3E zeigt, dass es notwendig ist, architekturspezifische Veränderungen des Parallelisierungsansatzes einzuführen.

Nachdem die generelle Parallelisierbarkeit von SPH-Verfahren auf verschiedenen Parallelrechnertypen geklärt ist, können nun in den Kapiteln 5 und 6 die in Abschnitt 1.2 auf Seite 4 dargestellten Zielsetzungen bearbeitet werden.

# Kapitel 5

# Design-Pattern für objektorientiertes, paralleles SPH

## 5.1 Zur Problemanalyse von objektorientierten SPH-Verfahren

Im Kapitel 4 wurden SPH-Verfahren auf ihre Parallelisierbarkeit hin untersucht. Das Aktivitätsdiagramm in Abb. 4.1 zeigt als ein Ergebnis der Analyse die allgemeine Struktur einer SPH-Simulation. Die einzelnen parallelisierbaren Teile eines SPH-Verfahrens wurden im Abschnitt 4.3 beschreiben. Speziell für Maschinen mit verteiltem Hauptspeicher wurde der Ablauf einer SPH-Simulation mit Gebietszerlegungsverfahren in Abbildung 4.13 dargestellt.

Im Folgenden werden die zentralen Fragestellungen der vorliegenden Arbeit, wie sie bereits in Abschnitt 1.2 formuliert wurden, angegangen:

- Entkopplung von SPH-Verfahren und Parallelisierungsmethoden

  Es werden objektorientierte Design-Pattern entwickelt, die es ermöglichen, die Aufgabenstellung der Implementierung eines SPH-Verfahrens von der Aufgabenstellung der Parallelisierung zu entkoppeln.

- Wiederverwendbare Klassenbibliothek für paralleles SPH

  Die Design-Pattern sind so gestaltet, dass einmal gefundene Lösungen für spätere SPH-Verfahren leicht wiederverwendet werden können.

- Anpassung an verschiedene Parallelrechner

  Die entwickelten SPH-Programme können durch eine lokale, gezielte Veränderung innerhalb eines Design-Pattern an verschiedene Hardwareplattformen angepasst werden.

- Rahmenprogramm für parallele SPH-Programme

  Die Design-Pattern geben eine Vorgehensweise an, wie Experten der Fachgebiete Physik, Mathematik und Informatik an der Implementierung eines objektorientierten parallelen SPH-Verfahrens zusammenarbeiten können.

Zur Umsetzung der oben genannten Ziele werden in den Abschnitten 5.2 und 5.3 die in der vorliegenden Arbeit entwickelten Design-Pattern für SPH-Verfahren dargestellt. Die Ergebnisse des Designs werden jeweils in UML-Diagrammen festgehalten. Im Anhang A findet sich eine Zusammenfassung der Design-Ergebnisse.

Im daran anschließenden Abschnitt 5.4 wird anhand von Beispielen auf die Implementierung der im Design-Schritt entwickelten Strukturen eingegangen. Die Beispiele richten sich an den typischen Problemfällen bei der Erstellung von SPH-Programmen aus:

1. Implementierung eines SPH-Verfahrens mit bestehenden Bibliothekskomponenten (Abschnitt 5.4.1)
2. Implementierung neuer Komponenten für spezielle SPH-Verfahren (Abschnitt 5.4.2)
3. Anpassung von SPH-Programmen an eine neue Parallelrechnerplattform (Abschnitt 5.4.3)

In den folgenden Abschnitten werden zur besseren Unterscheidung Design-Pattern und Klassen-Namen durch folgende Schriftarten hervorgehoben:

- Namen von Design-Pattern: Strategy-Pattern
- Namen von Klassen (Implementierung): `Integrator.execute()`

## 5.2 Design-Pattern für das SPH-Verfahren

Zunächst werden Design-Pattern vorgestellt, die dazu dienen, SPH-Verfahren strukturiert zu implementieren. Eines der Design-Ziele ist die Entkopplung von SPH-Methode und Parallelisierung. Die im Folgenden dargestellten Design-Pattern gehen also nicht auf Aspekte der Parallelisierung ein. Allerdings müssen die Strukturen so geartet sein, dass diese Design-Pattern auch in einem parallelen Programm unverändert Bestand haben. Wie diese Design-Pattern in einem parallelen Programm verwendet werden, wird in Abschnitt 5.3 ausgeführt.

Zunächst werden Design-Pattern für Elemente aus der Mathematik, die für SPH-Verfahren benötigt werden, erklärt. In den darauf folgenden Abschnitten werden Design-Pattern, die spezifisch für SPH-Verfahren sind, diskutiert.

### 5.2.1 Design-Pattern für Verfahren aus der Mathematik

**Integratoren**

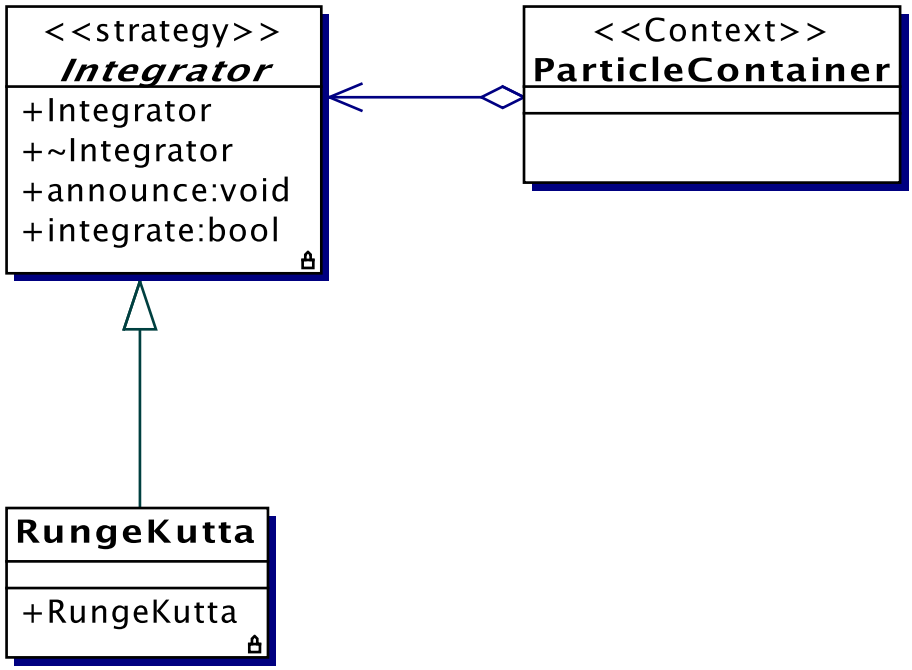



Abbildung 5.1: Auswahl von verschiedenen Integratoren über ein *Strategy*-Pattern. Siehe auch Anhang A.1.1. (Für eine Legende der Symbole für UML-Klassendiagramme siehe Abb. 2.9.)

Elemente aus der Mathematik, die in numerischen Simulationen Verwendung finden, sind *Gleichungslöser* und spezielle *Funktionen*. Zunächst werden Integratoren von gewöhnlichen Differentialgleichungen als ein Beispiel für einen in einem SPH-Verfahren vorkommenden Gleichungslöser untersucht.

Integratoren haben in der Numerik alle die gleiche äußere Form, da sich alle gekoppelten, gewöhnlichen Differentialgleichungen auf ein Problem von gekoppelten einfachen Differentialgleichungen reduzieren lassen (siehe [47]):

$$p\frac{dy_i(x)}{dx} = f_i'(x, y_1, \ldots, y_N). \qquad i = 1, \ldots, N \tag{5.1}$$

wobei $f'$ die Funktion über den Ableitungen der zu integrierenden Funktionen $y_i$ ist. $f'$ wird als *rechte Seite* (*engl.: right-hand-side*) der Differentialgleichung bezeichnet.

Integratoren unterscheiden sich nur durch die spezielle Zusammensetzung der zu integrierenden Funktion. Die Interpretation, d.h. die Bedeutung für das Simulationsprogramm, der Funktion oder der Parameter ist nicht Gegenstand der Mathematik. Eine Implementierung muss also auch keine Rücksicht darauf nehmen, ob ein

Parameter in einem Simulationsprogramm als *Zeit* oder beispielsweise als *Ort* interpretiert wird.

Die Eigenschaft, dass Integratoren gewöhnlicher Differentialgleichungen alle die gleiche Form haben, kann verwendet werden, um ein Design-Pattern für Integratoren zu finden. Eine Möglichkeit ist es, eine Klasse `Integrator` zu erstellen, die als Operationen (Methoden) verschiedene Gleichungslöser enthält. Dann entspräche eine Operation auf einem Objekt der Klasse `Integrator` dem Ausführen einer Integration. Der Operation müsste als Parameter die zu integrierende Funktion sowie geeignete Parameter übergeben werden.

Eine solche Implementierung ist eine objektorientierte Lösung des Problems, Integratoren in eine Simulation einzubringen. Diese Lösung hat aber folgende Nachteile:

- Da alle möglichen Integratoren, wie Runge-Kutta, Euler etc., als Operationen in eine Klasse implementiert werden, müssen diese Operationen alle unterschiedliche Namen haben. D.h., die Integration mit einem Euler-Integrator wäre `Integrator.Euler()` und die Integration mit einem Runge-Kutta-Integrator wäre `Integrator.RungeKutta()`. Da alle Integratoren die gleiche Form, also damit auch die gleichen Parameter haben, ist es nicht möglich, mit der objektorientierten Technik des Überladens von Operationen zu arbeiten.

  *Problem:* Die Integratoren sollen austauschbar in einem oder in verschiedenen Simulationsprogrammen verwendet werden können. Wenn die verschiedenen Integratoren aber unterschiedliche Namen haben, dann muss auch das Simulationsprogramm verändert werden, wenn ein Integrator gewechselt wird. Die Stelle im Programm, bei der der Integrator aufgerufen wird, ist abhängig vom Namen der Operation.

- Wenn ein neues Integrationsverfahren eingebunden werden soll, d.h., die Klasse `Integrator` soll erweitert werden, so müsste der Quelltext der ursprünglichen Implementierung vorliegen und verändert werden. Eine Erweiterung durch Vererbung würde eine neue Namensgebung der `Integrator`-Subklasse erfordern.

  *Problem:* Eine Veränderung eines bestehenden Quellcodes birgt die Gefahr, diesen bereits getesteten Code mit neuen Fehlern zu versehen. Die Voraussetzung, dass der Originalquellcode zur Verfügung stehen muss, ist unflexibel und unpraktikabel. Eine Auslieferung als Bibliothek würde so nicht möglich sein.

  Würden alle möglichen Integratoren in eine Klasse geschrieben, so würde diese zu groß und unhandlich. Letztlich würde ein Objekt dieser Klasse auch zu viel

Speicher zur Laufzeit verbrauchen.

Eine Erweiterung durch Vererbung aus einer Basisklasse `Integrator` verursacht die gleiche Problematik wie die verschiedenen Namen der Operationen. Der aufrufende Code müsste bei einer Erweiterung mitverändert werden.

- Offen bleibt noch die Frage, wie die Parameter für den Integrator übergeben werden. Die Parameter werden den Operationen (d.h. den verschiedenen Integratoren) durch ein *Parameterobjekt* übergeben.

  Es steht zu vermuten, wenn eine Implementierung in der oben beschriebenen Weise erfolgt, dass die verschiedenen Integratoren auch leicht voneinander abweichende Parameterlisten erhalten.

Das Strategy-Pattern gibt eine Lösung für Probleme dieser Art. Das Strategy-Pattern ist kurz im Anhang A.4.1 auf Seite 167 beschrieben (siehe auch [19]). Es wird angenommen, dass verschiedene Strategien zur Lösung eines Problems durch eine gemeinsame Abstraktion dargestellt werden können. Alle diese Strategien arbeiten auf einem bestimmten *Kontext*.

Durch Formel 5.1 ist klar, dass Differentialgleichungslöser durch eine gemeinsame, abstrakte Schnittstelle dargestellt werden können. Damit ist eine objektorientierte Implementierung für Integratoren durch das UML-Diagramm in Abbildung 5.1[1] gegeben. Die allen Integratoren gemeinsame Schnittstelle ist in der (abstrakten) Basisklasse `Integrator` implementiert. Diese Basisklasse stellt im Wesentlichen eine Operation `integrate()` zur Verfügung. Die zu integrierende Funktion mit den Anfangswerten und anderen Parametern wird durch den *Kontext* des Strategy-Patterns übergeben.

Konkrete Integratoren werden nun in Subklassen durch Vererbung aus der `Integrator`-Klasse gebildet, die nur die Operation `integrate()` überschreiben müssen.

Vorteile:

- Simulationen, die Integratoren mit diesem Design-Pattern verwenden, können ohne Veränderung des Programmcodes mit verschiedenen Integratoren gestartet werden. Alle Simulationen sehen nur die abstrakte Schnittstellenklasse `Integrator` und rufen dort die Operation `integrate()` auf.

- Die konkreten Implementierungen von Integratoren können getrennt von speziellen Simulationsprogrammen getestet werden. Durch das Strategy-Pattern sind diese Lösungen gut wiederverwendbar.

[1] Eine kurze Legende der Symbole für UML-Klassendiagramme gibt Abb. 2.9 auf Seite 36

- Soll ein neues Integrationsverfahren eingebracht werden, so ist die Stelle, an der die Erweiterung zu implementieren ist, durch das Design-Pattern klar vorgegeben. Eine neue Subklasse von `Integrator` wird erstellt und die Operation `integrate()` überschrieben.

Nachteile:

- Eine starke Zunahme der Anzahl von Klassen für Systeme mit vielen verschiedenen Integratoren.

**Kernfunktionen**

Als Beispiel für spezielle, mathematische Funktionen werden hier die in einem SPH-Verfahren verwendeten Kernfunktionen betrachtet. Auch hier lässt sich feststellen, dass alle Kernfunktionen durch eine gemeinsame, abstrakte Schnittstelle dargestellt werden können. Alle Kernfunktionen sind von der Form:

$$f \equiv f(|x - x'|, h) \tag{5.2}$$

Dabei ist $h$ der Wechselwirkungsradius und der Abstand zweier SPH-Teilchen ist gegeben durch $|x - x'|$. Damit kann auch für spezielle, mathematische Funktionen das Strategy-Design-Pattern verwendet werden. Eine abstrakte Basisklasse stellt als Operationen alle notwendigen Abbildungen der Kernfunktionen bereit. So kann also die Kernfunktion selbst durch eine Operation `kern(dist, h)` implementiert werden. Aber auch Ableitungen der Kernfunktionen können als Operationen implementiert werden.

Abbildung 5.2 auf der nächsten Seite zeigt das Design-Pattern für Kernfunktionen mit dem Beispiel der Implementierung eines Gauss- und eines Spline-Kernels. Das Beispiel zeigt, dass zur Optimierung der Kernfunktionsberechnung auch Kernfunktionen in tabellierter Form implementiert werden können, ohne dass dabei die abstrakte Schnittstelle der Kernfunktionen berührt wird (siehe Abb. 5.2, Klasse `TabularBetaSpline`). Über diesen Mechanismus können Laufzeitoptimierungen eingebracht werden, ohne dass der Rest des Programms verändert werden muss.

Der *Kontext* des Kernfunktions-Pattern sind die Eingabeparameter der Gleichung 5.2. Ein kurzer Überblick über das Design-Pattern mit verschiedenen konkreten Kernfunktions-Klassen findet sich in Anhang A.1.3.

Vorteile:

- Programmteile, die Kernfunktionen verwenden, können unabhängig von der Implementierung einer Kernfunktion arbeiten und müssen für verschiedene Kernfunktionen nicht angepasst werden.

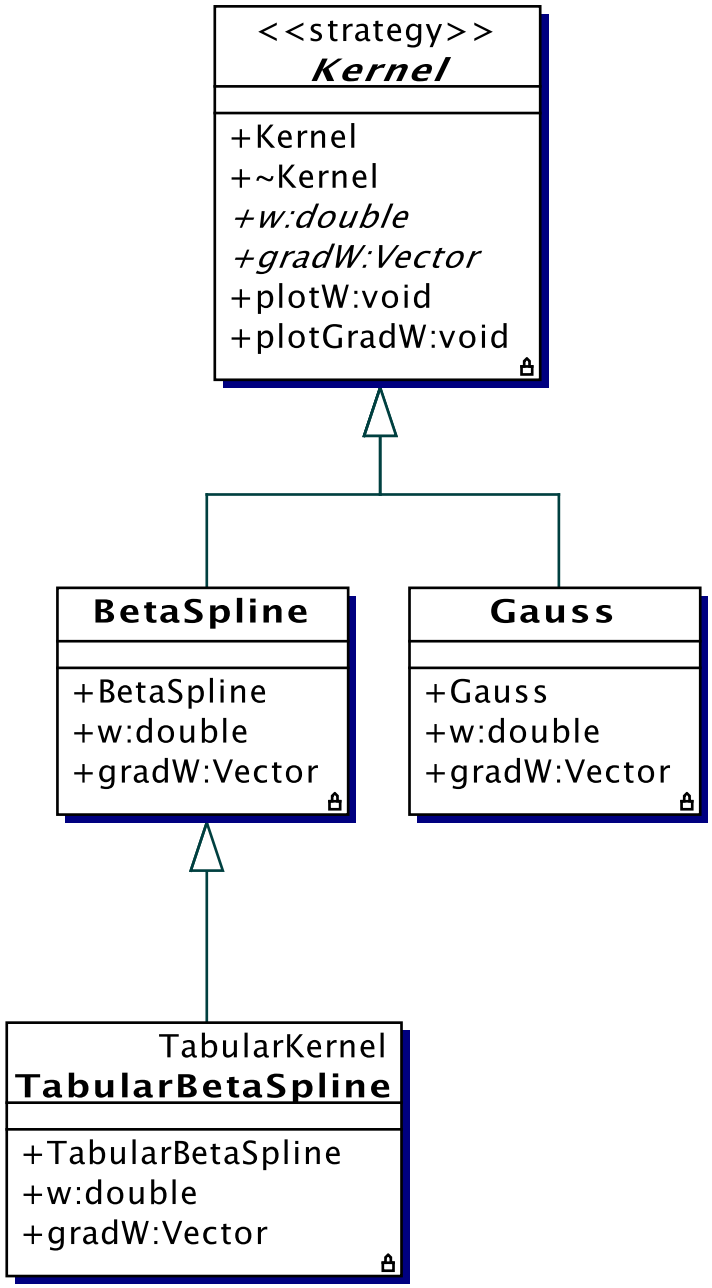



Abbildung 5.2: Auswahl von Kernfunktionen durch Strategy-Pattern.

Auch hier würde eine Programmierung mit prozeduralen Sprachen wie `C` oder `FORTRAN` nur über ein Konstrukt mit *Funktionszeiger* arbeiten können.

Nachteile:

- Der Overhead des virtuellen Methodenaufrufs zur Berechnung der Kernfunktion verursacht ggf. Effizienzverluste bei der Laufzeit.

Zusammenfassend kann festgestellt werden, dass mathematische Funktionen und Gleichungslöser durch das Strategy-Pattern abgebildet werden. Dadurch ist bei

einer Erweiterung des Systems um neue mathematische Funktionen oder Gleichungslöser ein Rahmen vorgegeben. Die Implementierung der Gleichungslöser und Funktionen ist damit unabhängig von den Programmteilen, die diese verwenden.

**Vektoren**

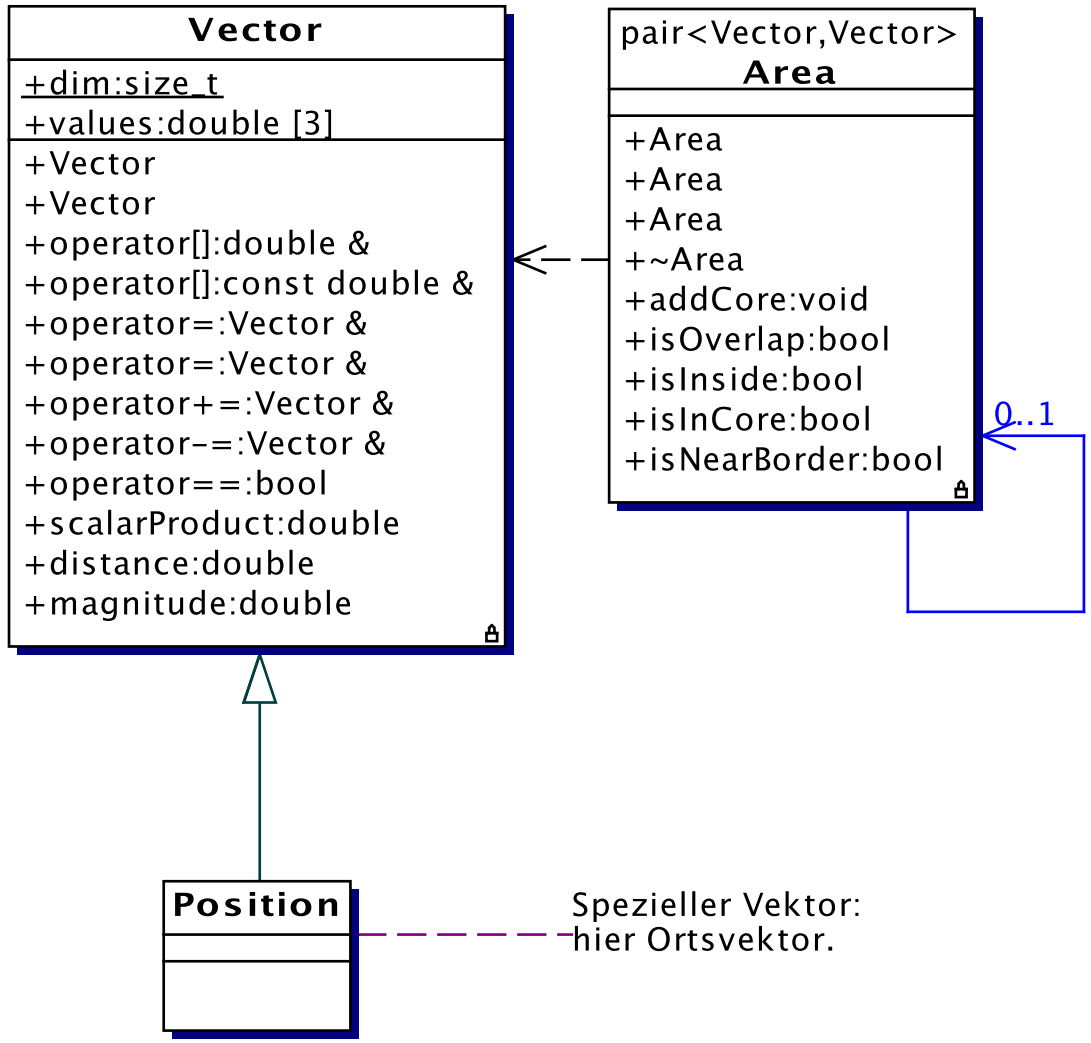



Abbildung 5.3: Vektoren als Objekte. Spezielle Vektorgrößen durch Vererbung. Zusammengesetzte Größen durch Aggregation.

Vektoren und andere mathematische Grundelemente werden als Objekte betrachtet. Speziell für die Implementierung in C++ kann hier die Möglichkeit des Überladens von *Operatoren* genutzt werden. Für Sprachen, die diese syntaktische Bequemlichkeit nicht bieten, werden *Operationen* definiert.

In der Klasse `Vector` werden alle zur Behandlung von Vektoren notwendigen Operationen definiert. Weiterführende mathematische Funktionen über Vektoren sind über die weiter oben schon beschriebenen Strategy-Pattern implementiert.

Aus der Klasse der Vektoren werden andere Größen abgeleitet. Die physikalische Größe eines Ortsvektors ist ein spezieller Vektor. Diese Größe wird durch Vererbung aus der Klasse `Vektor` erzeugt (siehe Abbildung 5.3).

Zusammengesetzte Größen, wie z.B. die Beschreibung einer Fläche, werden die Klasse `Vektor` aggregieren. Eine Fläche *hat* zur Beschreibung Vektoren, ist aber kein Vektor.

Von der häufig verwendeten Idee, Skalare, Vektoren und Matrizen über eine Vererbungshierarchie darzustellen, wurde hier aus Effizienzgründen kein Gebrauch gemacht. Vektoren, Matrizen und andere Grundelemente der Mathematik werden mit möglichst wenig Zusatz in eigenen Klassen implementiert.

### 5.2.2 Design-Pattern für SPH-Terme

#### SPH-Particles

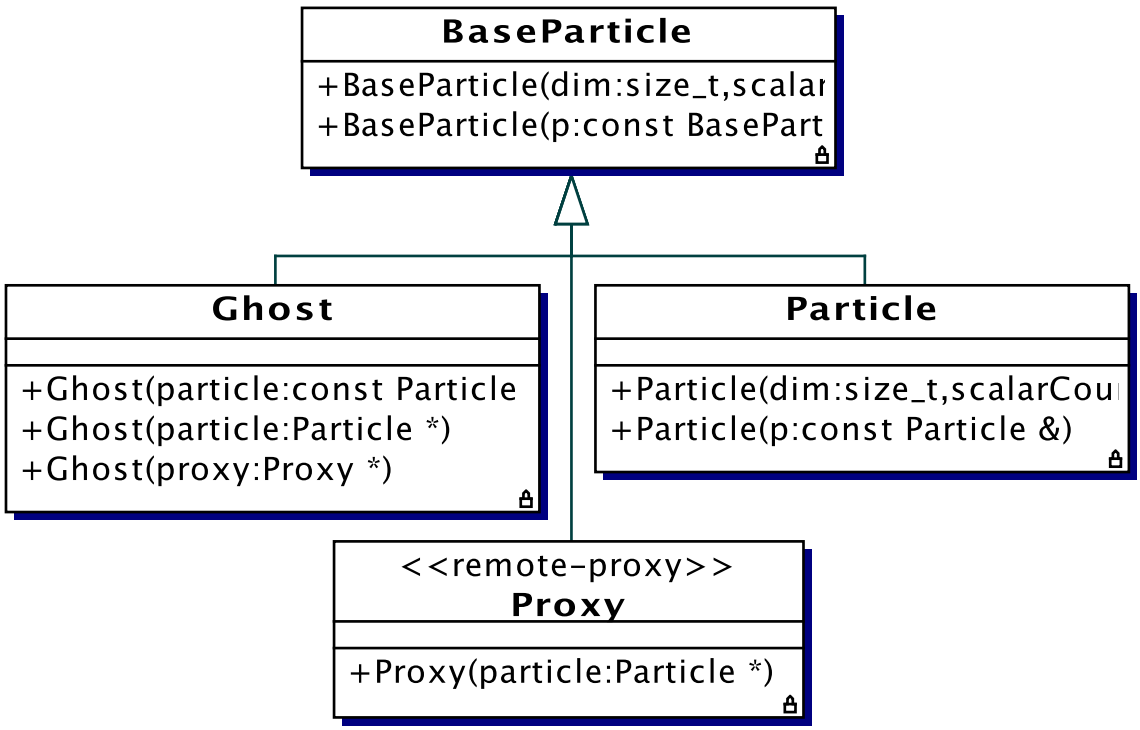



Abbildung 5.4: SPH-Teilchen durch Prototype-Pattern.

Im Folgenden werden die für SPH-Verfahren speziellen Design-Pattern diskutiert.

Zur Abbildung der SPH-Teilchen hat sich das Prototype-Design-Pattern bewährt. Das Prototype-Pattern gibt eine Klasse an, die prototypisch für eine weitere Anzahl von verschiedenen Klassen steht. Andere Klassen im System können sich auf diesen Prototyp konzentrieren und müssen nicht auf Unterschiede verschiedener Implementierungen achten. Die Klasse `BaseParticle` bildet die Schnittstellenklasse für das SPH-Teilchen-Pattern zum restlichen System (daher auch der alternative Name Interface-Pattern). Siehe [19] für eine weitere Diskussion der Einsatzmöglichkeiten des Prototyp-Design-Pattern außerhalb von Teilchensimulationsverfahren.

Aus den Voruntersuchungen aus dem Kapitel 4 ist bekannt, dass SPH-Verfahren bei der Parallelisierung an die verschiedenen Architekturen der Parallelrechner angepasst werden müssen. Für Rechner mit verteiltem Speicher müssen beispielsweise Gebietszerlegungsverfahren eingesetzt werden, um eine effiziente Parallelisierung zu erhalten. Eine Struktur für SPH-Teilchen muss also flexibel genug sein, um für alle diese Fälle verwendet werden zu können.

Aus der Diskussion der Gebietszerlegungsverfahren in Abschnitt 4.5.2 wurde deutlich, dass es zu den gewöhnlichen SPH-Teilchen für einen Rechenschritt noch Teilchen von Nachbarknoten bedarf. Diese nichtlokalen Teilchen werden als *Proxy*-Teilchen bezeichnet. Das Prototype-Pattern für SPH-Teilchen umfasst also die normalen SPH-Teilchen und die zugehörigen Proxy-Teilchen.

Vorausgreifend auf die im Kapitel 6 diskutierten Anwendungen muss auch noch auf andere SPH-Teilchen eingegangen werden. Die *Ghost*-Teilchen sind *virtuelle* SPH-Teilchen, die nicht eine physikalische Größe der Lösungsmenge, sondern eine Größe aus den Randbedingungen des Problems darstellen.[2] Mit den *Ghost*-Teilchen werden bei Problemen mit festen Randbedingungen die Ränder des Simulationsgebietes dargestellt.

Da es möglich ist, dass normale SPH-Teilchen eines Teilgebietes als Proxy-Teilchen in einem anderen Gebiet vorkommen, müssen die Teilchen ineinander überführt werden können, ohne dass es auf den restlichen Programmcode Auswirkungen hat. Für die Teile der Simulation, die mit den SPH-Teilchen rechnen, sind Implementierungsdetails nicht wichtig. Das in Abbildung 5.4 dargestellte Pattern für SPH-Teilchen genügt diesen Anforderungen. Dort sind die Konstruktoren für die SPH-Teilchen und die *Ghost*- und *Proxy*-Teilchen abgebildet. Über die Konstruktoren kann aus einem Particle ein Proxy- oder ein Ghost-Particle gemacht werden. Programmteile, die SPH-Teilchen verwalten oder bearbeiten müssen, brauchen sich aber um diesen Unterschied nicht zu kümmern. Andere Programmteile beziehen

[2] SPH-Verfahren wurden für Probleme mit offenen Rändern entwickelt, finden aber immer mehr Einsatz auch bei Problemstellungen mit festen Randbedingungen. Diese Ränder müssen in SPH-Verfahren durch *Ghost*-Teilchen dargestellt werden.

sich immer auf die *BaseParticle*-Schnittstellenklasse des Prototype-Patterns.

### Berechnung der SPH-Terme

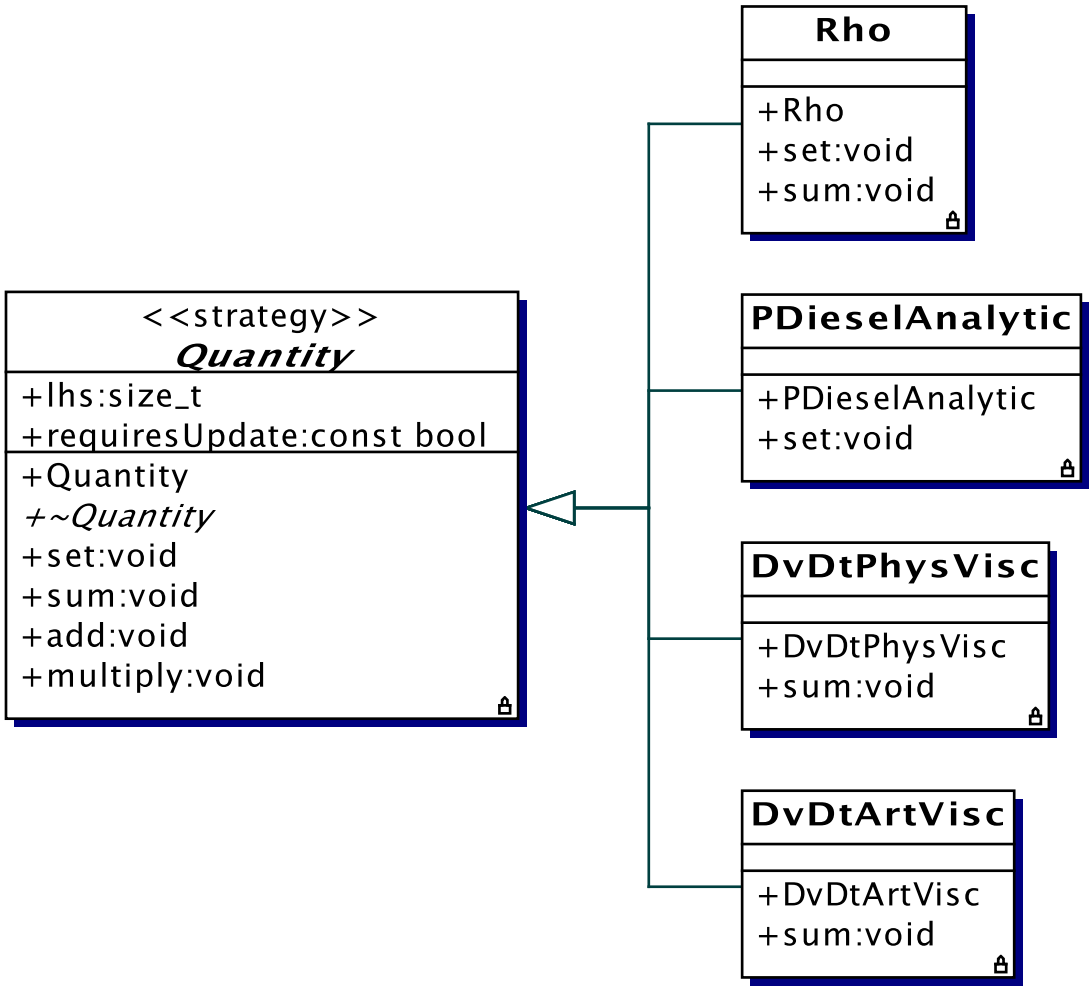



Abbildung 5.5: Berechnung der SPH-Terme.

Nachdem die Darstellung der Daten, auf denen die Berechnung von SPH-Verfahren ausgeführt wird, klar ist, muss nun geklärt werden, wie eine Gleichung in SPH-Formulierung (siehe Gl. 2.9 auf Seite 17) für ein objektorientiertes System abgebildet werden kann. Hier findet sich wieder das für mathematische Funktionen bewährte Strategy-Pattern. SPH-Terme sind ja nichts anderes als spezielle, mathematische Funktionen. Aus Darstellungsgründen wurde die Diskussion dieser SPH-Terme nicht in den Abschnitt über mathematische Funktionen gelegt.

Das Design-Pattern für SPH-Terme ist als UML-Diagramm in Abbildung 5.5 skizziert. Der *Kontext* des Strategie-Patterns — in Abb. 5.5 aus Gründen der Übersicht-

lichkeit weggelassen — ist durch die Klasse `ParticleContainer` gegeben. Ein Iterator[3] erlaubt den Zugriff auf die in einem `Particle-Container` enthaltenen SPH-Teilchen. Dabei wird über eine Liste von Teilchen vom Typ `BaseParticle` (siehe SPH-Teilchen-Pattern) iteriert. Über den *Kontext* des SPH-Term-Patterns (`Particle-Container`) und dem zugehörigen Iterator ensteht der Zusammenhang zwischen den SPH-Teilchen und den SPH-Termen.

Ein SPH-Term muss damit nichts über die Implementierungsdetails der SPH-Teilchen wissen und kann unabhängig davon entwickelt und getestet werden. Jeder SPH-Term wird in dem Strategy-Pattern als eine konkrete Subklasse von `Quantity` implementiert, so wie es schon im Design-Pattern für Integratoren im Abschnitt 5.2.1 auf Seite 101 gezeigt wurde.

Die einzelnen Terme der *rechten Seite* einer Differentialgleichung (siehe Gl. 5.1 auf Seite 101) sind die konkreten Strategien im SPH-Term-Design-Pattern. Die Berechnung der Teilchendichte wird dann beispielsweise durch eine konkrete Strategie als im SPH-Term-Pattern implementiert. In Abb. 5.5 ist der Teilchendichte-Term durch die Klasse `Rho` dargestellt. Die weiteren dargestellten konkreten SPH-Terme in Abb. 5.5 sind die Druckkraft in einem Luft-Diesel-Gemisch (Klasse `PDieselAnalytic`), die physikalische und die künstliche Viskosität einer Flüssigkeit (Klassen `DvDtPhysVisc` und `DvDtArtVisc`). Eine vollständigere Darstellung des SPH-Term-Pattern findet sich im Anhang A.1.5 auf Seite 157.

Vorteile:

- Die SPH-Terme einer Gleichung können leicht in anderen Simulationen wiederverwendet werden, da sie eine gemeinsame Schnittstelle durch das Strategy-Pattern erhalten.

  Beispielsweise können die schon im Abschnitt 4.2 diskutierten Terme der Gleichung 2.2 auf Seite 12 direkt als konkrete Strategien implementiert werden.

- Die SPH-Terme sind unabhängig vom Rest des Systems.

Nachteile:

- Das Strategy-Pattern bringt hier einen Laufzeitoverhead bei der Berechnung der SPH-Terme ein.

#### Der Quantity-Builder

Das fehlende Glied in der Kette, um eine vollständige SPH-Simulation aufbauen zu können, ist der Quantity-Builder. Der Quantity-Builder ist dem Builder-Design-Pattern,

[3] Ein Iterator kann als Design-Pattern verstanden werden.

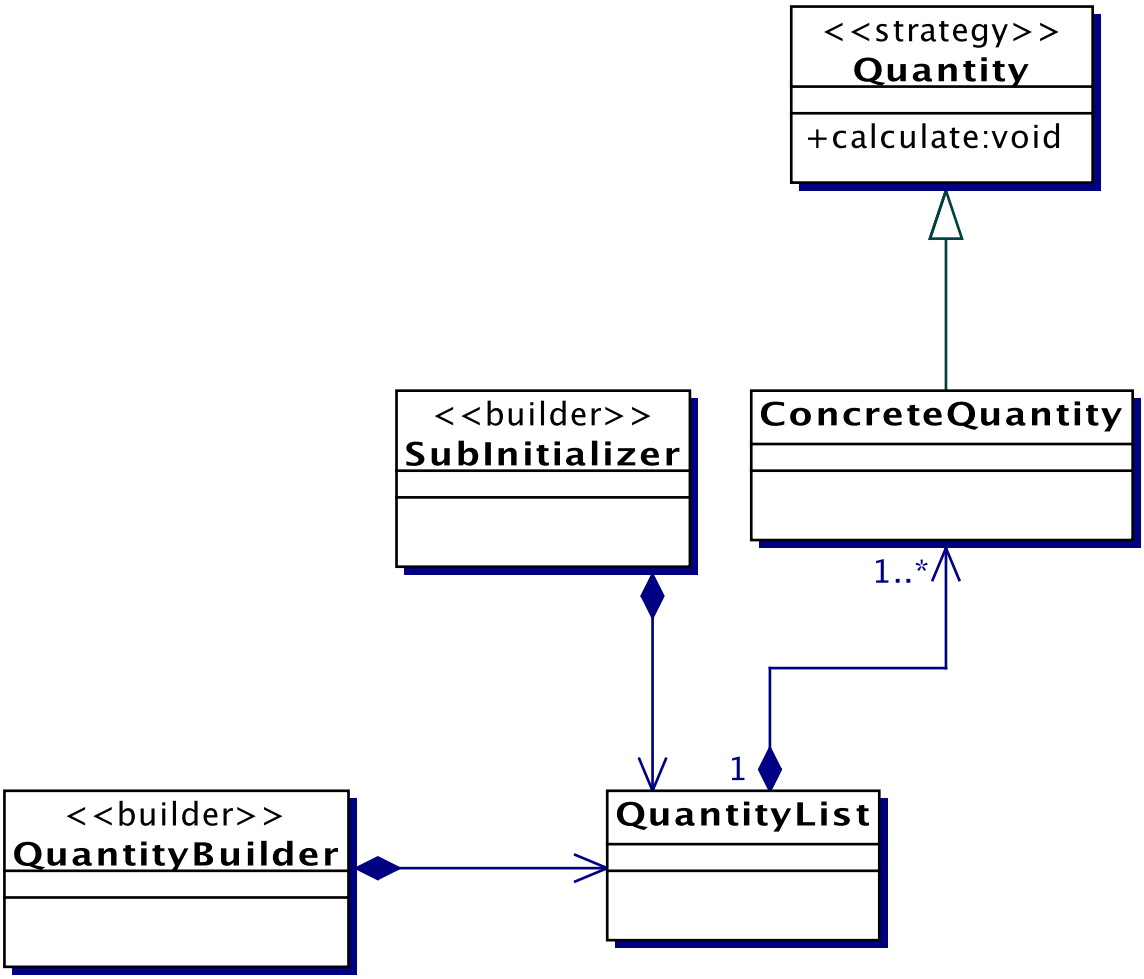



Abbildung 5.6: Erstellen der SPH-Terme durch den Quantity-Builder. Siehe auch Anhang A.16.

wie es in [19] beschrieben ist, ähnlich. Siehe auch Anhang A.16.

Im Quantity-Builder werden die konkreten SPH-Terme aus dem SPH-Term-Pattern ausgewählt und in Form einer Quantity-Liste zusammengefügt. Die Klassen `Quantity-List` und `Quantity-Builder` stehen dabei als Rahmenklassen in dem Design-Pattern Quantity-Builder zur Verfügung (siehe Abb. 5.6).

Der Entwickler einer SPH-Simulation wählt aus den konkreten Strategien des SPH-Term-Pattern die gewünschten SPH-Terme aus und stellt somit die SPH-Formulierung eines Problems zusammen. Dabei muss der Programmierer nicht auf die mögliche Ausführungsform des Programms eingehen. Eine parallele oder serielle Ausführung berührt das Erstellen der SPH-Simulation nicht. Dafür ist im Quantity-Builder-Pattern die Klasse `SubInitializer` zuständig (siehe Abschnitt 5.3.3).

Der einzige Hinweis auf die Möglichkeit zur parallelen Ausführung ist in der Property `requiresUpdate` der Schnittstellen-Klasse `Quantity` des SPH-Term-Pattern ent-

halten (siehe Abb. 5.5). Der Programmierer der konkreten SPH-Terme muss dabei angeben, ob nach der Berechnung eines SPH-Terms eine Datensynchronisation der SPH-Teilchen durchgeführt werden muss. Diese Aufgabe kann jedoch auf einer hohen Abstraktionsebene beschrieben werden:

> *Falls ein SPH-Term mit SPH-Teilchen rechnet, die eine Wechselwirkungsbeziehung zu anderen SPH-Teilchen haben, dann muss die* `requiresUpdate`-*Property gesetzt werden.*

Es ist also möglich, die späteren Synchronisations- und Kommunikationspunkte für einen eventuellen parallelen Programmablauf festzulegen, ohne dass der Entwickler des SPH-Programms sich mit den Details einer Parallelisierung auseinander setzen muss. Die Beschreibung für die Datensynchronisation ist hier auf die Abstraktionsebene des SPH-Verfahrens angehoben worden.

### 5.2.3 Zusammenhang der Design-Pattern

Die bisher vorgestellten Design-Pattern für eine SPH-Simulation greifen wie folgt ineinander:

- Die Grundlage aller Simulationen bildet das Design-Pattern SPH-Teilchen. Die SPH-Teilchen sind die Daten, auf denen die Berechnung ausgeführt wird. (Siehe Abb. 5.4.)
- Neue SPH-Verfahren werden durch Implementieren von konkreten SPH-Termen im Design-Pattern SPH-Term entwickelt (Abb. 5.5). Es steht ein Rahmen zur Verfügung, in den neue SPH-Terme eingebunden werden können, ohne dass dabei auf andere Teile der Simulation oder auf die Art der Ausführung des Programms — seriell oder parallel — eingegangen werden muss.

  Bei der Implementierung von SPH-Termen wird auf die Bibliothek von mathematischen Funktionen, wie beispielsweise das Kernel-Pattern, zugegriffen. Sind neue Funktionen oder Integratoren notwendig, so werden diese mit den dafür vorgesehenen Design-Pattern implementiert.
- Die implementierten SPH-Terme werden über das *Quantity-Builder*-Pattern zusammengefügt (Abb. 5.6.)
- Vollständig unabhängig davon kann über das Integrator-Pattern ein geeigneter Integrator für die Simulation ausgewählt werden. (Siehe Abb. 5.1).

Im Folgenden werden die Design-Pattern diskutiert, die eine serielle oder parallele Ausführung des Programms ermöglichen. Beispiele für die Implementierung der SPH-Design-Pattern sind in Abschnitt 5.4 gegeben.

## 5.3 Design-Pattern zur Parallelisierung von Teilchenmethoden

Im Abschnitt 5.2 wurden Design-Pattern zur Implementierung von SPH-Verfahren vorgestellt, ohne dass dabei auf die Art der Ausführung des Programms explizit eingegangen wurde. Ob ein SPH-Programm seriell oder parallel ausgeführt wird, ob es auf einer Maschine mit verteiltem Speicher oder auf einer Maschine mit gemeinsamem Speicher parallelisiert wird, ging aus den Design-Pattern für SPH-Verfahren nicht hervor. Genau diese starke Entkopplung von SPH-Verfahren und Parallelisierung macht eine Implementierung mit den hier vorgestellten Design-Pattern interessant.

Im Folgenden werden die Design-Pattern besprochen, die im Zusammenspiel mit den im Abschnitt 5.2 vorgestellten Design-Pattern einen wahlweise parallelen oder seriellen Ablauf ermöglichen. Dabei wird zunächst auf Strukturen eingegangen, die es ermöglichen, ein SPH-Verfahren so zu partitionieren, dass eine Ausführung auf verschiedenen Rechnertypen möglich ist. Da im Allgemeinen davon ausgegangen werden muss, dass das SPH-Programm auf Rechnern mit verteiltem Hauptspeicher zur Ausführung kommt, müssen die hier diskutierten Design-Pattern flexibel genug sein, um diese Anforderung abzudecken.

### 5.3.1 Design-Pattern für Simulationsgebiete

Das Simulation-Pattern ist die Grundstruktur eines objektorientierten SPH-Verfahrens. Das Simulation-Pattern muss eine Ausführung eines SPH-Programms auf seriellen wie auch parallelen Rechnertypen ermöglichen. In der Analyse im Kapitel 4 wurde ein SPH-Verfahren in seine parallelisierbaren Teile strukturiert und die speziellen Anpassungen für verschiedenen Parallelrechnertypen diskutiert.

In Abbildung 5.7 sind die Klassen einer Subsimulation dargestellt. Aus dem Diagramm ist erkennbar, dass die Basisklasse `BaseSubSimulation` des Design-Patterns Simulation spezialisiert wird in Subsimulationen mit besonderen Aufgabestellungen. Die Subsimulation zur Berechnung der SPH-Terme wird in der Klasse `SubSimulation` implementiert. Für Simulationsgebiete, die den Rand einer Simulation darstellen, ist eine spezielle Klasse `BoundarySimulation` von der Klasse `BaseSubSimulation` abgeleitet. Diese `BoundarySimulation` wird weiter spezialisiert in Simulationsgebiete, die das Einbringen und Entnehmen von SPH-Teilchen simulieren.

Auf dieser Grundlage kann ein Simulationsraum in Teilräume mit jeweils speziellen Verantwortlichkeiten zerlegt werden. In Abb. 5.8 ist für einen zweidimen-

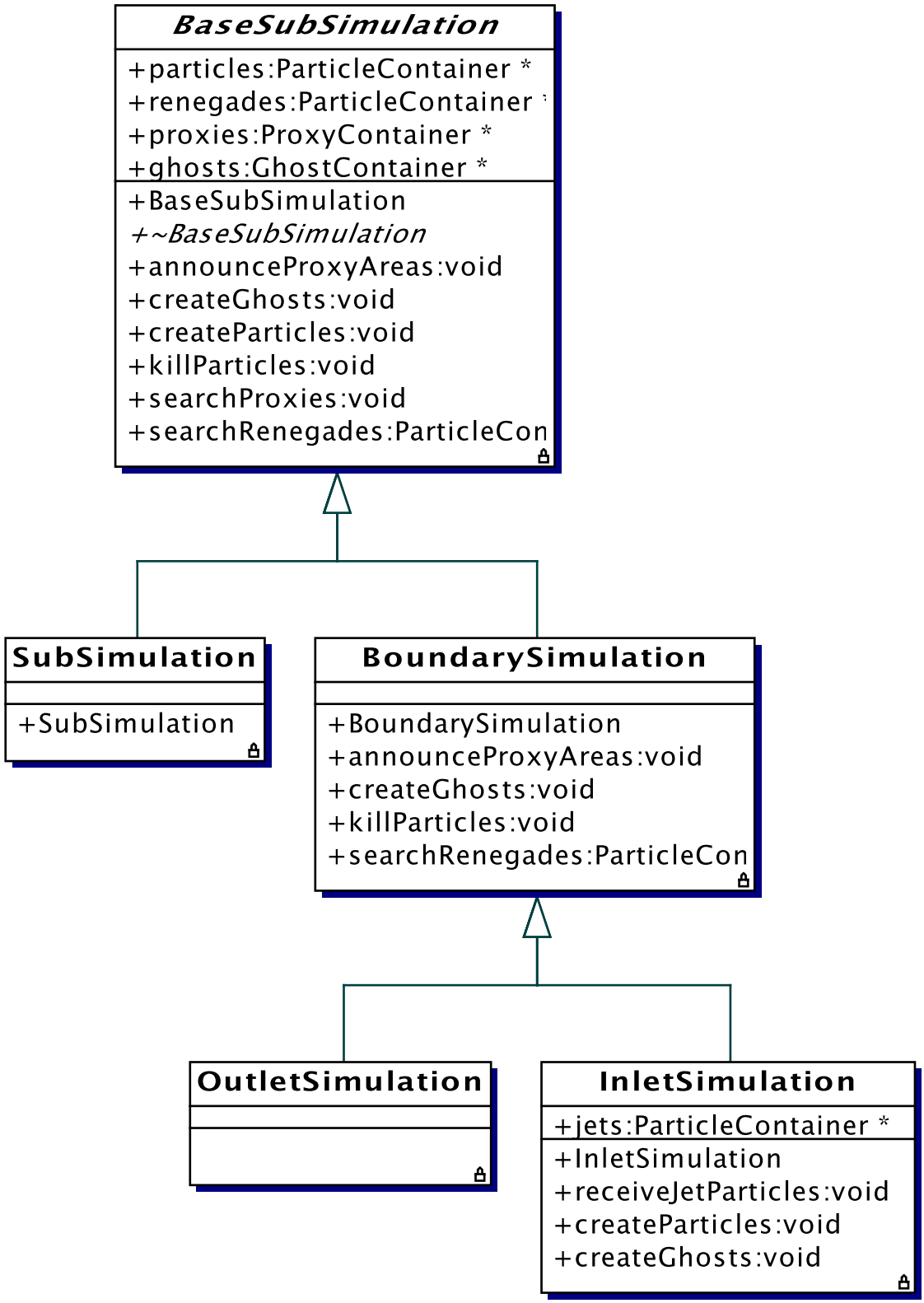



Abbildung 5.7: Das Simulation-Design-Pattern als Grundstruktur einer SPH-Simulation.

| | | | | |
|---|---|---|---|---|
| Boundary Simulation | Boundary Simulation | Boundary Simulation | Boundary Simulation | Boundary Simulation |
| Inlet Simulation | Sub Simulation | Sub Simulation | Sub Simulation | Boundary Simulation |
| Inlet Simulation | Sub Simulation | Sub Simulation | Sub Simulation | Outlet Simulation |
| Inlet Simulation | Sub Simulation | Sub Simulation | Sub Simulation | Boundary Simulation |
| Boundary Simulation | Boundary Simulation | Boundary Simulation | Boundary Simulation | Boundary Simulation |

Abbildung 5.8: Beispiel für die Anordnung verschiedener konkreter Simulationsklassen eines Simulation-Patterns.

sionalen Fall ein Simulationsgebiet in gleichmäßige Subsimulationsgebiete aufgeteilt. Dabei können verschiedene konkrete Simulationsklassen eingesetzt werden, um den gesamten Simulationsraum zu definieren. Eine solche Einteilung ist für den SPH-Programmierer intuitiv und nahe an der eigentlichen Problemstellung des SPH-Verfahrens.

### 5.3.2 Design-Pattern für Kommunikatoren

Mit Hilfe des Simulation-Pattern kann der logische Simulationsraum in Teilräume unterteilt werden. Durch das Simulations-Pattern wird aber nicht festgelegt, wie die einzelnen Simulationsteilräume miteinander in Verbindung stehen. Speziell die Datenkommunikation zwischen den Teilräumen ist nicht in der Verantwortung des Simulation-Pattern.

Dem Kommunikator-Pattern wird die Aufgabe der Synchronisation und Kommunikation der möglichen parallelen Ausführungseinheiten (Subsimulationen) zugewiesen. Das Kommunikator-Pattern besteht aus zwei zusammenhängenden Design-Pattern: dem PrincipalCommunicator- und dem SubCommunicator-Pattern. Die beiden Teile des Kommunikator-Pattern folgen dabei jeweils der Idee des bereits bekannten

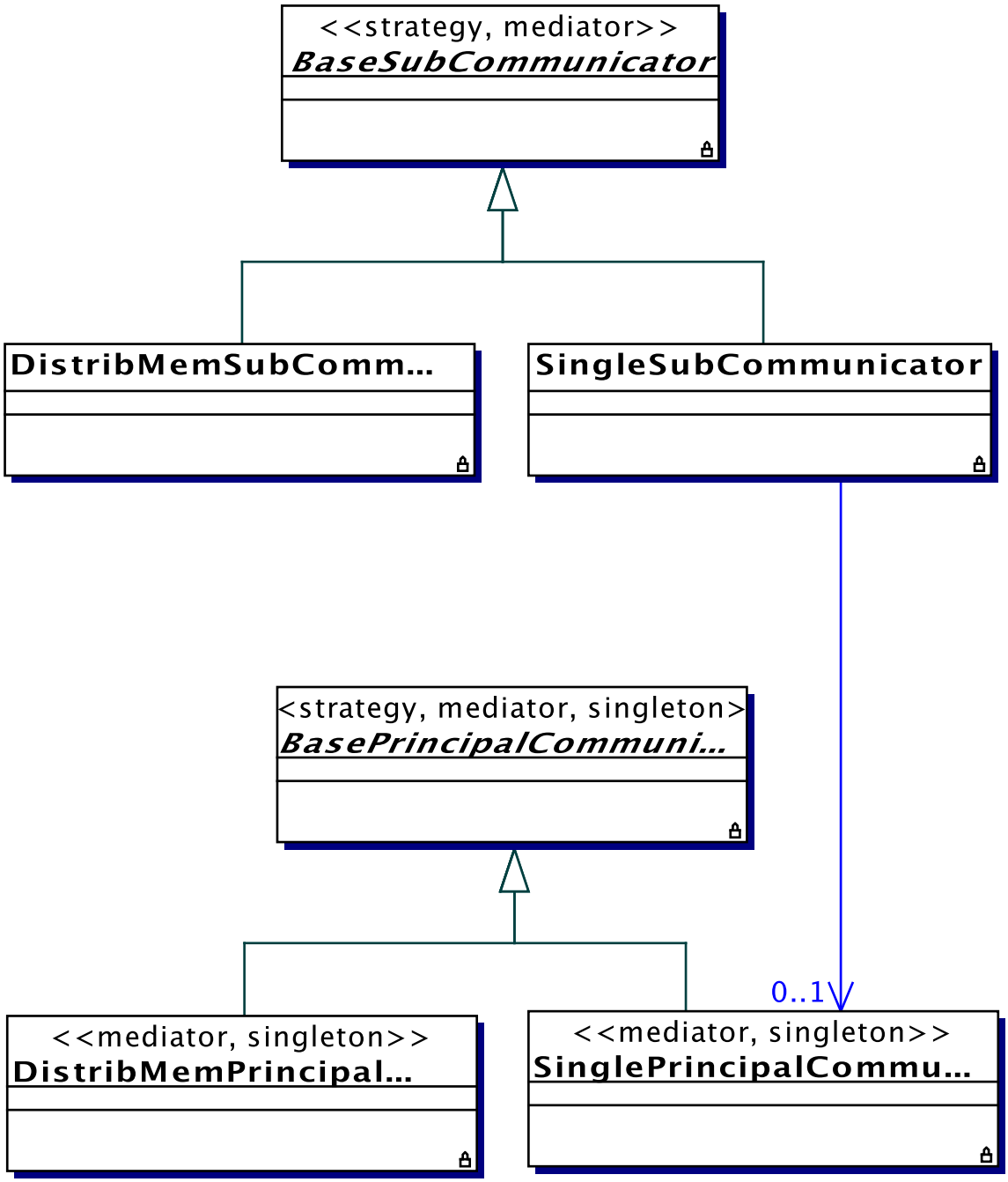



Abbildung 5.9: Principal und Sub-Kommunikatoren.

Strategy-Pattern. (Siehe Abb. 5.9).

Die Klasse `BasePrincipalCommunicator` bildet die Basisklasse des Design-Pattern der Principal-Kommunikatoren. Diese Kommunikatoren haben eine übergeordnete Verantwortlichkeit und koordinieren die Sub-Kommunikatoren. Im Falle einer Ausführung auf einem Rechner mit verteiltem Hauptspeicher übernehmen die Principal-Kommunikatoren die Aufgaben die Sub-Kommunikatoren über die Knoten

des Rechners zu verteilen und zu koordinieren.

Die zweite Hälfte des Kommunikator-Pattern besteht aus den Sub-Kommunikatoren mit der Klasse `BaseSubCommunicator` als Basisklasse. Jeder Sub-Kommunikator hat die Aufgabe, mit einem oder mehreren zugewiesenen Sub-Kommunikatoren synchronisiert zu kommunizieren. Das Wissen über die Topologie des Zusammenhangs der Sub-Kommunikatoren ist nicht in der Verantwortung der Sub-Kommunikatoren selbst, sondern wird von den Principal-Kommunikatoren übernommen. Die tatsächliche Datenkommunikation ist in den Sub-Kommunikatoren implementiert.

Im Design-Pattern der Kommunikatoren findet die Unterscheidung der verschiedenen Rechnertypen statt. Die konkreten Implementierungen der Basisklassen der beiden Teile des Design-Pattern haben das Wissen über die vorliegende Topologie. Für serielle (von Neumann) Rechner sind die konkreten Subklassen `SinglePrincipalCommunicator` und `SingleSubCommunicator` implementiert. Werden diese Kommunikatoren den konkreten Simulationsgebieten aus dem Simulation-Pattern zugewiesen, so erfolgt eine Programmausführung mit Kommunikation auf serieller Weise. Die Kommunikation zwischen Sub-Kommunikatoren erfolgt hier über einen kontrollierten Zugriff auf gemeinsame Objekte (*engl.: shared objects*).

Zur Ausführung auf einer Maschine mit verteiltem Hauptspeicher werden den Simulationsgebieten Kommunikatoren der Klassen `DistribMemSubCommunicator` und `DistribMemPrincipalCommunicator` zugewiesen. Diese Varianten der Kommunikatoren bilden die Schnittstelle zu betriebssystemnahen Kommunikationsbibliotheken wie beispielsweise MPI oder dem am Wilhelm-Schickard-Institut der Universität Tübingen entwickelten objektorientierten System TPO++.

Für Rechner mit gemeinsamem Hauptspeicher werden eigene Sub-Kommunikatoren implementiert (in Abb. 5.9 nicht dargestellt). Durch Implementieren dieser spezialisierten Sub-Kommunikatoren für Shared-Memory-Architekturen wird das SPH-Programm auf einem Rechner wie beispielsweise der NEC SX-4 ausführbar.

Über dieses Design-Pattern ist die technische Umsetzung der Kommunikation und Synchronisation zwischen parallelen Einheiten gekapselt gegenüber dem Rest des Systems. Die Subsimulationsgebiete brauchen kein Wissen über die physikalische Topologie des Rechners, auf dem das Programm abläuft. Damit ist es unter der Voraussetzung, dass Rechner binärkompatibel sind, möglich, ein und dasselbe Programm ohne erneutes Kompilieren auf einem Rechner mit verteiltem Hauptspeicher sowie auf einem seriellen Rechner auszuführen. Das Linux-Cluster *Kepler* ist ein Beispiel für diese Möglichkeit (siehe Abschnitt 6.2). Ein Beispiel für eine Implementierung des Kommunikator-Pattern findet sich in Abschnitt 5.4.3.

### 5.3.3 Zusammenhang der Design-Pattern

Mit den bisher vorgestellten Design-Pattern ist es vollständig möglich, ein SPH-Verfahren parallel und objektorientiert zu implementieren. Zum Abschluss muss noch der Überblick der Zusammenhänge der SPH-Design-Pattern geklärt werden. Dabei wird auch die Initialisierung des Systems veranschaulicht und mit dem Mediator-Pattern ein weiteres, notwendiges Design-Pattern eingeführt.

#### Initialisierung des Systems

Abbildung 5.10 zeigt das UML-Klassendiagramm der Zusammenhänge zwischen einigen wichtigen Klassen aus den zentralen SPH Design-Pattern. Zu erkennen sind dabei die Klassen `Kernel` und `Integrator` aus den Strategie-Pattern, die im Abschnitt 5.2.1 bereits besprochen wurden.

Die Klasse `RHSCalculator` implementiert hier einen Iterator[4] zur Berechnung der rechten Seite eines Integrators und verwendet damit das SPH-Term-Pattern und das Quantity-Builder-Pattern zur Berechnung der SPH-Terme.

Die Verantwortlichkeiten der Klassen `BaseSubSimulation` und `BaseSubCommunicator` wurden bei der Besprechung der Design-Pattern Simulation und Kommunikator ebenfalls bereits vorgestellt.

Neu sind die beiden Klassen `Mediator` und `SubInitializer`. Die Klasse `Mediator` stellt dabei mit den Klassen `BaseSubCommunicator` und den Klassen zur Berechnung eines SPH-Terms ein Mediator-Pattern dar (siehe Abbildung im Anhang A.15 auf Seite 170). Die Verantwortlichkeit des Mediators ergibt sich direkt aus der Anforderung, dass die Berechnung der eigentlichen SPH-Simulation von der Aufgabe der Parallelisierung entkoppelt sein muss. Die Parallelisierung, also Kommunikation und Synchronisation, sowie die Berechnung der SPH-Terme hängen dennoch implizit zusammen. Die Verknüpfung der beiden Aufgabestellungen kann jedoch weder den Design-Pattern, die für die Parallelisierung zuständig sind, noch den Design-Pattern für das SPH-Verfahren zugeordnet werden, ohne dass dabei ungewollte, enge Kopplungen zwischen diesen beiden Verantwortlichkeiten entstehen.

Das Meta-Pattern (siehe [24]) *Pure Fabrication* bietet als Lösung für dieses Problem die Einführung rein künstlicher Klassen, die diese Verantwortlichkeit übernehmen.[5] Das *Mediator*-Design-Pattern eignet sich zur Implementierung dieses Meta-Pattern besonders. Die Klasse `Mediator` übernimmt die Aufgabe, die Sub-Kommunikatoren mit den zugehörigen Subsimulationen und allen Teilen des SPH-

[4] Ein Iterator kann auch als Design-Pattern verstanden werden.

[5] Da dieser Klasse kein Name aus dem Problemraum von SPH-Verfahren zugewiesen werden kann, wird diese hier gleich dem verwendeten Design-Pattern genannt: Mediator.

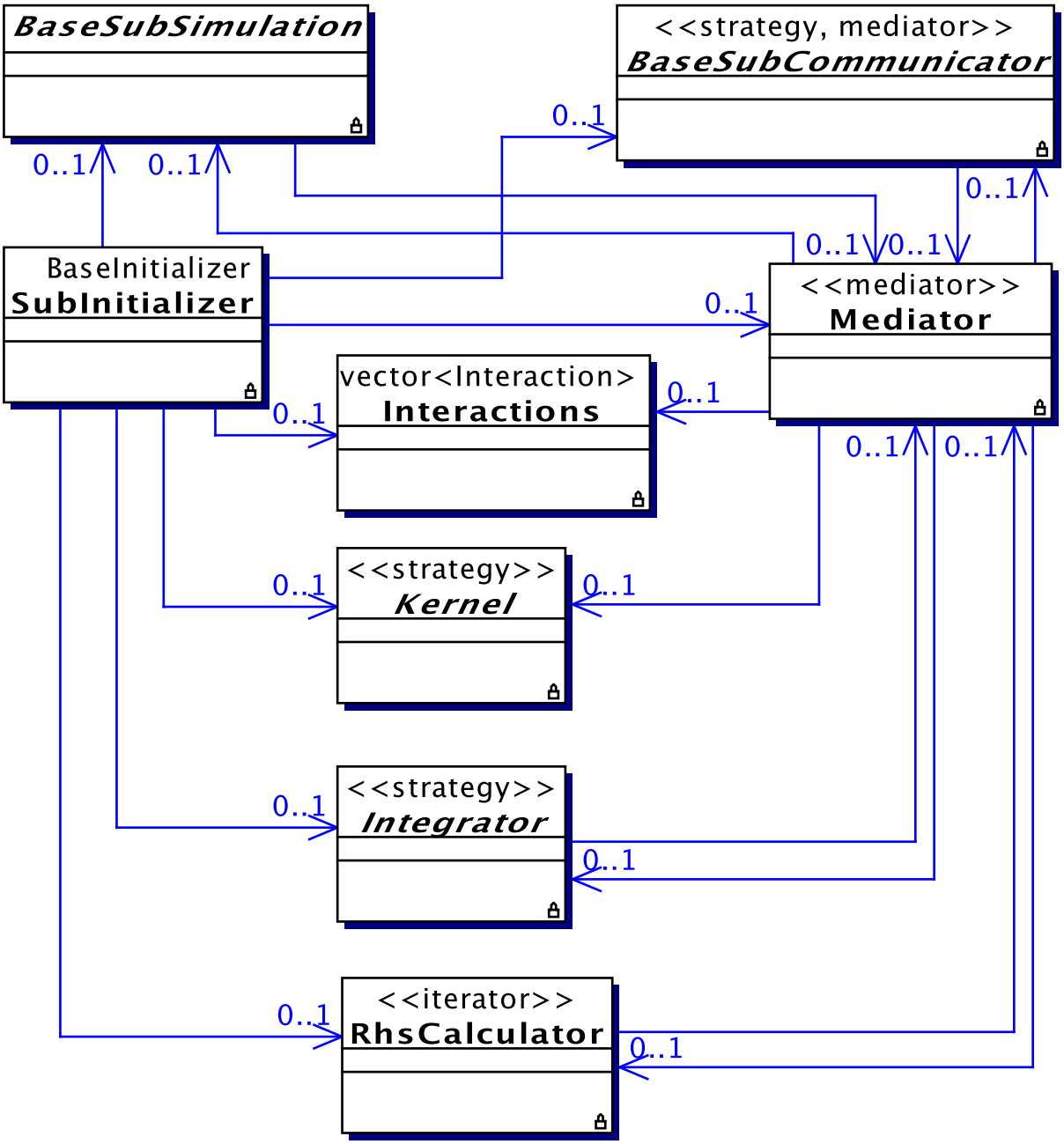



Abbildung 5.10: Initialisierung einer Simulationseinheit.

Verfahrens zu koordinieren. Die Klasse `Mediator` ist damit stark an verschiedene Teile des Systems gekoppelt und schlecht wiederverwendbar. Dieses *Opfer* zahlt sich aus, da dadurch die Klassen zur Berechnung des SPH-Verfahrens und die Klassen zur Parallelisierung eines SPH-Verfahrens entkoppelt werden.

Im Diagramm in Abbildung 5.10 ist dies sichtbar. Die Pfeile in dem UML-Diagramm zeigen allgemeine Abhängigkeitsbeziehungen zwischen den beteiligten

Klassen auf. Es ist zu sehen, dass zwischen den Klassen zur Berechnung der SPH-Terme (`Kernel`, `Integrator`, `RhsCalculator`, `Interactions`) und den Klassen zur Parallelisierung des Systems (`BaseSubSimulation`, `BaseSubCommunicator`) keine direkten Abhängigkeiten bestehen.

Die Klasse `SubInitializer` hat die Aufgabe, das System zu starten. Hier werden die geeigneten Kommunikatoren, Simulationsklassen und SPH-Terme ausgewählt. Dies geschieht im Allgemeinen über vom Benutzer vorgegebenen Eingabe-Parameter. Diese können über eine graphische Oberfläche oder eine Datei eingelesen werden.

## 5.4 Zur Implementierung der SPH-Design-Pattern

Die Anwendung der in den Abschnitten 5.2 und 5.3 beschriebenen Design-Pattern für objektorientiertes paralleles SPH wird im Folgenden anhand von Beispielen erläutert. Die Beispiele richten sich an den wichtigsten Anwendungsfällen aus:

1. Implementierung eines SPH-Verfahrens mit bestehenden Bibliothekskomponenten (Abschnitt 5.4.1)
2. Implementierung neuer Komponenten für spezielle SPH-Verfahren (Abschnitt 5.4.2)
3. Anpassung von SPH-Programmen an eine neue Parallelrechnerplattform (Abschnitt 5.4.3)

### 5.4.1 SPH-Verfahren unter Verwendung bestehender Komponenten

Um ein SPH-Verfahren mit bestehenden Bibliothekskomponenten zu implementieren, muss im einfachsten Fall nur die Konfigurationsdatei des SPH-Rahmenprogramms angepasst werden.

Die Implementierung des SPH-Rahmenprogramms heißt `sph2000` (siehe auch [27] und [20]). Die vollständige Konfigurationsdatei ist im Anhang B abgedruckt. Die Zeilennummern der im vorliegenden Abschnitt abgebildeten Auszüge aus der Konfigurationsdatei ermöglichen ein leichtes Wiederfinden der entsprechenden Stelle in der Konfigurationsdatei im Anhang.

Für das Beispiel nehmen wir an, dass alle SPH-Terme, Integratoren und andere Funktionalitäten bereits implementiert sind. Dann müssen nur folgende Komponenten ausgewählt werden:

- die richtige Kernfunktion
- ein passender Integrator
- die notwendigen SPH-Terme

Die richtigen Komponenten zu bestimmen ist eine rein fachliche Aufgabe des Experten für SPH-Verfahren.

**Die Kernfunktion**

```
57  ### Choose a kernel function.
58  #
59  # Possible values are:
60  #   BetaSpline, Box, Cusp, Spline35.
61  #  (Do not use the Gauss kernel, it is unsteady)
62  Kernel : Cusp
```

Aus den vorhandenen Kernfunktionen (siehe Kernfunktions-Pattern im Anhang A.1.3 auf Seite 155) wird die für das Simulationsvorhaben korrekte Kernfunktion ausgewählt. Dazu wird in der Konfigurationsdatei der entsprechende Parameter gesetzt.

**Der Integrator**

```
74  ### Choose an integrator.
75  # More about Timing see below.
76  # Possible values are:
77  #  Euler:      Runge Kutta of first order
78  #  Heun:       Runge Kutta of second order
79  #  RungeKutta: *The* Runge Kutta method, 4.th order
80  Integrator : Heun
```

Aus den vorhandenen Integratoren (siehe Integrator-Pattern im Anhang A.1.1 auf Seite 152) wird der für das Simulationsvorhaben korrekte Integrator ausgewählt.

**Die SPH-Terme**

```
50  ### Choose the kind of simulation.
51  # More about injection and fluids see below.
52  # Possible values are:
53  #    Injection: Diesel injection into a air filled internal-
54  # combustion engine
55  Simulation : Injection
```

Aus den verschiedenen Szenarien für SPH-Simulationen kann eins ausgewählt werden. Hier nur das Szenario: *Injection*. Dies konfiguriert das Simulation-Pattern (Anhang A.1.2 auf Seite 153). Die verschiedenen Simulationsklassen sind im Abschnitt 5.3.1 beschrieben.

```
130  ### Boundary condition: What does a particle, when it crosses
131  # the boundary?
132  # Reflecting: it reflects back into the simulation area.
133  # Open: it leaves the simulation area forever.
134  # Periodic: it leaves this side an injects on the opposite side.
135  Boundary : Reflecting
```

Die speziellen Simulationsklassen, die für die Randbedingungen zuständig sind, können über den Parameter *Boundary* konfiguriert werden.

```
197  ### Set the position of the nozzle/inlet
198  # If you want to simulate the whole jet, then choose 'Middle';
199  # if you want to simulate the half jet, generating the whole
200  # by mirroring (like in sph98), then choose 'Corner'.
201  # By default, the inlet is on the y-axis, injecting into the
202  # x-direction.
203  # Possible values are:
204  # Corner # the x-y-Corner, x=0, y=0
205  # Middle # in the middle of the y-axis, x=0
206  InletPosition : Middle
```

Die *Inlet*-Subsimulationen werden über den Parameter *InletPosition* konfiguriert.

**SubInitializer** Die Konfigurationsdatei wird nun über den SubInitializer (siehe Abschnitt 5.3.3 auf Seite 118) eingelesen und zur Programmlaufzeit als ein Parameterobjekt den einzelnen Subsimulationsgebieten zur Verfügung gestellt. Über das Parameterobjekt erstellt der Quantity-Builder (siehe Abschnitt 5.2.2 auf Seite 110) die Liste der SPH-Term-Objekte.

Für den Fall, dass alle Komponenten einer SPH-Simulation in der Bibliothek vorhanden sind, können die Komponenten über eine Konfigurationsdatei zusammengestellt werden, ohne dass das Programm neu erstellt werden muss. Es ist offensichtlich, dass die Eingabe der Parameter für die Konfigurationsdatei über eine graphische Benutzerschnittstelle erfolgen kann.

### 5.4.2 Implementierung neuer SPH-Verfahren

Ein weiterer wichtiger Anwendungsfall ist die Implementierung eines neuen SPH-Verfahrens. Dazu müssen neue Komponenten der Bibliothek hinzugefügt werden. Auch hier bieten die SPH-Design-Pattern ein Rahmenprogramm zur Unterstützung dieser Tätigkeit an.

An einem Beispiel werden folgende neue Funktionalitäten der SPH-Bibliothek hinzugefügt:

- eine neue Kernfunktion

- ein neuer Integrator
- ein neuer SPH-Term

Dazu wird das Kernel-Pattern, das Integrator-Pattern und das SPH-Term-Pattern verwendet. Die weiteren Design-Pattern bleiben unverändert, da alle SPH-Verfahren von der gleichen Struktur sind.

**Neue Kernfunktion**

```
1   class Gauss : public Kernel
2   {
3    public:
4
5     /** Constructor to initialize the data members */
6     Gauss(unsigned long group, size_t dim, double smoothingLength);
7     /** Calculates the kernel value */
8     virtual double w(double distance) const;
9     /** Calculates the kernel derivation */
10    virtual Vector
11      gradW(double distance, const Vector& distanceVector) const;
12
13  private:
14
15    /** The inverse smoothing length */
16    double reciprocH;
17    /** A pre-factor for w */
18    double factorW;
19    /** A pre-factor for grad w */
20    double factorGradW;
21  };
```

Eine neue Kernfunktion wird mittels des Kernel-Patterns (Anhang A.1.3) implementiert. Der neue Kernel — im Beispiel ein Gauss-Kernel — muss von der Basisklasse `Kernel` ableiten. Die Klasse `Kernel` gibt die zu überschreibenden Methoden (`w()` und `gradW()`) sowie die Signatur des Konstruktors vor. Im Anhang B auf Seite 187 ist der Quelltext der Implementierung des Gauss-Kernels angegeben.

Die Implementierung dieser Klasse ist ausreichend, um eine neue Kernfunktion in die Bibliothek für SPH-Verfahren einzubringen. Der Entwickler kann sich lokal und gezielt auf die Aufgabe konzentrieren, eine numerisch korrekte Kernfunktion zu implementieren. Auswirkungen auf den Rest der Bibliothek sind durch die Design-Pattern ausgeschlossen.

**Neuer Integrator**

```
1  class Integrator
2  {
3   public:
4
5   /**
6    * Constructor to initialize the data members.
7    * The idCount indicates, how much variables the integrator
8    * needs to integrate one quantity (the concrete integrator
9    * knows this and sets this constant data member via
10   * the constructor).
11   */
12  Integrator(unsigned long group, Mediator* mediator, size_t order);
13  /** The a virtual destructor for this abstract class */
14  virtual ~Integrator();
15  /**
16   * The integrator must know the IdStore,
17   * but the IdStore can't be created
18   * without an existing integrator.
19   * The SubInitializer resolves this dependency by
20   * (1) creating an integrator
21   * (2) creating an IdStore with the data from the integrator
22   * (3) calling this member function to announce the
23   *     IdStore with the integrator.
24  void announce(IdStore* pids);
25  /**
26   * Integrates all particles in the list.
27   * If the integration is ready, we return true.
28   * If the integration is not finished and we need a
29   * new right hand side first, we return false.
30   */
31  bool integrate(Time deltaT, ParticleContainer* particles);
32
33 protected:
34
35  /**
36   * The mediator of the subgroup.
37   */
38  Mediator* mediator;
39  /**
40   * The IdStore with the indices in the particle quantity list
41   */
42  IdStore* ids;
43 };
```

Analog zur Implementierung der Kernfunktion muss bei einem Integrator von der Basisklasse `Integrator` abgeleitet werden. Die ist dem Integrator-Pattern zu entnehmen (Anhang A.1.1 auf Seite 152). Der konkrete Integrator muss die Operation

`integrate()` überschreiben. Dabei wird z.B. ein Runge-Kutta-Integrator wie in [47] beschrieben implementiert. Zu bemerken ist, dass der Integrator notwendig eine Referenz auf den Mediator zum Subsimulationsgebiet haben muss (siehe Abb. 5.10 auf Seite 119).

**Neuer SPH-Term: Rho**

```
1  // ***** project includes *****
2  #include "Rho.hpp"
3  #include "BaseParticle.hpp"
4  /**
5   * Calculate the (mass)density.
6   */
7  class Rho : public Quantity
8  {
9   public :
10
11  /**
12   * Constructor to set the indices for the access
13   * into the particles quantities.
14   * The first parameter is the index for the sum to calculate,
15   * the rest is needed to calculate one term of the sum.
16   */
17  Rho(size_t rho, size_t mass, bool requiresUpdate = false);
18  /** The member to reset the quantity to 0. */
19  virtual void set(ParticleContainer*) const;
20  /** The formula to sum up the quantity. */
21  virtual void sum(Interactions* i) const;
22 }
23 //
24 // Implementation of class Rho
25 //
26 Rho::Rho(size_t rho, size_t mass, bool requiresUpdate = false)
27     : Quantity(rho, requiresUpdate), mass(mass)
28 {
29 }
30 void
31 Rho::set(ParticleContainer* pc) const
32 {
33     // for all particles do...
34     for (ParticleContainer::iterator p = pc->begin();
35          p < pc->end(); ++p)
36     {
37         p->scalarQuantity[lhs] = 0.0;
38     }
39 }
40 void
```

```
41  Rho::sum(Interactions* i) const
42  {
43      for (Interactions::iterator ia = i->begin(); ia < i->end(); ++ia)
44      {
45        // sum up the interacting particle i
46        ia->i->scalarQuantity[lhs] +=
47                ia->j->scalarQuantity[mass] * ia->w;
48        // sum up the interacting particle j, but only,
49        //if it is no ghost or proxy!
50        if (!ia->jIsFake) {
51          ia->j->scalarQuantity[lhs] +=
52                ia->i->scalarQuantity[mass] * ia->w;
53        }
54      }
55  }
```

Die Klasse `Rho` implementiert den SPH-Term der Dichteberechnung. Zu beachten ist hier, dass im Konstruktor von `Rho` die Eigenschaft `requiresUpdate` auf `false` gesetzt wird. Zur Dichteberechnung sind alle Daten lokal vorhanden. Rho muss, wie alle SPH-Terme, von der Klasse `Quantity` abgeleitet werden (siehe Anhang B). Nachdem der neue SPH-Term implementiert ist, muss dieser noch über den QuantityBuilder dem Rest des Systems bekannt gegeben werden. Im Anhang B ist der QuantityBuilder mit den bereits implementierten SPH-Termen als Quelltext abgebildet.

Auch hier konzentriert sich der Entwickler eines neuen SPH-Terms auf die eigentliche Aufgabe der Implementierung des SPH-Terms. Zur eigentlichen Klasse des neuen SPH-Terms muss lediglich der `QuantityBuilder` als zusätzliche Klasse verändert werden. Durch die Methode `useRhoDirect()` soll die Veränderung im `QuantityBuilder` kurz erläutert werden.

### Implementierung des Quantity-Builders

```
1   void
2   QuantityBuilder::useRhoDirect(bool withUpdate = false)
3   {
4       string rhoFluctuation = map->getValue("DensityFluctuation");
5       Quantity* q = 0;
6
7       if (rhoFluctuation == "Continuous")
8       {
9           q = new Rho(ids.rho, ids.m, withUpdate);
10      }
11      else if (rhoFluctuation == "Discontinuous")
12      {
13          q = new RhoDiscontinuous(ids.rho, ids.m, withUpdate);
14      }
```

```
15
16        quantityList->push_back(q);
17    }
```

Der `QuantityBuilder` liest aus der initialisierten Parameter-Map den vom Benutzer eingestellten Konfigurationswert für den SPH-Term `Rho` und erstellt abhängig davon ein neues Objekt der Klasse `Rho`. Dieser neue SPH-Term wird dann in die Liste der SPH-Terme (`QuantityList`) eingehängt.

### 5.4.3 Anpassung von SPH an Parallelrechner

Im Abschnitt 5.4.2 wurde dargelegt, wie es möglich ist, neue SPH-Terme zu implementieren. Dabei musste in keinem Fall auf das spätere Ausführungsmodell — ob seriell oder parallel — eingegangen werden. Im Folgenden soll anhand der Implementierung der Kommunikatoren für Rechner mit verteiltem Speicher skizziert werden, wie mit den SPH-Design-Pattern ein SPH-Programm auf verschiedenen Rechnertypen ausgeführt werden kann.

Aus dem Design-Pattern der Kommunikatoren (Abb. 5.9) ist ersichtlich, dass zwei Klassen implementiert werden müssen, um ein SPH-Programm auf einem Rechner mit verteiltem Speicher ausführen zu können: der `DistribMemPrincipalCommunicator` und der `DistribMemSubCommunicator`. Abhängig davon, welcher Rechner eingesetzt wird, muss in diesen Klassen die systemabhängige Kommunikationsbibliothek verwendet werden. Im hier aufgeführten Beispiel werden die Methoden `announceComm()` und `receiveParticles()` dargestellt. Mit der Methode `announceComm()` stellt der Sub-Kommunikator die Verbindung zu allen anderen Sub-Kommunikatoren her. Da jedes Simulationsgebiet einen Sub-Kommunikator hat, ist dadurch die Verbindung zwischen den Simulationsgebieten hergestellt. Die Methode `receiveParticles()` demonstriert den dann folgenden Datenaustausch, um SPH-Teilchen in das lokale Subsimulationsgebiet einzuladen.

Das hier gezeigte Beispiel verwendet TPO++ als Kommunikationsprimitive.

#### Implementierung eines Kommunikators

```
1   class DistribMemSubCommunicator : public BaseSubCommunicator
2   {
3     public:
4
5     void
6     DistribMemSubCommunicator::announceComm(int myrank,
7                                             TPO::Communicator* subcomm)
8     {
```

```
 9       rank = TPO::CommWorld.rank();
10       grouprank    = myrank;
11       commSubgroup = subcomm;
12     }
13
14     void
15     DistribMemSubCommunicator::receiveParticles(ParticleContainer*)
16     {
17       ParticleContainer particles;
18       TPO::net_back_insert_iterator<ParticleContainer> pit(particles);
19       TPO::CommWorld.recv(pit, 0);
20
21       mediator->receiveParticles(&particles);
22     }
23   }
```

Die Methode `announceComm()` verwendet die Kommunikationsprimitiven von TPO++, um sich bei allen anderen Kommunikatoren anzumelden.

Über die Methode `receiveParticles` wird dem lokalen Kommunikator ein Teilchencontainer zugesendet. Hier wird über die Klasse `Mediator` aus dem Mediator-Pattern nun der empfangene Teilchencontainer an das Subsimulationsgebiet übergeben. Die Klasse `Mediator` entkoppelt die systemabhängigen Kommunikationsprimitiven von dem Rest des SPH-Programms. Dies ist in Abb. 5.10 als UML-Diagramm dargestellt. Das Beispiel zeigt keine vollständige Implementierung, verdeutlicht aber die klare Entkopplung der Programmteile, die für die Parallelisierung verantwortlich sind, von den Programmteilen, die die eigentliche SPH-Berechnung durchführen.

## 5.5 Zwischenergebnis — Nutzen der Design-Pattern für SPH

Durch die in der vorliegenden Arbeit entwickelten SPH-Design-Pattern ist die Aufgabenstellung der Implementierung eines SPH-Verfahrens von der Aufgabe der Parallelisierung eines SPH-Verfahrens vollständig entkoppelt. Die im Abschnitt 5.2 diskutierten Design-Pattern geben eine Vorgehensweise zur Implementierung von SPH-Verfahren an. Die Aufgabe, ein SPH-Verfahren zu parallelisieren, wird lokal durch Implementierung von Kommunikatoren und Simulationsklassen in den im Abschnitt 5.3 dargestellten Design-Pattern gelöst.

Die Design-Pattern zur Parallelisierung von SPH-Verfahren aus Abschnitt 5.3 ermöglichen die Anpassung eines SPH-Programms an verschiedene Typen von Parallelrechnern, ohne dass die Implementierung des eigentlichen SPH-Verfahrens

verändert werden muss. Die Implementierung spezieller Kommunikator- und Simulationsklassen sind dafür verantwortlich, ein SPH-Verfahren effizient auf Rechner mit verteiltem oder Rechner mit gemeinsamem Hauptspeicher anzupassen.

Da verschiedenen Aufgabenstellungen klar getrennte Design-Pattern zugewiesen wurden und dadurch die beteiligten Klassen voneinander entkoppelt wurden, sind einzelne Lösungen gut wiederverwendbar. Beispiele dafür sind die Integratoren, die Kernfunktionen, die einzelnen SPH-Terme, die Kommunikatoren und Subsimulationsklassen.

Durch die vorgestellten Design-Pattern ist eine Rollenverteilung bei der Implementierung eines objektorientierten parallelen SPH-Verfahrens vorgegeben. Der *Physiker* kann sich auf die Aufgabenstellung der Entwicklung, Implementierung und Test von SPH-Termen und SPH-Verfahren konzentrieren. Der *Mathematiker* kann geeignete Integratoren und andere mathematische Hilfsmittel an einer klar vorgegebenen Stelle implementieren. Der *Informatiker* konzentriert sich auf die effiziente Abbildung des Programms auf verschiedene Hardwareplattformen.

Durch die Design-Pattern entsteht ein klar strukturiertes Rahmenprogramm zur Zusammenarbeit verschiedener Experten an der Implementierung von objektorientierten parallelen SPH-Verfahren.

Im Kapitel 6 werden verschiedene mit den hier vorgestellten Vorgehensweisen implementierte Teilchensimulationsverfahren quantitativ untersucht.

# Kapitel 6

# Anwendungen

Im Kapitel 5 wurden alle für die vorliegende Arbeit formulierten Zielsetzungen erreicht, es muss nur noch die Anwendung der objektorientierten SPH-Design-Pattern auf realistische SPH-Verfahren untersucht werden. In diesem Kapitel wird die letzte Zielsetzung angegangen und die Effizienz der Parallelisierung mit objektorientierten Design-Pattern untersucht.

Dazu werden die folgenden Fragestellungen bearbeitet:

1. Effizienz eines objektorientierten, parallelen SPH-Programms.

   Ein SPH-Verfahren zur Simulation von Dieseleinspritzung wird auf einem Rechner mit verteiltem Speicher implementiert und vermessen. Diese Implementierung kann mit den Referenzmessungen aus Kapitel 4 verglichen werden.

2. Vergleich von objektorientierten und prozeduralen Implementierungen.

   Es wird eine serielle SPH-Anwendung, die mit den Methoden der vorliegenden Arbeit implementiert wurde, genauer untersucht, sodass eine Aussage über den Laufzeitverlust von objektorientierten Programmen gegenüber prozeduralen Programmiersprachen für Teilchensimulationsverfahren getroffen werden kann.

3. Anwendung auf andere Teilchensimulationsverfahren.

   Es wird anhand einer Monte-Carlo-Simulation kurz darauf eingegangen, wie die Methoden der vorliegenden Arbeit auf andere Teilchensimulationsverfahren angewendet werden können.

## 6.1 Referenzimplementierungen

Im Zusammenhang mit den Voruntersuchungen über die Parallelisierbarkeit von SPH-Verfahren im Kapitel 4 wurden bereits zwei Testimplementierungen von SPH-

Verfahren diskutiert.

#### NEC SX-4

Auf Rechnern mit gemeinsamem Speicher, wie der NEC SX-4, konnten durch geeignete Wahl der Algorithmen und Datenstrukturen parallele Effizienzen von bis zu 90% erreicht werden (siehe Abschnitt 4.4.2 auf Seite 76). Architekturen mit gemeinsamem Speicher skalieren aber nur mit relativ wenigen CPU, sodass Architekturen mit verteiltem Speicher interessanter erscheinen.

#### Cray T3E

Unter der Berücksichtigung der Architektur der Cray T3E konnten SPH-Verfahren mit einer parallelen Effizienz von bis zu 65% implementiert werden (siehe Abschnitt 4.5.3 auf Seite 92). Besonders dieses Ergebnis stellt einen guten Referenzpunkt für die Untersuchung der Effizienz der Methoden der vorliegenden Arbeit dar.

## 6.2 Dieseleinspritzung

Als Anwendung der in der vorliegenden Arbeit entwickelten Methoden zur Parallelisierung von SPH-Verfahren mit objektorientierten Design-Pattern wird ein komplexes SPH-Verfahren implementiert.

Das SPH-Verfahren wird hier verwendet, um Aufschlüsse über den Vorgang des Zerstäubens von Diesel beim Einspritzen in eine Brennkammer zu bekommen. Interessant ist dies, da wegen der hohen optischen Dichte von Diesel eine Messung des Vorgangs in den ersten $\mu$-Metern nach Austritt aus der Einspritzdüse nicht möglich ist. Der weitere Verlauf des Zerstäubens ist messbar und bekannt. Somit ist ein Vergleich der Simulationsergebnisse mit Messungen möglich. Interessant ist hier allerdings nicht der Vergleich mit Messdaten aus der Automobilindustrie, sondern der Vergleich mit einem Referenzprogramm, um Aufschlüsse über die parallele Effizienz der Implementierung mit Design-Pattern zu bekommen.

### 6.2.1 SPH-Simulation der Dieseleinspritzung mit `sph2000`

Zunächst sollen die Simulationsergebnisse eines Programmlaufs mit 13.000 SPH-Teilchen dargestellt werden. In den Abbildungen 6.1 auf der nächsten Seite und 6.2 auf Seite 134 kann man die Bewegung der Stützstellen (SPH-Teilchen) mitverfolgen. Dabei werden Dieselteilchen — in den Abbildungen als kleine Punkte dargestellt — mit einer Geschwindigkeit von $400\frac{m}{s}$ eingespritzt, sodass diese die Luftteilchen — in

**Teilchenverteilung bei Dieseleinspritzung** I

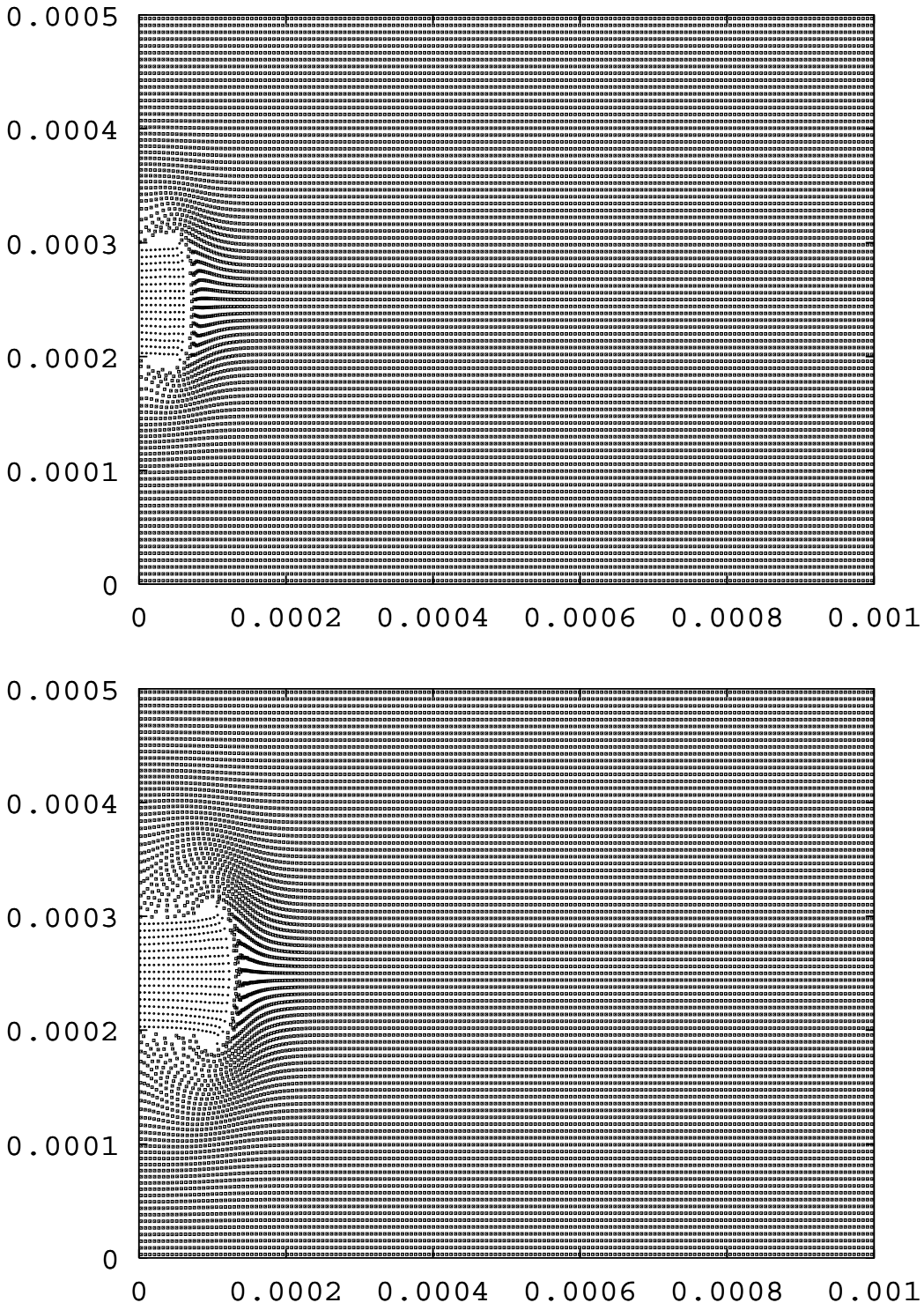



Abbildung 6.1: Dieseleinspritzung zu den Zeiten 150 und 300.

**Teilchenverteilung bei Dieseleinspritzung** II

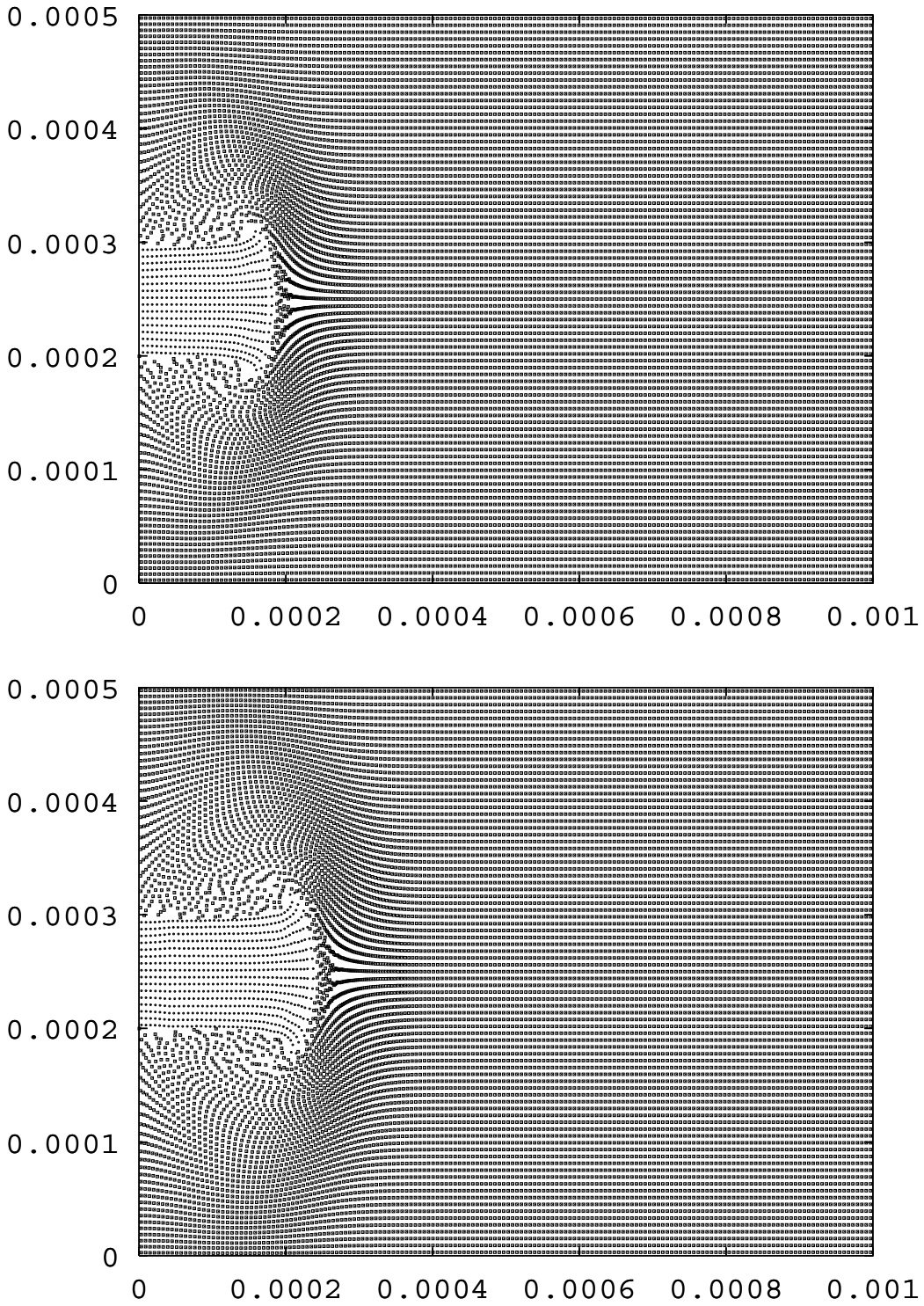



Abbildung 6.2: Dieseleinspritzung zu den Zeiten 450 und 600.

den Abbildungen als größere Quadrate dargestellt — wegdrücken. Diese Abbildungen geben einen Eindruck, wie die Simulationsergebnisse einer SPH-Simulation zur Dieseleinspritzung aussehen.

### 6.2.2 Laufzeitverhalten von `sph2000` auf Kepler

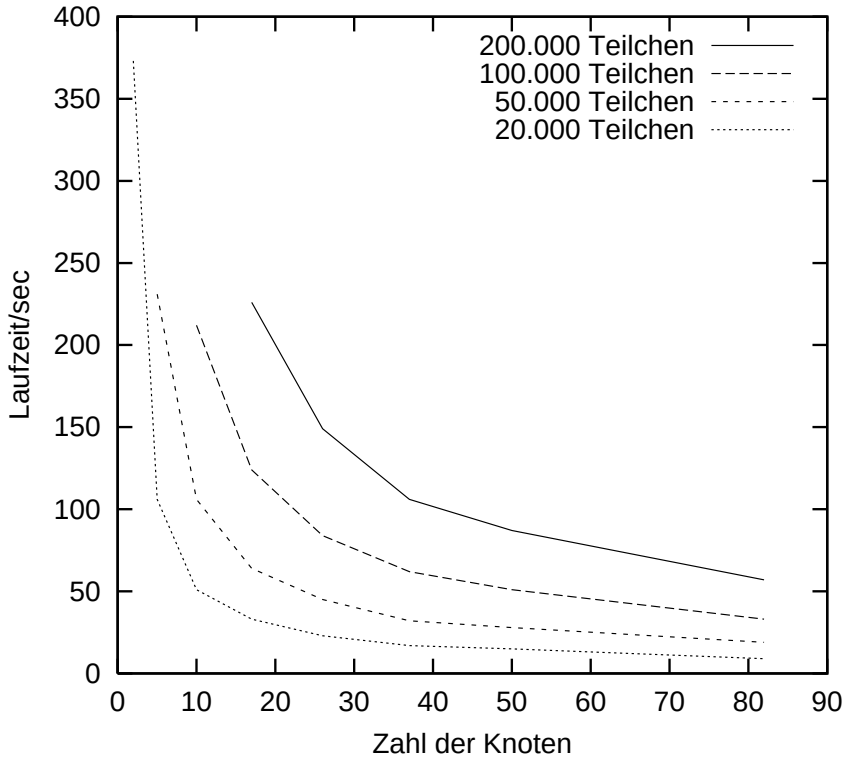



Abbildung 6.3: Laufzeitverhalten von `sph2000` auf Kepler mit 200.000, 100.000, 50.000 und 20.000 Teilchen.

Die Laufzeitmessungen von `sph2000` wurden auf dem Kepler-Cluster (siehe Abschnitt 2.2.1) ausgeführt. Das Kepler-Cluster ist eine Maschine mit verteiltem Speicher. Als systemnahe Kommunikationsbibliothek zur Implementierung des Kommunikator-Pattern wurde TPO++ verwendet (siehe Abschnitt 5.4.3 auf Seite 127).

Von den 96 Knoten des Kepler-Clusters konnten bis zu 82 Knoten zur Ausführung des SPH-Programms verwendet werden. Der Simulationsraum der SPH-Simulation wurde in dem Programm in gleichmäßige Quadrate aufgeteilt. Jedes Simulationsgebiet wird einem Knoten zugewiesen. Damit ist die Anzahl der verwendeten Knoten für die Simulationsrechnung eine Quadratzahl. Da der Principal-Kommunikator

| | Teilchenzahl in Tausend | | | |
|---|---|---|---|---|
| | 200 | 100 | 50 | 20 |
| Knoten | Laufzeit in sec | | | |
| 2 | N.A. | N.A. | N.A. | 373,285 |
| 5 | N.A. | N.A. | 231,322 | 106,655 |
| 10 | N.A. | 212,150 | 106,334 | 51,173 |
| 17 | 226,796 | 124,381 | 64,351 | 33,450 |
| 26 | 149,431 | 84,420 | 45,807 | 23,198 |
| 37 | 106,172 | 62,964 | 32,488 | 17,750 |
| 50 | 87,288 | 51,109 | 28,173 | 15,214 |
| 82 | 57,226 | 33,243 | 19,303 | 9,104 |

Tabelle 6.1: Laufzeitmessung von `sph2000` auf Kepler.

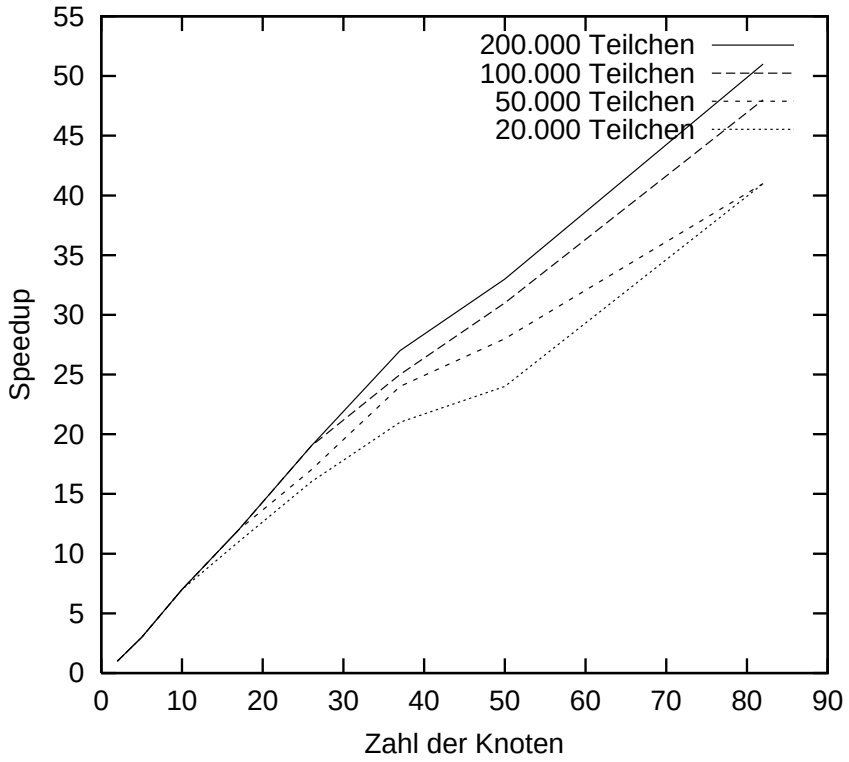



Abbildung 6.4: Speedup von `sph2000` auf Kepler mit 200.000, 100.000, 50.000 und 20.000 Teilchen auf bis zu 82 Knoten.

ebenfalls einen Knoten belegt, ist die Gesamtanzahl der Knoten auf dem Kepler-Cluster für einen Simulationslauf immer $N^2 + 1$ mit $N = (1, 2, 3, \ldots)$.

In der Tabelle 6.1 sind die gemessenen Laufzeiten des Simulationsprogramms dargestellt. Gemessen wurde die Laufzeit zwischen der Initialisierung — d.h. dem

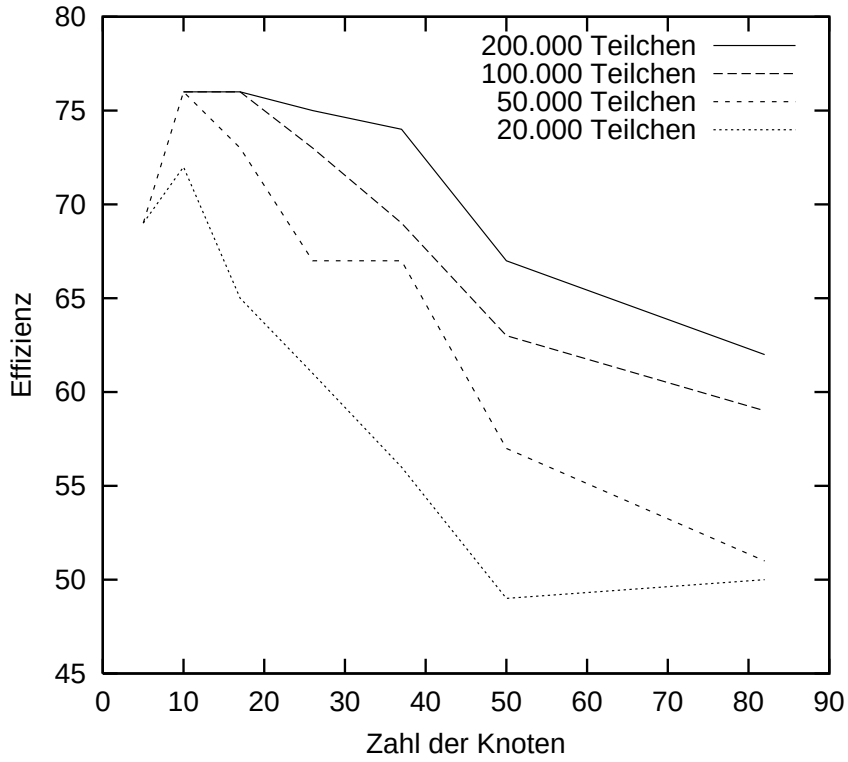



Abbildung 6.5: Parallele Effizienz von `sph2000` auf Kepler mit 200.000, 100.000, 50.000 und 20.000 Teilchen auf bis zu 82 Knoten.

Einlesen der ersten Teilchendaten — und der Datenausgabe am Programmende. Die Eingabe- und Ausgabefunktionen sind nicht parallelisiert und stellen bei den Datenmengen einen deutlichen Laufzeitoverhead dar. Bei 200.000 Teilchen braucht eine Datenausgabe bis zu 30 Sekunden. Dieser Overhead ist aber bei seriellen und parallelen Programmen gleich und wurde aus der Messung herausgenommen.

Zur Messung wurde eine vollständige Dieseleinspritzung simuliert, allerdings nur über einen verkürzten Simulationszeitraum. Für eine Simulation über einen längeren Zeitraum steigt die Programmlaufzeit linear mit dem Integrationszeitraum an, was für die Untersuchung der parallelen Effizienz nicht interessant ist.

Die Messung wurde mit 20.000, 50.000, 100.000 und 200.000 Teilchen durchgeführt. In Abb. 6.3 ist das Laufzeitverhalten der Simulation für die unterschiedlichen Teilchenzahlen graphisch dargestellt. Dabei ist die Laufzeitveränderung bei der Variation der Anzahl der Knoten aufgezeichnet.

Aus den gemessenen Laufzeiten lassen sich nun der Speedup und die parallele Effizienz errechnen (siehe Abschnitt 2.2.3). In Abb. 6.5 ist die parallele Effizienz der

Messung dargestellt. Die Speedup-Kurven für die Simulationsläufe mit den verschiedenen Teilchenzahlen zeigt Abb. 6.4.

### 6.2.3 Diskussion

Die parallelen Effizienzen der SPH-Simulation auf Kepler nehmen für größer werdende Teilchenzahlen zu. Dieses Verhalten konnte schon bei der Referenzimplementierung auf der Cray T3E beobachtet werden. Wenn die Teilchenzahlen pro Knoten zu klein werden, so überwiegt der Kommunikationsanteil. Für große Teilchenzahlen ist die parallele Effizienz auf Kepler bei 82 Knoten noch bei über 62%. Dies steht in einem guten Vergleich mit den Ergebnissen der Referenzimplementierung auf der Cray T3E.

Das mit den Methoden der vorliegenden Arbeit parallelisierte SPH-Simulationsverfahren weist also auch im Vergleich mit anderen Parallelisierungsmethoden eine gute parallele Effizienz auf.

## 6.3 Kataklysmische Variable

Im Folgenden soll noch auf die Problematik des Vergleichs von objektorientierten und prozeduralen Programmen eingegangen werden. Besonders für technisch-wissenschaftliche Anwendungen wird den objektorientierten Programmiersprachen nachgesagt, weniger effizient zu sein. Für den Fall der SPH-Programme wird dies anhand eines mit den SPH-Pattern der vorliegenden Arbeit implementierten SPH-Verfahrens untersucht.

Unter kataklysmische Variable (*engl.: cataclysmic variable*) — kurz: CV — versteht man veränderliche Doppelsternsysteme, die durch das Roche-Modell (siehe Abbildung 6.6) beschrieben werden. Dabei ist der Primärstern ein Weißer Zwerg und der Sekundärstern ein Hauptreihenstern, der sein Roche-Volumen vollständig ausfüllt und daher Masse an den Primärstern verliert.

Die vom Sekundärstern abgegebene Masse wird vom Primärstern akkretiert. Es bildet sich eine Akkretionsscheibe aus. Durch das Auftreffen der Masse des Sekundärsterns auf die Akkretionsscheibe bildet sich ein Leuchtfleck, der beobachtet werden kann.

Kataklysmische Variable haben eine hohe Umlauffrequenz. Im Falle der kataklysmischen Variable *OYCar* ist die Bahnperiode ca. 1,5 Stunden. Der mitrotierende Leuchtfleck wird auf der Erde als pulsierendes Signal aufgenommen.

Das Simulationsprogramm SPH++ simuliert ca. 100 Bahnperioden der kataklysmischen Variable *OYCar*; das entspricht einer Simulation von ca. 6,3 Tagen.

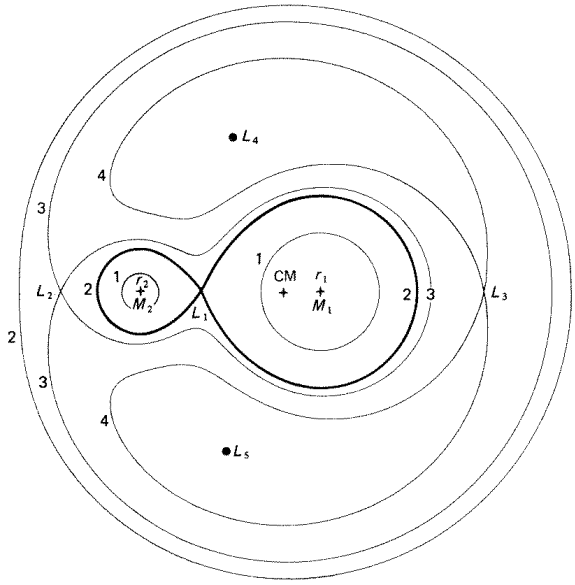



Abbildung 6.6: Roche-Potential: Äquipotentiallinien, Schnitt durch die Bahnebene für das Massenverhältnis q=$M_1/M_2$=5. Die dicke Linie stellt die Roche-Lobes dar.

### 6.3.1 SPH-Simulation von CV *OYCar* mit SPH++

SPH++ ist ein mit den Methoden aus Kapitel 5 implementiertes Programm zur Simulation von akkretierenden Doppelsternsystemen.

In Abbildung 6.7 auf der nächsten Seite sind Simulationsergebnisse aus der Simulation der CV *OYCar* mit dem Programm SPH++ graphisch dargestellt. Im direkten Vergleich mit der Abbildung 6.8 auf Seite 141 — in der die gleiche Simulation mit einem Referenz-Simulationsprogramm abgebildet ist — ist zu erkennen, dass das SPH++-Programm die gleichen Ergebnisse liefert. Eine genauere Untersuchung zeigt die Korrektheit des objektorientierten Simulationsprogramms SPH++.

Für die Untersuchungen der vorliegenden Arbeit ist das genaue Vorgehen, wie zwei Simulationsergebnisse miteinander verglichen werden können, nicht weiter interessant. Es soll an dieser Stelle lediglich vermerkt werden, dass SPH++ als Produktionscode für SPH-Simulationen tauglich ist. Im Folgenden wird das hier interessante Laufzeitverhalten des Simulationsprogramms SPH++ und des Referenzprogramms (C-Code) diskutiert.

**Ergebnis der Simulation einer Akkretionsscheibe mit SPH++**

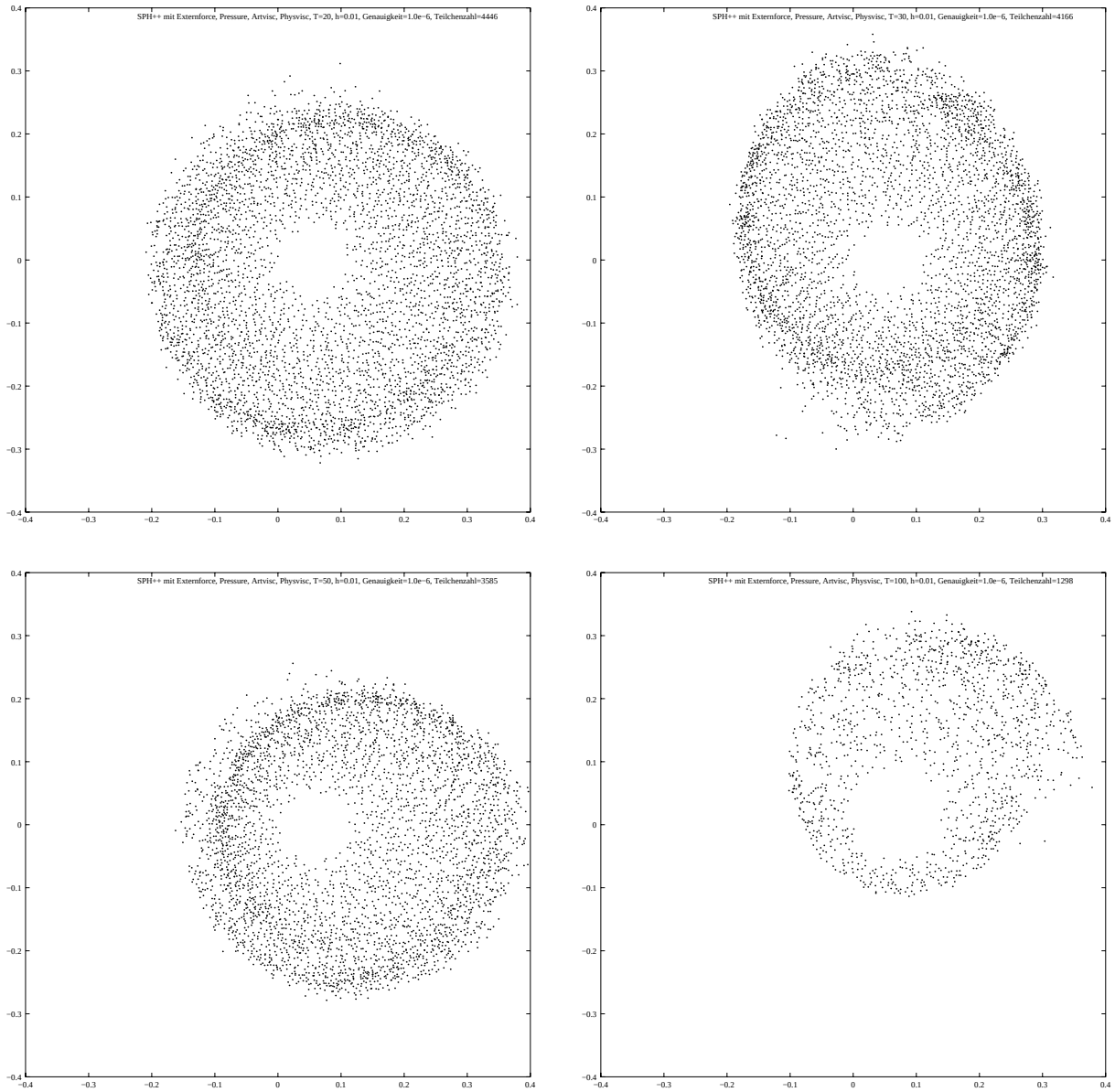



Abbildung 6.7: Mit SPH++ berechnete Akkretionsscheiben unter der Einwirkung der Gravitations-, der Druck-, der künstlichen und der physikalischen Viskositätskraft.

| Programm | Schritte | Zeit [sec] | Zeit/Zeitschritt |
|---|---|---|---|
| SPH++ | 12 | 86,58 | 7,21 |
| C-Prog. | 12 | 37,65 | 3,14 |

Tabelle 6.2: Laufzeitvergleich der Simulationen von Abbildungen 6.7 und 6.8.

**Vergleichssimulation einer Akkretionsscheibe (C-Programm)**

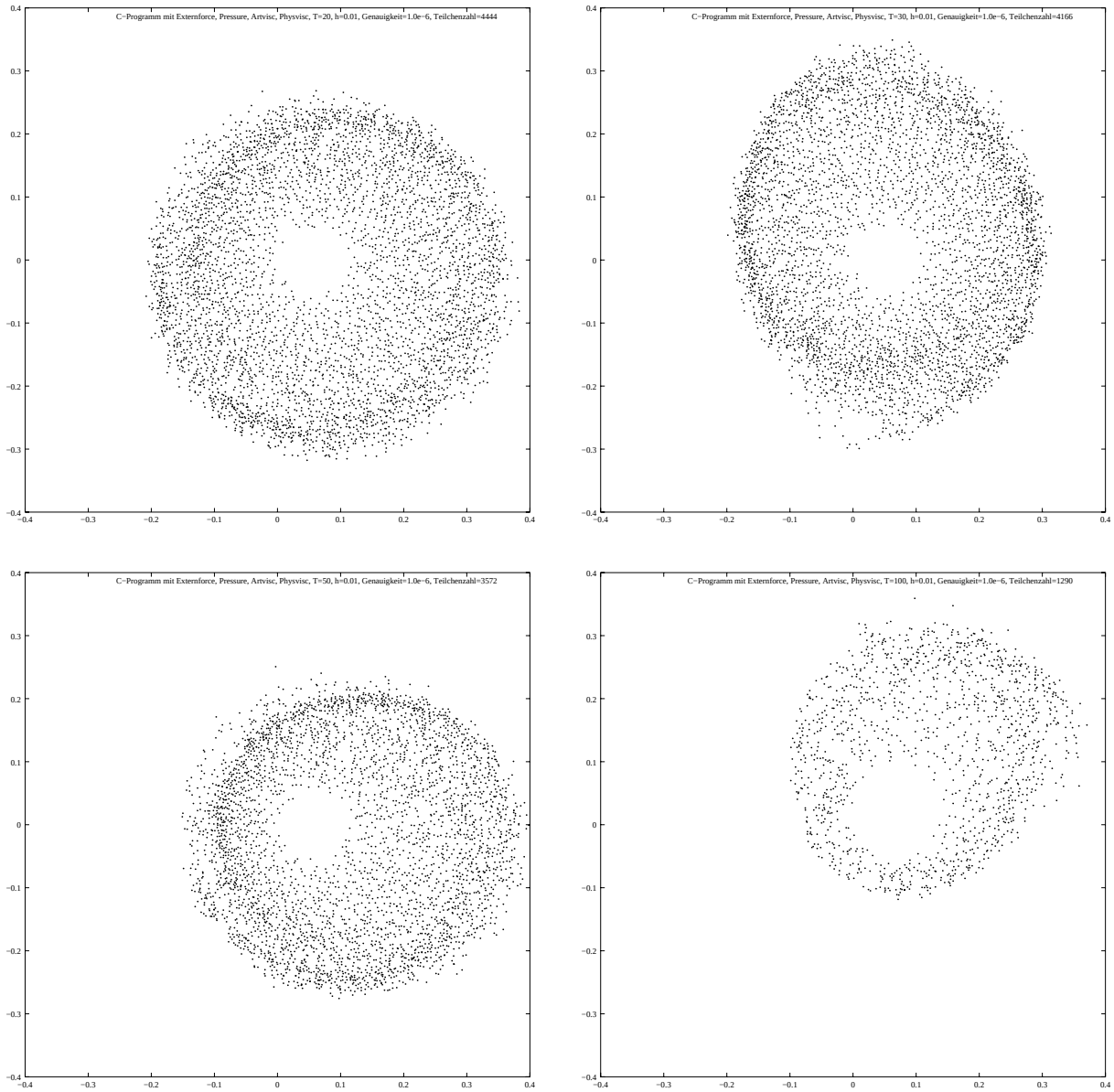



Abbildung 6.8: Die mit dem C-Programm berechneten Akkretionsscheiben unter der Einwirkung der Gravitations-, der Druck-, der künstlichen und der physikalischen Viskositätskraft.

### 6.3.2 Laufzeitverhalten der Simulation mit SPH++

Interessant ist die Veränderung des Laufzeitverhaltens eines seriellen Simulationsprogramms mit einem Programmierparadigma wie der Sprache `C` (oder `FORTRAN77`) gegenüber einem seriellen Programm mit den in Kapitel 5 vorgestellten Design-Pattern.

| Abb. | Programm | Schritte | Zeit [sec] | Zeit/Zeitschritt |
|---|---|---|---|---|
| 6.7 | SPH++ | 29.341 | 123.427,78 | 4,21 |
| 6.8 | C-Prog. | 29.398 | 47.150,56 | 1,60 |

Tabelle 6.3: Gesamtlaufzeit der Simulationen von Abbildungen 6.7 und 6.8.

In Tabelle 6.3 finden sich die Gesamtlaufzeiten der in den Abbildungen 6.7 und 6.8 dargestellten Simulationsläufe.

Es ist zu sehen, dass das SPH++-Programm für den vollständigen Simulationslauf 29.341 Zeitschritte integriert hat, gegenüber 29.398 Zeitschritten des Referenzprogramms in C. Dabei hat das SPH++-Programm insgesamt eine Laufzeit von 123.427,78 Sekunden benötigt. Das C-Programm hat dagegen 47.150,56 Sekunden für den Simulationslauf gebraucht.

Daraus ergibt sich eine Laufzeiterhöhung um den Faktor 2,617 des SPH++-Programms gegenüber einem Referenzprogramm in C. Die unterschiedliche Anzahl von Zeitintegrationsschritten ist durch das chaotische Verhalten von SPH-Simulationen zu erklären und kann unter der Voraussetzung, dass das Programm richtig arbeitet, vernachlässigt werden.

In Tabelle 6.2 sind die Zeiten pro Integrationsschritt in Sekunden für einen ähnlichen Simulationslauf dargestellt. Auch hier ist das SPH++-Programm um einen Faktor 2,296 langsamer als das C-Programm.

Die Laufzeiten hängen von den anfänglichen Teilchenverteilungen, dem Wechselwirkungsradius und nicht zuletzt von der verwendeten CPU ab. In Tabelle 6.4 sind verschiedene Testläufe mit SPH++ im Vergleich mit dem Referenzprogramm (`C`) dargestellt. In der letzten Spalte ist für das SPH++-Programm der Faktor notiert, um den das SPH++-Programm langsamer ist als das Referenzprogramm.

Bei den Laufzeittests wurde ein mittlerer Faktor von 3,038 gemessen, den das SPH++-Programm langsamer ist. Außerdem bemerkenswert ist, dass die Laufzeitfaktoren von dem verwendeten Rechnertyp (CPU) abhängen. Diese Schwankungen sind aber auf die Optimierungsmöglichkeiten der verwendeten Compiler (GNU C++) zurückzuführen.

### 6.3.3 Diskussion

Die Untersuchungen zeigen, dass eine Implementierung einer SPH-Simulation der kataklysmischen Variable *OYCar* mit den objektorientierten Design-Pattern aus Kapitel 5 um einen Faktor 2,617 langsamer ist als ein Referenzprogramm in `C`.

| Programm | h | Teil.zahl[a] | WWP.[b] | Steps | Arch.[c] | Zeit | Zeit/Step | LV[d] |
|---|---|---|---|---|---|---|---|---|
| C-Prog. | 0,005 | 5406 (-3) | 2369 | 21 | Sparc | 40,57 | 2,03 | – |
| “ | “ | “ | “ | “ | Celeron | 35,85 | 1,79 | – |
| “ | “ | “ | “ | “ | Pentium | 20,25 | 1,01 | – |
| SPH++ | “ | “ | “ | “ | Sparc | 180,79 | 9,04 | 4,46 |
| “ | “ | “ | “ | “ | Celeron | 110,54 | 5,53 | 3,08 |
| “ | “ | “ | “ | “ | Pentium | 68,28 | 3,41 | 3,37 |
| C-Prog. | 0,01 | 5406 (-4) | 16433 | 12 | Sparc | 42,26 | 3,52 | – |
| “ | “ | “ | “ | “ | Celeron | 37,65 | 3,14 | – |
| “ | “ | “ | “ | “ | Pentium | 22,83 | 1,90 | – |
| SPH++ | “ | “ | “ | “ | Sparc | 141,87 | 11,82 | 3,36 |
| “ | “ | “ | “ | “ | Celeron | 86,58 | 7,21 | 2,30 |
| “ | “ | “ | “ | “ | Pentium | 47,28 | 3,94 | 2,07 |
| C-Prog. | 0,02 | 5406 (-9) | 74090 | 9 | Sparc | 99,86 | 11,10 | – |
| “ | “ | “ | “ | “ | Celeron | 92,96 | 10,33 | – |
| “ | “ | “ | “ | “ | Pentium | 56,31 | 6,26 | – |
| SPH++ | “ | “ | “ | “ | Sparc | 355,62 | 39,51 | 3,56 |
| “ | “ | “ | “ | “ | Celeron | 217,45 | 24,16 | 2,34 |
| “ | “ | “ | “ | “ | Pentium | 123,74 | 13,75 | 2,20 |
| C-Prog. | 0,01 | 1000 | 556 | 6 | Sparc | 2,25 | 0,37 | – |
| “ | “ | “ | “ | “ | Celeron | 1,55 | 0,26 | – |
| “ | “ | “ | “ | “ | Pentium | 0,77 | 0,13 | – |
| SPH++ | “ | “ | “ | “ | Sparc | 8,51 | 1,42 | 3,78 |
| “ | “ | “ | “ | “ | Celeron | 5,32 | 0,89 | 3,43 |
| “ | “ | “ | “ | “ | Pentium | 3,19 | 0,53 | 4,14 |
| C-Prog. | 0,02 | 1000 | 2576 | 6 | Sparc | 3,83 | 0,64 | – |
| “ | “ | “ | “ | “ | Celeron | 3,01 | 0,50 | – |
| “ | “ | “ | “ | “ | Pentium | 1,60 | 0,27 | – |
| SPH++ | “ | “ | “ | “ | Sparc | 10,95 | 1,82 | 2,86 |
| “ | “ | “ | “ | “ | Celeron | 6,94 | 1,16 | 2,31 |
| “ | “ | “ | “ | “ | Pentium | 3,70 | 0,62 | 2,31 |

Tabelle 6.4: Es werden die Laufzeiten des C-Programms bzw. von SPH++ in verschiedenen Konfigurationen angegeben. Die Integrationszeit ist dabei 100 sec bei einer Genauigkeit von $10^{-6}$.

[a] In der Klammer steht die Anzahl der Teilchen, die das Simulationsgebiet verlassen haben.
[b] Anzahl der Wechselwirkungspartner beim Anfang des ersten Integrationsschrittes.
[c] **Sparc:** Ultra-Sparc 10/300 MHz; **Celeron:** Intel-Celeron/416 MHz (128k L2-Cache, Solaris 8); **Pentium:** Pentium III/733 MHz (256k L2-Cache, SuSE-Linux 7.0)
[d] Laufzeitverlängerung von SPH++ gegenüber dem C-Programm.

Im Durchschnitt ist das hier untersuchte objektorientierte Programm SPH++ einen Faktor 3,038 langsamer als ein vergleichbares Programm in C.

Die Effizienz von objektorientierten Programmen hängt stark von den verwen-

deten Datenstrukturen ab. Optimierungen mit speziellen Bibliotheken wie BLITZ++ können die Effizienzlücke zwischen objektorientierten und prozeduralen Programmen schließen.

## 6.4 Strahlungstransport bei akkretierenden Neutronensternen

Als weitere Anwendung wurde eine Monte-Carlo-Simulation (siehe Abschnitt 2.1.3 auf Seite 17) zur Berechnung des Strahlungstransports bei akkretierenden Neutronensternen mit den objektorientierten Design-Pattern aus Kapitel 5 implementiert. Es wird damit kurz gezeigt, dass die Design-Pattern der vorliegenden Arbeit auf andere Teilchensimulationsverfahren angepasst werden können.

Die physikalische Problemstellung ist ähnlich des Problems aus Abschnitt 6.3. Der Massefluss des Begleitsterns eines Neutronensterns in einem Doppelsternsystem bildet eine Akkretionsscheibe um den Neutronenstern. Durch Reibungsverluste in der Akkretionsscheibe fällt das Material auf den Neutronenstern. Dabei bildet sich Plasma (freie Elektronen und Protonen) aus, das im Magnetfeld des Neutronensterns auf einen Punkt fokussiert wird. Der Auftreffpunkt des Materials auf den Neutronenstern (Hotspot) kann als Leuchtfleck (Lichtsignal im Röntgenbereich) von erdnahen Röntgensatelliten beobachtet werden. Durch die Simulation soll die beobachtete Leuchtsignalkurve reproduziert werden. Dazu wird eine Monte-Carlo-Simulation durchgeführt.

Hier kann gezeigt werden, dass sich die in dieser Arbeit entwickelte Klassenbibliothek auch für andere Teilchensimulationsverfahren eignet.

Zur Auswahl von Integratoren und Termen der Differentialgleichungen wurden Strategy-Pattern verwendet. Lediglich die Basissimulationsklassen wurden an die Situation der Monte-Carlo-Simulation angepasst.

In Abbildung 6.9 ist das einem Composite-Pattern entlehnte Grundgerüst einer Monte-Carlo-Simulation dargestellt, so wie es in dem Programm implementiert wurde. Die Simulationsgebiete entsprechen dem physikalischen Modell des Vorgangs in dem Plasma in einer Akkretionssäule (`Column`) auf einen Neutronenstern (`Neutronstar`) auftrifft. Die dort erzeugten Photonen gelangen entweder durch die Neutronensternatmosphäre (`Neutronstaratmosphere`) oder durch die Akkretionssäule (`Column`) in den Außenraum (`Outerspace`), wo sie in gerader Linie bis auf die Beobachtungsstation in Erdnähe projiziert werden.

Es soll hier nicht weiter auf die Parallelisierung von Monte-Carlo-Simulationen eingegangen werden, da die Berechnungen der einzelnen Photonen in einer Monte-

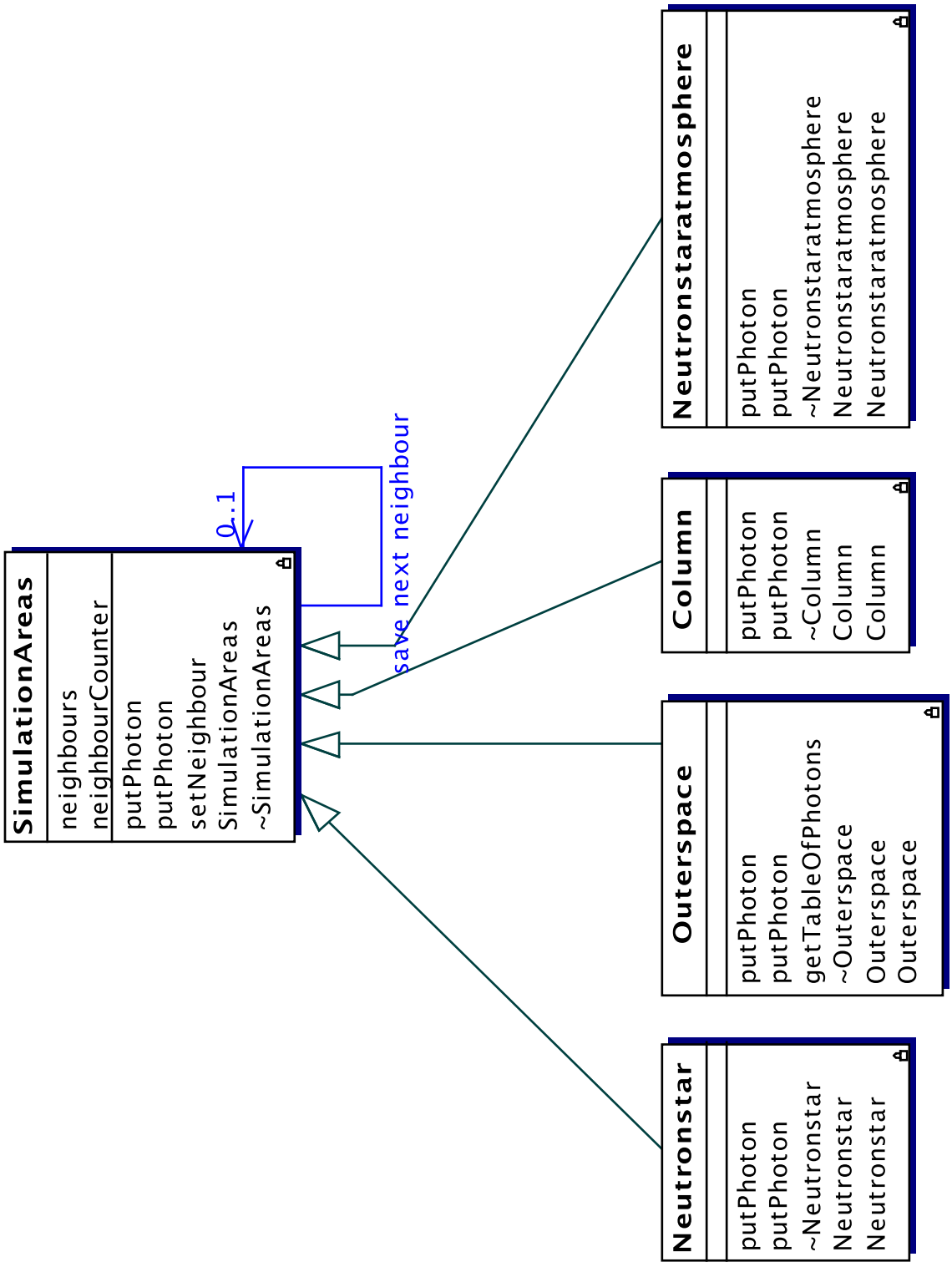



Abbildung 6.9: Simulationsgebiete einer Monte-Carlo-Simulation.

Carlo-Simulation — im Gegensatz zu den SPH-Teilchen — voneinander unabhängig sind und somit eine Parallelisierung gleichbedeutend zu einem gleichzeitigen Ausführen mehrerer Programmläufe ist. Einzig der verwendete Zufallszahlengenerator muss korrekte, unabhängige Verteilungswerte liefern.

Zur Implementierung der Monte-Carlo-Simulation mit Design-Pattern wurde von Thorsten Laib eine Diplomarbeit angefertigt [33].

# Kapitel 7

# Zusammenfassung

In der vorliegenden Arbeit wurden Teilchensimulationsverfahren anhand von Smoothed Particle Hydrodynamics auf ihre Parallelisierbarkeit hin untersucht. Als Programmiermethodik wurde die objektorientierte Programmierung mit Design-Pattern eingesetzt.

Im Bereich objektorientierter, paralleler Teilchensimulationsverfahren gibt es in der Literatur keine Implementierungen von SPH. Die veröffentlichten parallelen Implementierungen von SPH-Verfahren sind SPH-Verfahren, die auf spezielle Typen von Parallelrechnern angepasst sind. Als paralleles Programmiermodell werden in der Literatur maschinennahe Bibliotheken wie beispielsweise MPI verwendet. Ein Leistungsvergleich der verschiedenen parallelen SPH-Implementierungen ist aufwendig, da die Untersuchungen der parallelen SPH-Implementierungen meist von Physikern, mit dem Interesse an dem eigentlichen, numerischen Simulationsergebnis, durchgeführt werden. In Kapitel 3 wurde ein Überblick und ein Vergleich der aktuellen parallelen SPH-Implementierungen gegeben.

Im Kapitel 4 wurden SPH-Verfahren auf ihre Parallelisierbarkeit hin untersucht. Die Analyse zeigt die parallelisierbaren Teile eines SPH-Programms auf. Referenzimplementierungen auf der NEC SX-4 und der Cray T3E von parallelem SPH zeigen die grundsätzliche Parallelisierbarkeit. Auf NEC SX-4 konnten SPH-Verfahren mit einer parallelen Effizienz von bis zu 90% implementiert werden. Auf Cray T3E konnte mit einem zweistufigen Gebietszerlegungsverfahren auf 512 Knoten eine parallele Effizienz von 65% erzielt werden.

Im Kapitel 5 wurden die zentralen Fragestellungen der vorliegenden Arbeit angegangen und beantwortet. Durch die Entwicklung und Verwendung von objektorientierten Design-Pattern konnte die Aufgabenstellung der Implementierung eines SPH-Verfahrens von der Aufgabenstellung der Parallelisierung eines SPH-Verfahrens entkoppelt werden. Im Abschnitt 5.4 sind dafür konkrete Beispiele angegeben. Durch die Design-Pattern werden verschiedene Verantwortlichkeiten, wie die Implemen-

tierung eines Integrators oder die Implementierung von SPH-Termen, auf unterschiedliche Klassen des Systems verteilt. Dadurch sind die einzelnen implementierten Lösungen gut wiederverwendbar. Wiederverwendbarkeit lässt sich nicht quantitativ bemessen; die Struktur der Design-Pattern aus Kapitel 5 lassen aber die Unabhängigkeit der einzelnen Klassen erkennen. Durch eben diese Struktur sind auch die Rollen der einzelnen Entwickler klar verteilt. Physiker, Mathematiker und Informatiker mit den Aufgabenstellungen zur Implementierung von SPH-Termen, Integratoren und Parallelisierungskonzepten finden für jede Aufgabenstellung klar voneinander entkoppelte Design-Pattern.

Die Funktionsweise und Leistungsfähigkeit der in der vorliegenden Arbeit entwickelten Verfahren wurde anhand von aktuellen Simulationsproblemen aus der Physik in Kapitel 6 dargestellt. Dabei konnte eine parallele Implementierung der Simulation einer Dieseleinspritzung auf dem Kepler-Cluster der Universität Tübingen eine parallele Effizienz von über 62% auf 82 Knoten erreichen. Verglichen mit der Referenzimplementierung der Voruntersuchungen aus dem Kapitel 4 stellt dies die Leistungsfähigkeit der objektorientierten Design-Pattern für SPH-Verfahren klar heraus.

## 7.1 Zur Effizienz der Implementierungen

Die Untersuchungen der vorliegenden Arbeit zeigen, dass es möglich ist, mit objektorientierten Vorgehensweisen ein effizientes, massiv-paralleles Teilchensimulationsprogramm zu implementieren. Dabei wurden bisher noch keine besonderen Optimierungen verwendet. Es ist zu erwarten, dass durch die Hinzunahme bekannter Optimierungstechniken aus dem Bereich des objektorientierten Programmierens in C++, wie den aus dem objektorientierten Simulationspaket *POOMA* bekannten *expression templates*, die Effizienz weiter verbessert werden kann.

Auch der Einsatz von speziellen objektorientierten Bibliotheken, wie beispielsweise *BLITZ++*, kann zu deutlichen Performanceverbesserungen führen. Während der Entwicklung der objektorientierten Klassenbibliothek konnte durch Optimierung der internen Darstellung von Klassen, wie z.B. dem `TeilchenContainer`, die Programmlaufzeit auf ein Fünftel der Programmlaufzeit der Erstimplementierung reduziert werden. Bibliotheken, die speziell auf die Implementierung von Vektoroperationen ausgelegt sind, versprechen in diesem Bereich eine weitere Verbesserung.

Das Thema *Lastverteilung* wurde in dieser Arbeit nicht behandelt. Aus der Struktur der Design-Pattern für paralleles, objektorientiertes SPH geht aber hervor, an welcher Stelle Algorithmen zur Lastverteilung angesetzt werden müssen. Die Kommuni-

katoren und Simulationsklassen greifen auf eine weiter unten liegende Schicht zur Parallelisierung zu. In diesem Bereich setzen auch die Lastverteilungsverfahren ein. Es ist zu diskutieren, ob die Lastverteilung von den Parallelisierungs-Design-Pattern der SPH-Bibliothek oder von der weiter unten liegenden Parallelisierungsschicht implementiert werden soll. Aus dem Blickwinkel einer klaren Aufgabenverteilung und Wiederverwendbarkeit sollte eine Lastverteilung in den eigentlichen Parallelisierungsschichten ansetzen. Dann können diese Lastverteilungsverfahren auch für andere Simulationspakete wiederverwendet werden. Eine Implementierung von Lastverteilungsverfahren in den Kommunikations- und Simulations-Design-Pattern der SPH-Bibliothek ist nur für die auf dieser Bibliothek aufsetzenden Verfahren gültig. Es ist zu erwarten, dass es allgemeine Lastverteilungsverfahren gibt, die einen größeren Gültigkeitsbereich haben als nur für die in der vorliegenden Arbeit diskutierten Teilchensimulationsverfahren. Auch hier könnte durch die Verwendung eines geeigneten Design-Pattern die Möglichkeit geschaffen werden, von allgemeinen Lastverteilungsverfahren auf spezielle Anpassungen für verschiedene Simulationspakete zu konkretisieren und die grundlegenden Algorithmen wiederzuverwenden.

## 7.2 Zur Akzeptanz und Effizienz von Design-Pattern

Was in der vorliegenden Arbeit noch unberücksichtigt bleiben musste, war die Fragestellung, inwieweit die Verwendung von Design-Pattern einen Einfluss auf die Arbeitsweise der Anwendungsentwickler hat. Es ist interessant zu untersuchen, ob die in der vorliegenden Arbeit entwickelten Design-Pattern, die auf Wiederverwendbarkeit ausgelegt sind, bei mehrfacher Wiederverwendung einen Effizienzgewinn in der Entwicklung neuer SPH-Verfahren bringen.

Im Rahmen der vorliegenden Arbeit wurden sechs Diplomarbeiten ausgearbeitet. Es ist offensichtlich, dass sich aus einer solch geringen Anzahl von Arbeiten keine fundierte, quantitative Aussage über die Effizienz der Wiederverwendung ableiten lässt.

Aus den Erfahrungen der Arbeiten kann gesagt werden, dass eine gute Strukturierung durch Design-Pattern im Allgemeinen einen Vorteil gegenüber einem prozeduralen oder datenflussorientieren Vorgehen aufzeigt. Allerdings haben die Diplomanden starke, individuelle Unterschiede in ihren Arbeitsweisen gezeigt, sodass auch Diplomarbeiten, die nicht mit Design-Pattern erstellt wurden, zu einem guten Ergebnis kommen können. Es gab auch objektorientierte Arbeiten, bei denen sich durch eine ungeschickte Verwendung von Design-Pattern und Klassen eine Verschlechterung gegenüber prozedural programmierten Programmen einstellte. Gerade diese

individuellen Unterschiede der einzelnen Programmierer motivieren aber zu einer allgemein gültigen Strukturierung durch Design-Pattern.

Einen deutlichen Vorteil des Einsatzes von objektorientierten Design-Pattern gegenüber anderen Vorgehensweisen ist die klare Strukturierung der Programme, die Möglichkeit, Aufgaben auf verschiedene Projekt-Teams koordiniert verteilen zu können, und die verbesserte Wiederverwendbarkeit von Lösungen. Auch wenn einzelne Programmierer mit prozeduralen Programmiersprachen erfolgreich SPH-Verfahren implementieren können, so haben alle diese prozeduralen Implementierungen den Nachteil, dass sie für andere Programmierer schwer nachvollziehbar sind. Durch die Strukturierung von SPH-Programmen mittels der in der vorliegenden Arbeit vorgestellten Design-Pattern ist eine für alle Programmierer gleichartige Arbeitsweise vorgegeben. Damit ist auch eine Einlernphase für neue Entwickler überschaubarer.

Ganz besonders ist die in der vorliegenden Arbeit dargestellte Entkopplung von SPH-Verfahren und Parallelisierung eine Neuentwicklung für parallele SPH-Programme. Auch Kritiker des objektorientierten wissenschaftlichen Rechnens werden sich durch diesen klaren Vorteil letztlich überzeugen lassen.

# Anhang A

# Design-Pattern-Katalog für SPH-Simulationen

Im Folgenden werden die in der vorliegende Arbeit entwickelten Lösungen zur Parallelisierung von SPH-Verfahren anhand von objektorientierten Design-Pattern zusammengestellt. Eine ausführliche Diskussion der Design-Pattern findet sich im Kapitel 5.

Die Design-Pattern werden in kurzer Form anhand von folgenden Kriterien dargestellt:

*Name:* Der Name des Design-Pattern gibt bereits einen Anhaltspunkt über die Einsatzmöglichkeiten des jeweiligen Patterns.

*Problem:* beschreibt das Problemumfeld, in dem das jeweilige Design-Pattern eine Lösung bietet.

*Lösung:* Der Lösungsweg wird in der Regel durch ein UML-Klassendiagramm begleitet.

*Konsequenz:* Jede Lösung hat Vorteile und Nachteile. Unter *Konsequenz* werden die Lösungswege der Design-Pattern kurz diskutiert.

*Verwandte Pattern:* Verweise auf verwandte Pattern für das jeweilige Problemumfeld.

Im Abschnitt A.1 werden die in der vorliegenden Arbeit entwickelten Design-Pattern für SPH-Verfahren aufgelistet. Abschnitt A.2 gibt eine Zusammenfassung der zur Parallelisierung notwendigen Design-Pattern. Design-Pattern, die den Zusammenhang der Funktionsweise eines parallelen, objektorientierten SPH-Programms beschreiben, sind in Abschnitt A.3 dargestellt. Die Standard-Design-Pattern aus [19], die in der vorliegenden Arbeit Verwendung gefunden haben, sind in Abschnitt A.4 angegeben.

## A.1 SPH-Design-Pattern

### A.1.1 DGL-Integrator

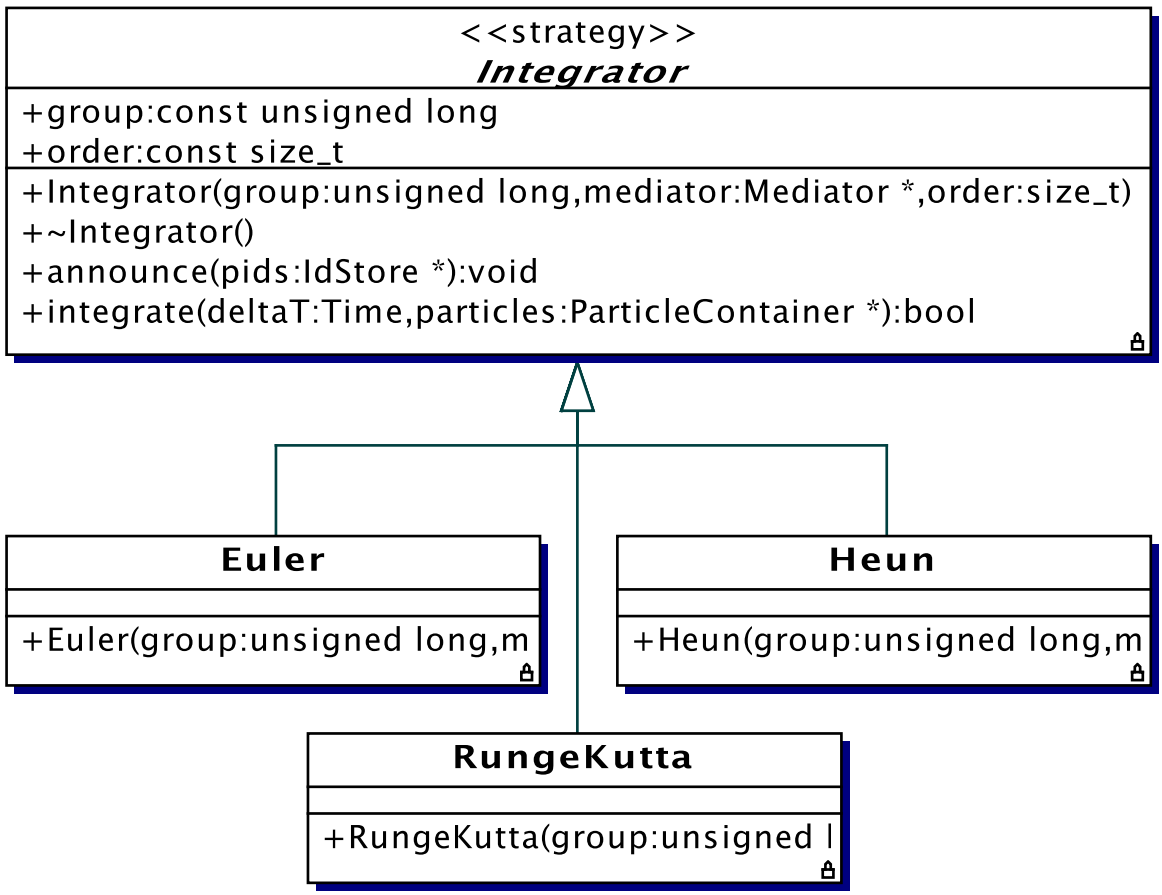



Abbildung A.1: UML-Diagramm des Design-Pattern *DGL-Integrator* zur Auswahl von Zeitintegratoren.

*Name:* **DGL-Integrator**

*Problem:* Zur Lösung einer gewöhnlichen Differentialgleichung muss ein Integrator ausgewählt werden.

*Lösung:* Die Integratoren werden in einem Strategy-Pattern angeordnet. Über die gemeinsame Schnittstelle (Typ) *Integrator* können die Integratoren für verschiedene Problemstellungen frei gewählt werden. Der Kontext des Strategy-Patterns *DGL-Integrator* ist problemabhängig. Bei SPH-Simulationen wird ein *Particle-Container* verwendet.

*Konsequenz:*

- Integratoren können unabhängig von anderen Problemteilen entwickelt und getestet werden.
- Einmal geschriebene Integratoren können für verschiedenste Problemstellungen wiederverwendet werden.
- DGL-Integrator entkoppelt die Lösung einer gewöhnlichen Differentialgleichung von anderen Aufgabenstellungen.

*Verwandte Pattern:* Strategy (Abb. A.12 auf Seite 167)

### A.1.2 SPH-Simulation

*Name:* **Simulation**

*Problem:* Ein Teilchensimulationsprogramm soll auf *seriellen* Rechnern, Rechnern mit *gemeinsamem* Hauptspeicher und Rechnern mit *verteiltem Hauptspeicher* ohne Änderungen im Programmcode laufen. Dazu muss es eine Basisstruktur geben, die diese Verantwortlichkeit übernimmt.

*Lösung:* Das *Simulation*-Pattern gibt eine Stuktur an, in der eine neue Simulation eingefügt wird. Simulationen die aus mehreren unterschiedlichen Einheiten bestehen, werden durch eine Struktur ähnlich dem *Composite*-Pattern zusammengefügt.

*Konsequenz:*

- Beim Entwurf eines SPH-Programms muss nicht auf Details der Hardware eingegangen werden.
- Zur effizienten Ausführung auf verschiedenen Hardwareplattformen bedarf es weiterer Pattern, die mit Expertenwissen über Parallelrechner implementiert werden müssen.
- *Simulation*-Pattern entkoppelt die Aufgabe der Parallelisierung und Kommunikationsoptimierung von der Aufgabe der numerischen Simulation.

*Verwandte Pattern:* Composite (Abb. A.13 auf Seite 168)

*Diagramm:* **SPH-Simulation**

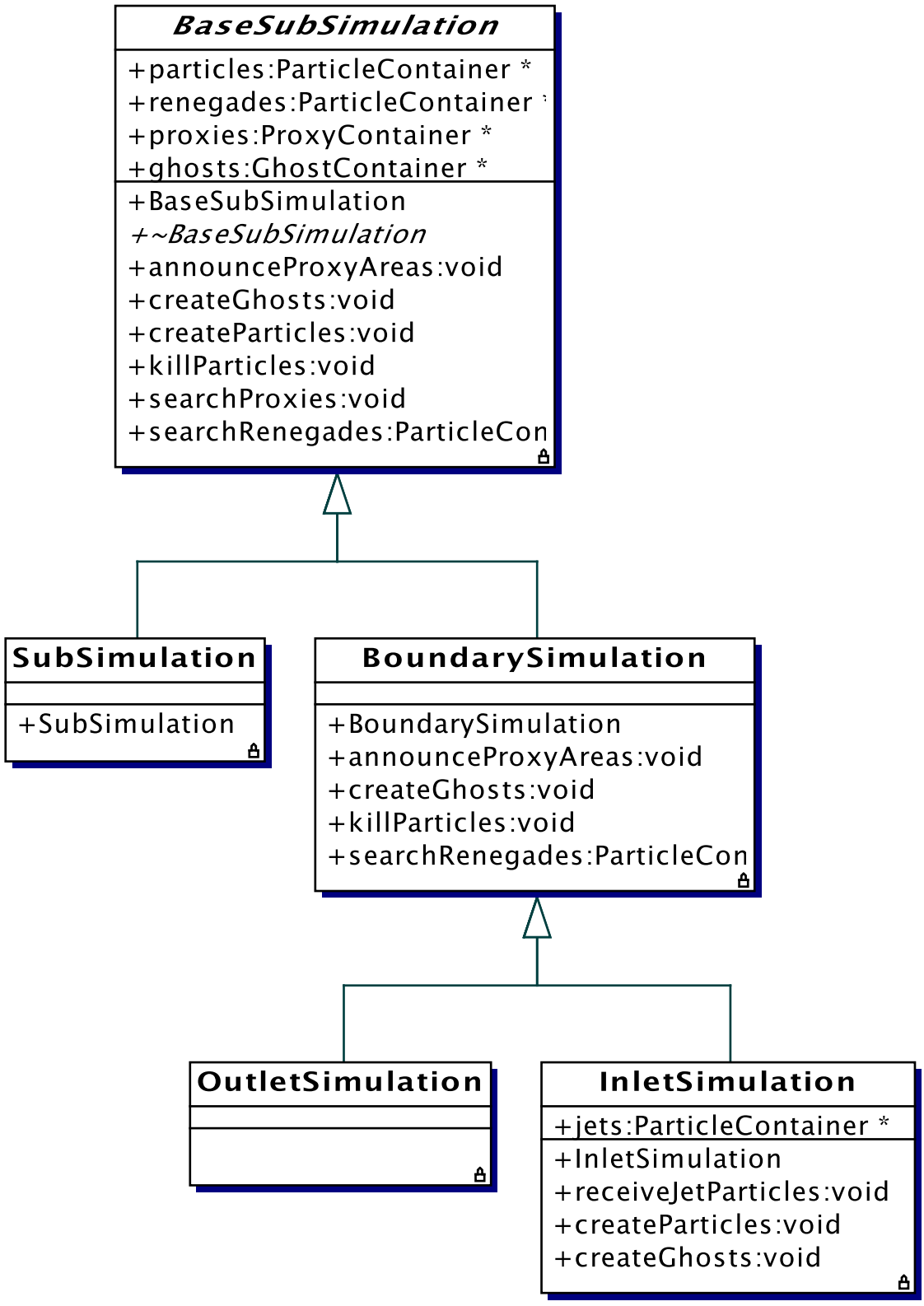



Abbildung A.2: UML-Diagramm als Grundstruktur einer SPH-Simulation.

### A.1.3 SPH-Kernel

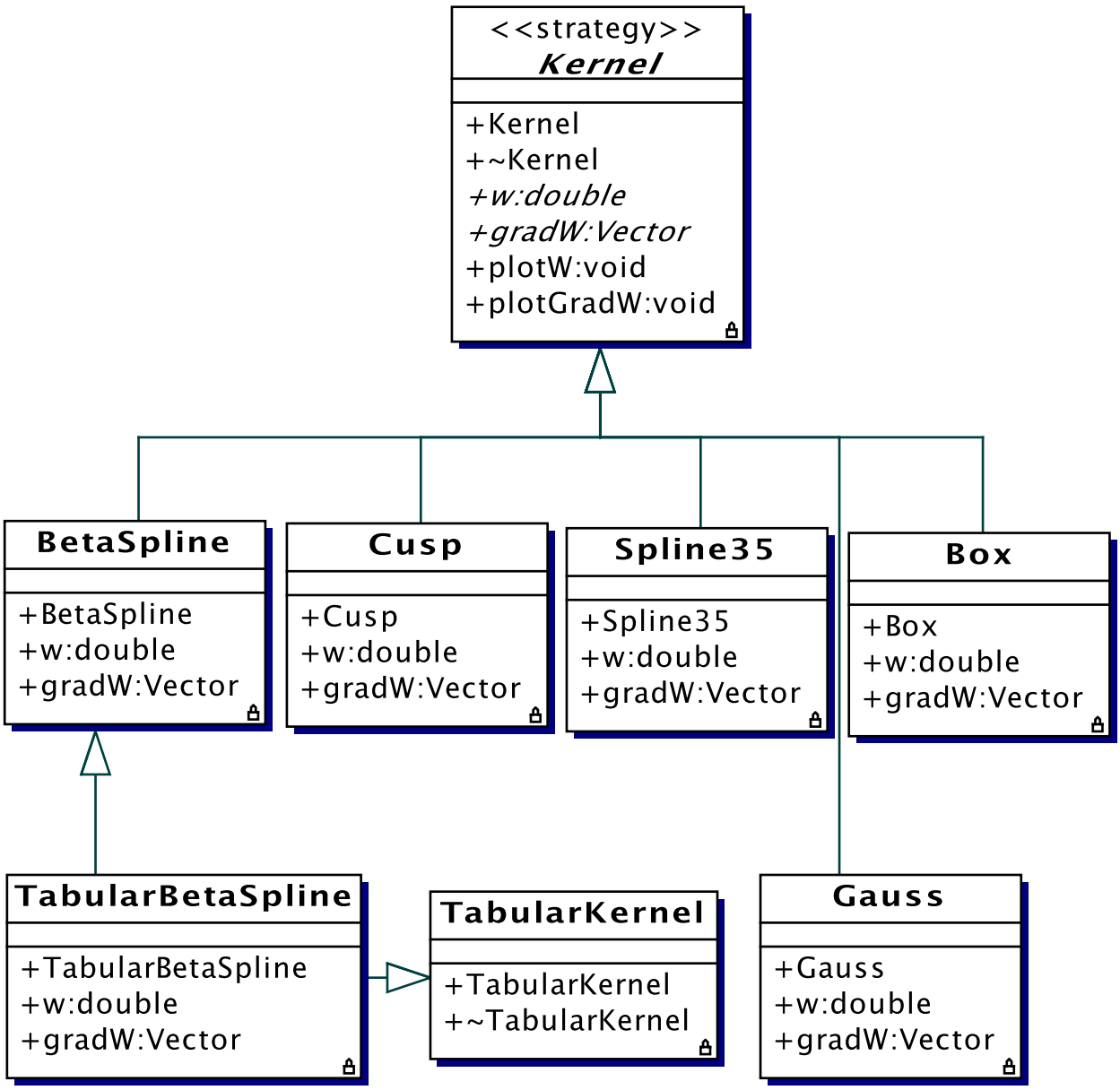



Abbildung A.3: UML-Diagramm für das Design-Pattern zur Auswahl von SPH-Kernel.

*Name:* **SPH-Kernel**

*Problem:* Aus verschiedenen, problemabhängigen Kernel muss ein geeigneter Kernel ausgewählt werden.

*Lösung:* Die SPH-Kernel werden über ein Strategy-Pattern angeordnet. *Kernel* ist die Schnittstelle zu anderen Programmteilen. Der Kontext dieses Strategy-Pattern

sind die Eingabeparameter der Kernelberechnungsfunktionen.

*Konsequenz:*

- Einmal implementierte Kernel können wiederverwendet werden.
- Die Implementierung eines Kernels ist durch die einheitliche Schnittstelle gekapselt, sodass ein Kernel auch in tabellierter Form implementiert werden kann.
- *SPH-Kernel* unterstützt *Low Coupling/High Cohesion.*

*Verwandte Pattern:* Strategy (Abb. A.12 auf Seite 167)

### A.1.4 SPH-Particle

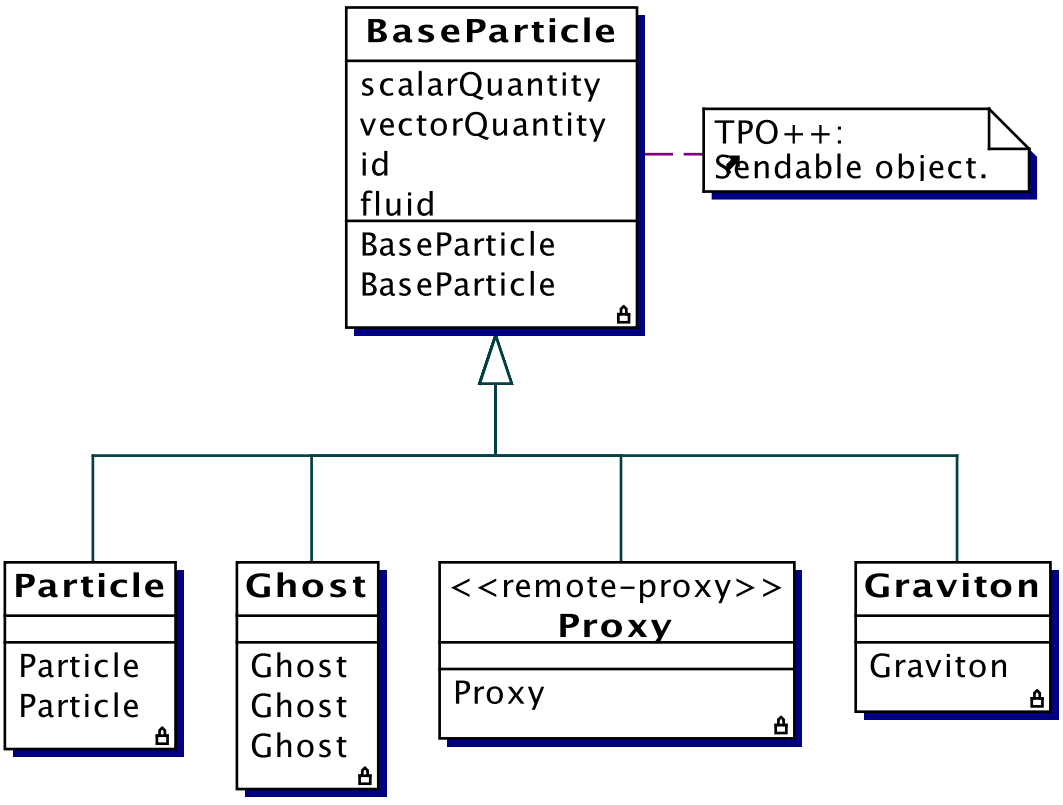



Abbildung A.4: UML-Diagramm des SPH-Particle-Pattern.

*Name:* **SPH-Particle**

*Problem:* Teilchensimulationen arbeiten auf Datenmengen (Teilchen). Die Daten müssen so organisiert sein, dass die Ausführung auf verschiedenen Hardwareplattformen ohne Änderung möglich ist.

*Lösung:* Alle Teilchen-Klassen werden durch eine gemeinsame Schnittstelle dargestellt. Damit sind Teilchen, *Ghost*-Teilchen und *Proxy*-Teilchen ineinander überführbar und für andere Programmteile identisch.

*Konsequenz:*

- Ausführung auf Parallelrechnern mit gemeinsamem und verteiltem Hauptspeicher wird ohne Programmveränderung möglich.
- Entkoppelung von Simulationsprogramm und Parallelisierung wird gefördert.
- Ein Laufzeitoverhead wird beim Zugriff auf die Simulationsdaten eingeführt.

*Verwandte Pattern:* Interface, Remote (siehe [19])

### A.1.5 SPH-Term-Pattern

*Name:* **SPH-Term**

*Problem:* Eine Differentialgleichung wird aus verschiedenen Termen zusammengesetzt. Die einzelnen Terme sollten getrennt wiederverwendbar sein. Die Struktur der Terme muss, unabhängig von der Plattform, auf der das Programm ausgeführt wird, Bestand haben. Die Verantwortlichkeit zum Zusammenbau der Terme darf sich nicht mit der Verantwortlichkeit zum Bau eines einzelnen Terms überlappen.

*Lösung:* SPH-Term-Pattern ist ein Strategy-Pattern für physikalische Größen *Quantity* zusammen mit einem *Builder* zum Aufbau eines Term. Ein Mediator-Pattern verbindet die SPH-Terme mit dem Kontext von anderen Gleichungslösern (wie Integrator).

Jede *Quantity* hat eine konfigurierbare Eigenschaft (Property), die es einem *Iterator* über dem SPH-Term ermöglicht, Kommunikations- und Datensynchronisation einzuführen, falls das erforderlich ist.

*Konsequenz:*

- Das SPH-Term-Pattern entkoppelt die Entwicklung einzelner Terme von dem Aufbau einer Gleichung (*Quantity* vs. *Builder*).
- Einzelne Terme einer Gleichung können wiederverwendet werden.
- Die Entkopplung von Simulation und Parallelisierung wird gefördert.

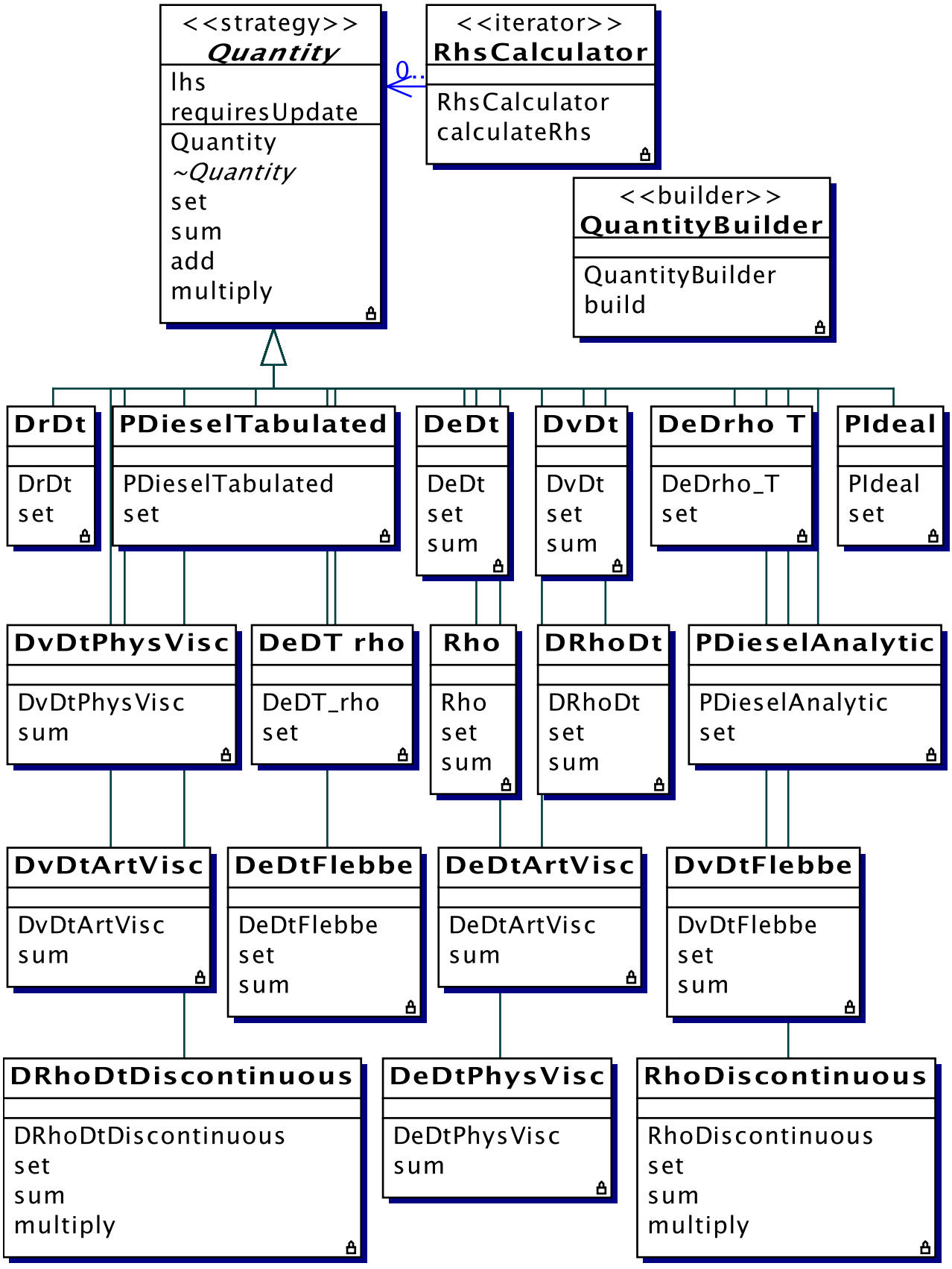



Abbildung A.5: UML-Diagramm des Aufbaus eines SPH-Terms.

- Iteratoren über die SPH-Terme bieten Schnittstellen zu anderen Programmteilen.

*Verwandte Pattern:* Strategy, Iterator, Mediator

## A.2 Design-Pattern zur Parallelisierung

### A.2.1 Pattern: Communicator

*Name:* **Communicator**

*Problem:* Um ein Programm auf verschiedenen parallelen Hardwareplattformen ausführen zu können, muss angenommen werden, dass zwischen Programmteilen kommuniziert wird. Die Kommunikation muss für alle Typen von Parallelrechner gleich aussehen und von dem Simulationsprogramm entkoppelt sein.

*Lösung:* Jeder *SPH-Simulation* wird ein *Communicator* zugeordnet. Ein Communicator ist durch eine Schnittstelle definiert. Implementierungen verschiedener Communicatoren sind für unterschiedliche Hardwareplattformen zuständig.

*Konsequenz:*

- Kommunikation von Daten erfolgt innerhalb des Communicator-Patterns und ist gegenüber anderen Programmteilen entkoppelt.
- Verschiedene Hardwareplattformen (gemeinsamer und getrennter Hauptspeicher) können über die gleiche Communicator-Schnittstelle programmiert werden.
- Entkoppelung von Parallelisierung und Simulation wird gefördert.

*Verwandte Pattern:* SPH-Simulation (Abb. A.1.2 auf Seite 153)

*Diagramm:* **Communicator**

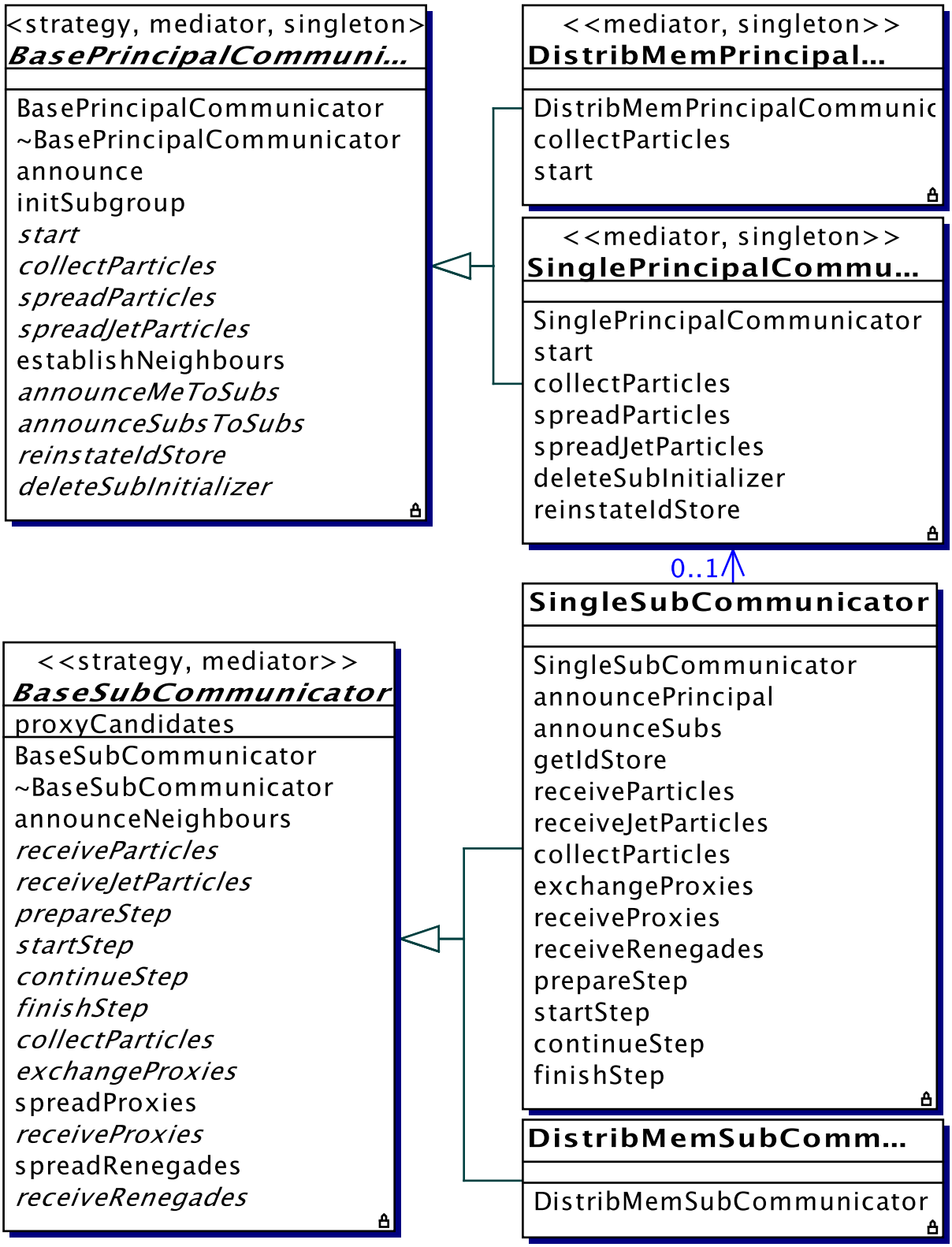



Abbildung A.6: UML-Diagramm für Design-Pattern für Kommunikatoren.

### A.2.2 Zusammenhang: Principals

Der Zusammenhang der Klassen der Hauptsimulation. Eine Diskussion dieser Design-Pattern findet sich in Abschnitt 5.3 auf Seite 113.

*Diagramm:* **Principals**

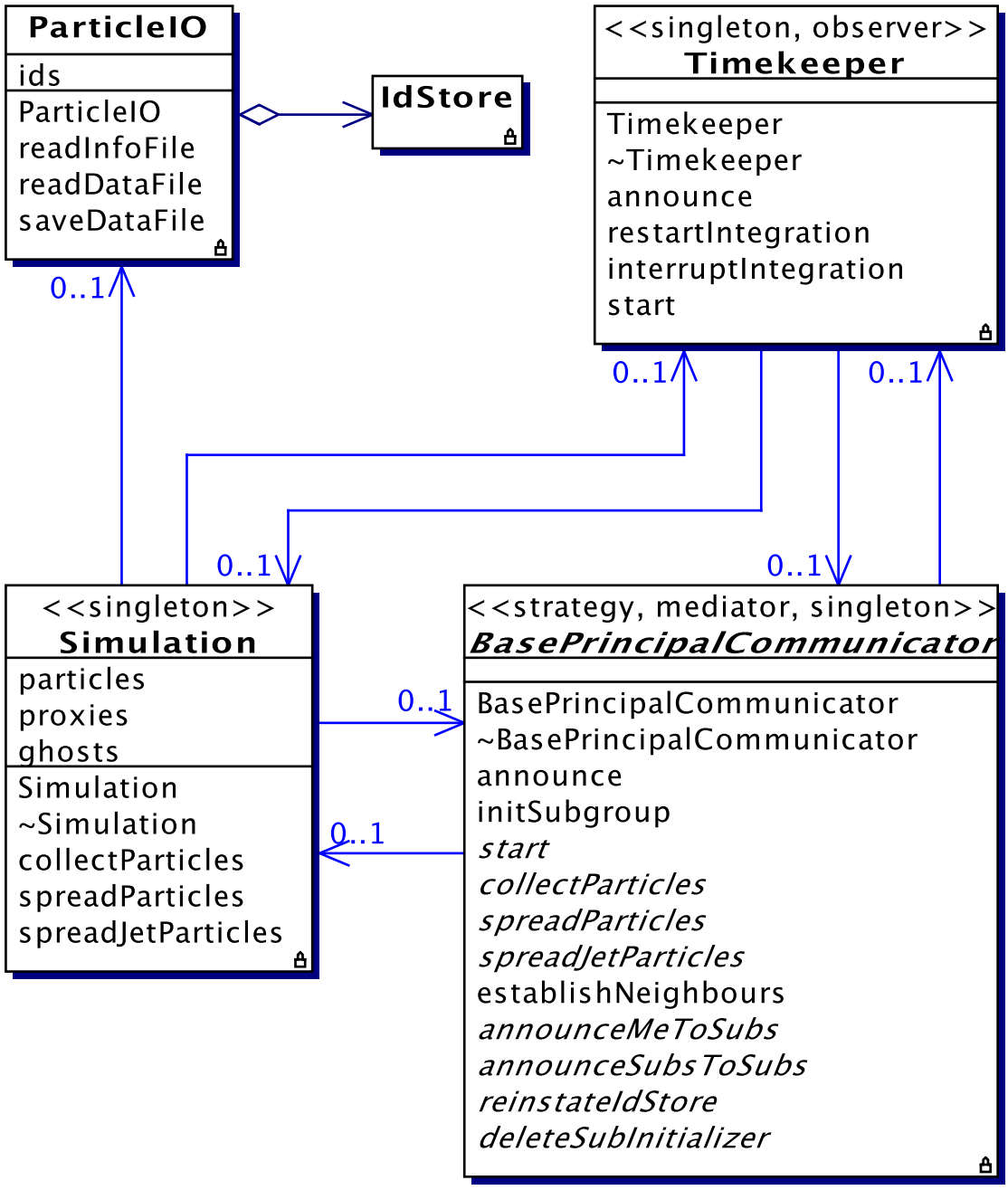



Abbildung A.7: Zusammenhang der Principal-Klassen.

### A.2.3 Zusammenhang: Subgroups

Der Zusammenhang der Klassen der nebenläufigen Untersimulationen. Eine Diskussion dieser Design-Pattern findet sich in Abschnitt 5.3 auf Seite 113.

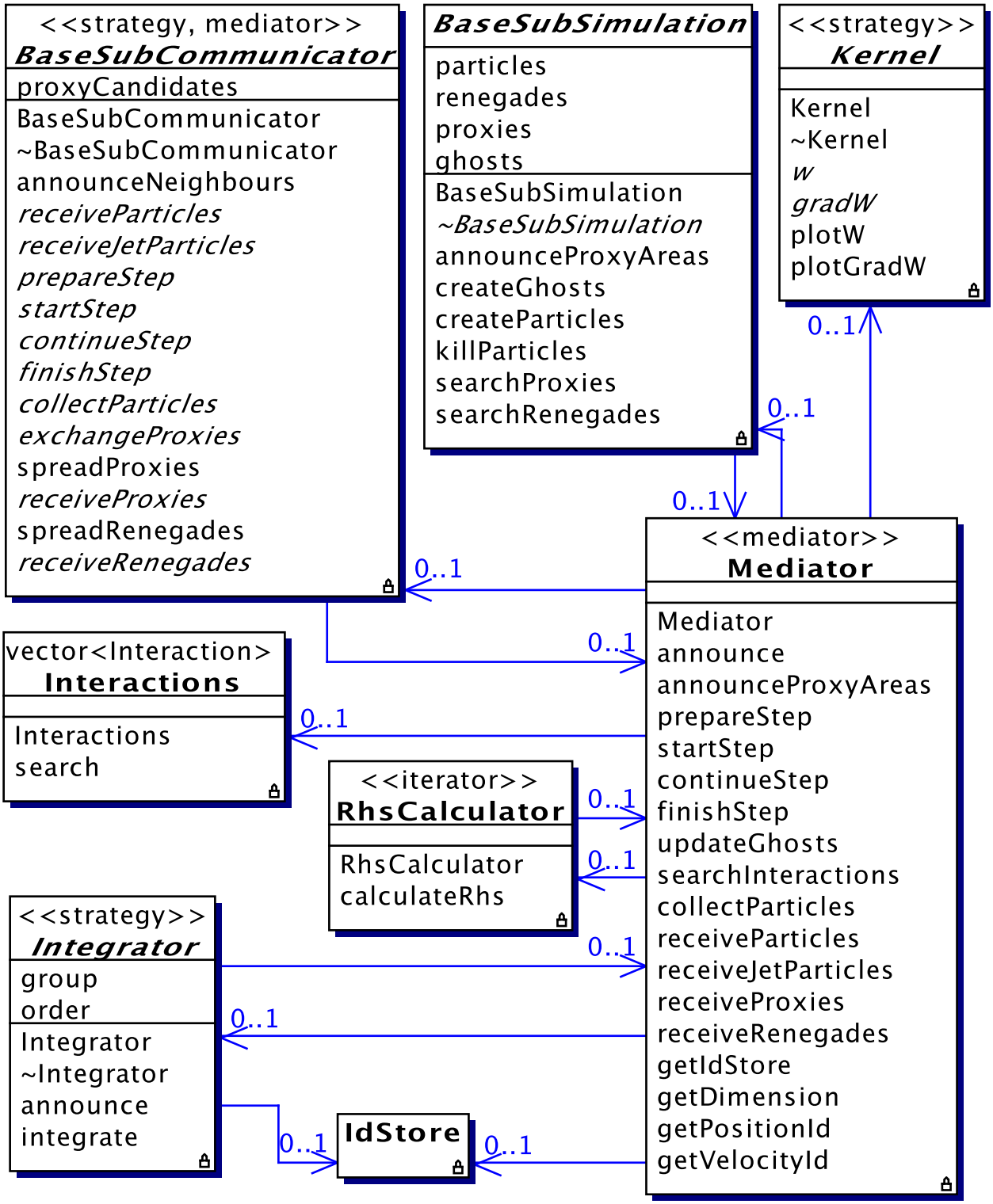



Abbildung A.8: Zusammenhang der Subgroup-Klassen.

## A.3 Sonstige UML-Diagramme

### A.3.1 Objektdiagramm: Quantity-Loop

Die Abfolge der Aufrufe zwischen beteiligten Objekten bei der Berechnung eines SPH-Terms.

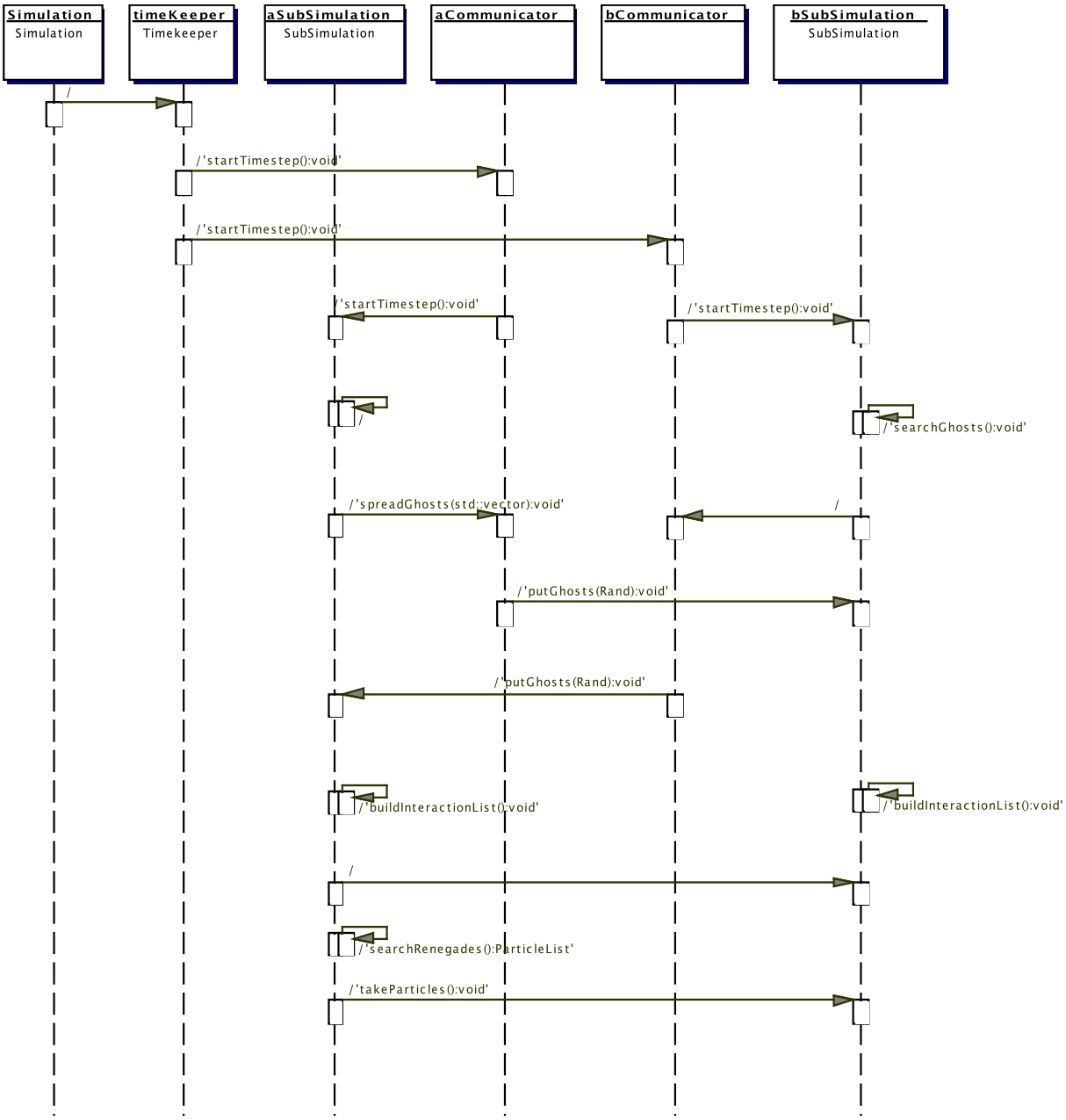



Abbildung A.9: Methodenaufrufe auf Objekten bei SPH-Berechnung.

### A.3.2 Subgroup-Deployment-Diagramm

Zusammenhang der Objekte auf einem Prozessorknoten bei verteilter SPH-Simulation.

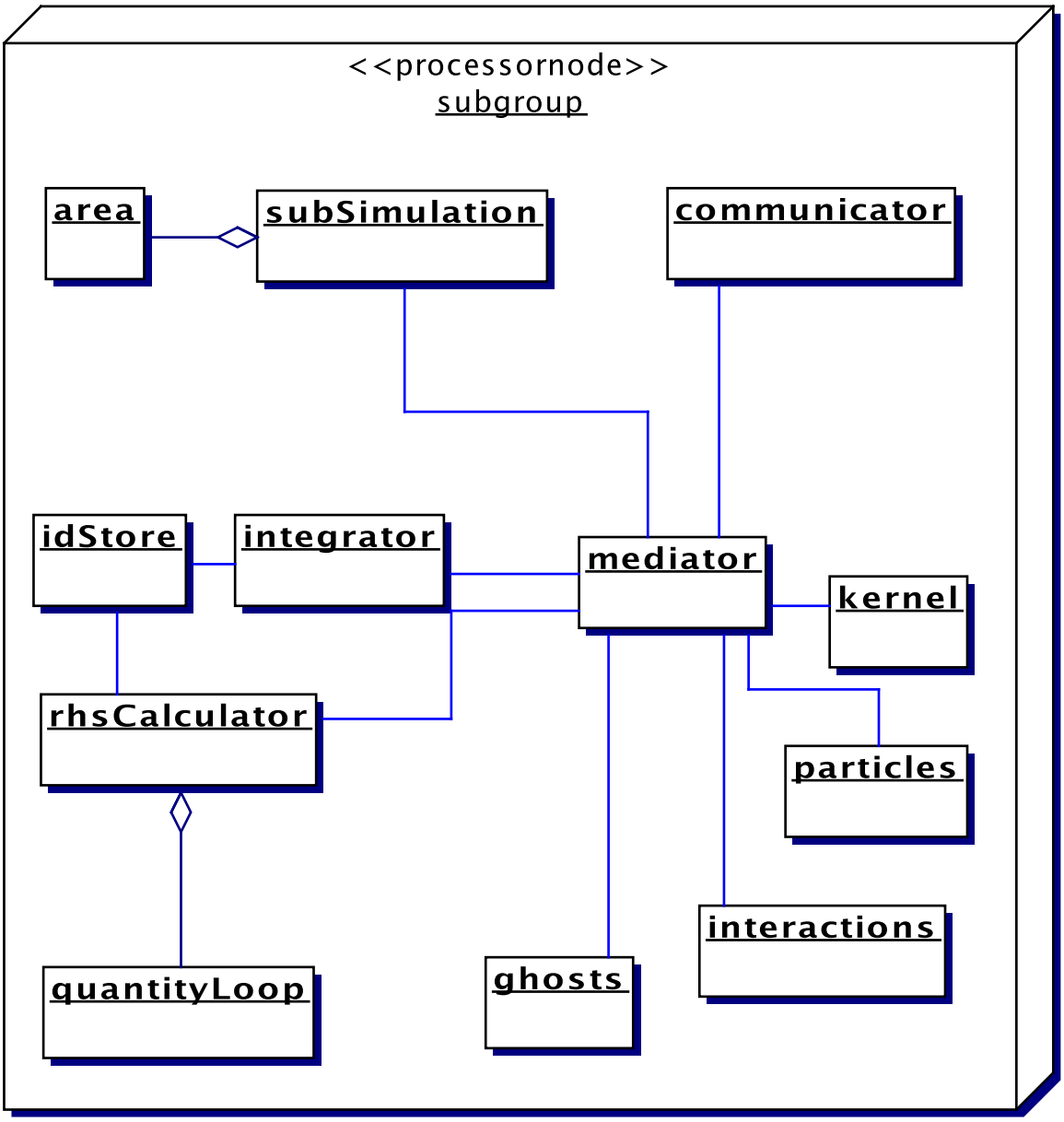



Abbildung A.10: Verteilung der Objekte auf die Prozessorknoten.

### A.3.3 Simulation Areas: Monte-Carlo-Simulation

*Name:* **Monte-Carlo-Simulation**

*Problem:* Eine Simulation kann in verschiedene Teilsimulationen aufgeteilt werden. Die Teilsimulationen haben einen einfachen Zusammenhang in der Form, dass Ergebnisdaten einer Simulation als Eingabedaten einer weiteren Simulation verwendet werden. Die Daten untereinander (Teilchen) haben keine Korrelation. Dies ist bei Monte-Carlo-Simulationen der Fall.

*Lösung:* Teilsimulationen werden als *Composite*-Pattern aufgebaut. Die Basisklasse (Simulation) regelt den Datenfluss zwischen den konkreten Subsimulationen.

*Konsequenz:*

- Kommunikation und Simulation sind entkoppelt.
- Die Teilsimulationen können auf verschiedenen Hardwareplattformen oder heterogenen Netzwerken verteilt werden.
- Es können mehrere Eingabe-Simulationsgebiete Daten für einen Konsumenten produzieren.

*Verwandte Pattern:* Composite, Simulation

*Diagramm:* **Monte-Carlo-Simulation**

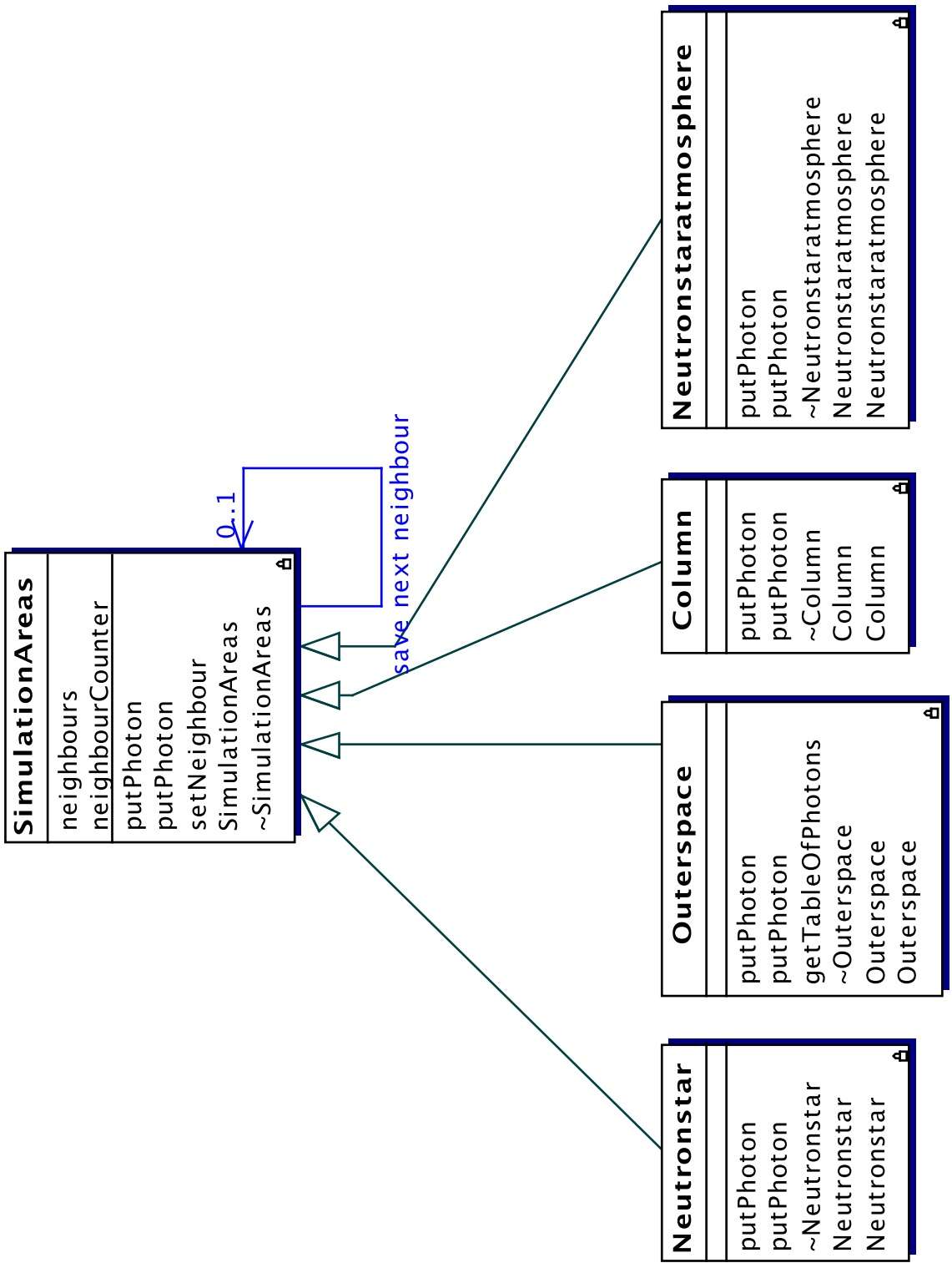



Abbildung A.11: Simulationsgebiete einer Monte-Carlo-Simulation.

## A.4 GoF-Design-Pattern

Die folgenden Design-Pattern sind Standard-Pattern der *GoF — Gang of Four*, die Autoren des Standardwerks [19] (Gamma, Helm, Johnson, Vlissides). Diese Pattern bildeten die Grundlage der Untersuchungen zu den Design-Pattern für paralleles SPH.

### A.4.1 Strategy-Pattern (GoF)

*Name:* **Strategy**

*Problem:* Aus einer Familie von Algorithmen zur Lösung eines Problems muss einer ausgewählt werden können.

*Lösung:* Siehe Abb. A.12.

*Konsequenz:*

- Algorithmen werden hierarchisch in Familien strukturiert.
- Eine Alternative zum Vererbungsmechanismus. Vererbung stellt eine starke Kopplung zwischen Klassen (Super- und Subklasse) dar, die in manchen Situationen nicht erwünscht ist.
- Das Strategie-Pattern vermeidet aufwendige `if-then-else`-Konstrukte.

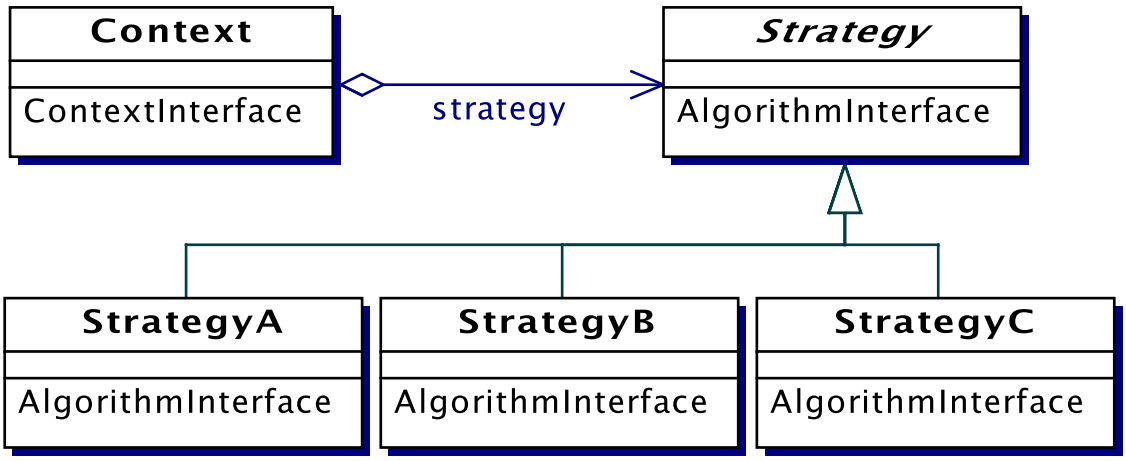



Abbildung A.12: Das GoF-Strategy-Pattern.

### A.4.2 Composite-Pattern (GoF)

*Name:* **Composite**

*Problem:* Ein Problem lässt sich in eine baumartige Struktur untergliedern, bei dem ein Teil als Summe seiner Einzelteile dargestellt werden kann. Beispiel: Eine Linie ist ein Bildelement, ein Bildelement kann aus mehreren Bildelementen aufgebaut sein.

*Lösung:* Siehe Abb. A.13.

*Konsequenz:*

- Überall dort, wo im Programm ein Objekt des Composite-Patterns erwartet wird, kann entweder ein zusammengesetztes oder ein elementares Objekt eingesetzt werden.
- Die Verwendung von zusammengesetzten Strukturen wird vereinfacht.
- Neue elementare Teile einer Struktur können leicht hinzugefügt werden.

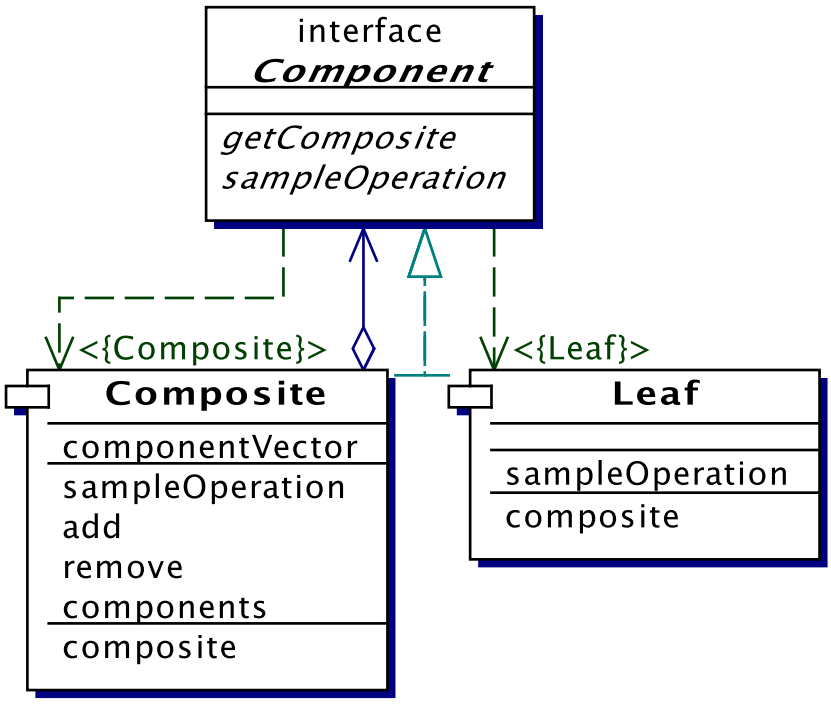



Abbildung A.13: Das GoF-Composite-Pattern.

### A.4.3 Singleton-Pattern (GoF)

*Name:* **Singleton**

*Problem:* Von einer Klasse darf es nur eine einzige Instanz (Objekt) im System geben (Beispiel: exklusiver Zugriff auf Rechenressource).

*Lösung:* Ein Klasse stellt sicher, dass nur eine Instanz existiert. Die Möglichkeiten der Implementierung sind stark sprachabhängig.

*Konsequenz:*

- Kontrollierter Zugriff auf Instanz.
- Es kann leicht eine bestimmte Anzahl $N$ von Instanzen zugelassen werden.
- Flexibler als statische Klassenoperationen.

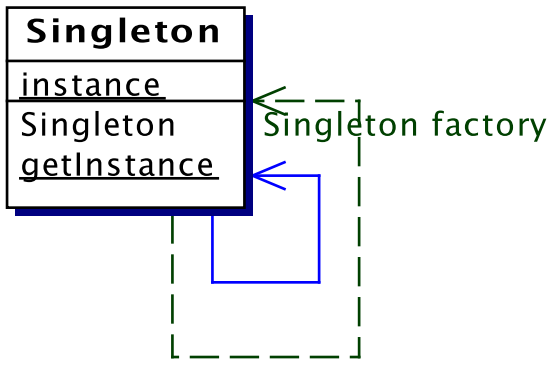



Abbildung A.14: Das GoF-Singleton-Pattern.

### A.4.4 Mediator-Pattern (GoF)

*Name:* **Mediator**

*Problem:* Objekte, die nicht direkt in Beziehung zueinander stehen, müssen miteinander kooperieren.

*Lösung:* Eine spezielle Klasse übernimmt die Koordination von Objekten.

*Konsequenz:*

- Mediator fördert *Low Coupling/High Cohesion*.
- Der Mediator ist eine stark gekoppelte, nicht wiederverwendbare Klasse.
- Die Flexibilität und Wiederverwendbarkeit des Gesamtsystems wird erhöht.

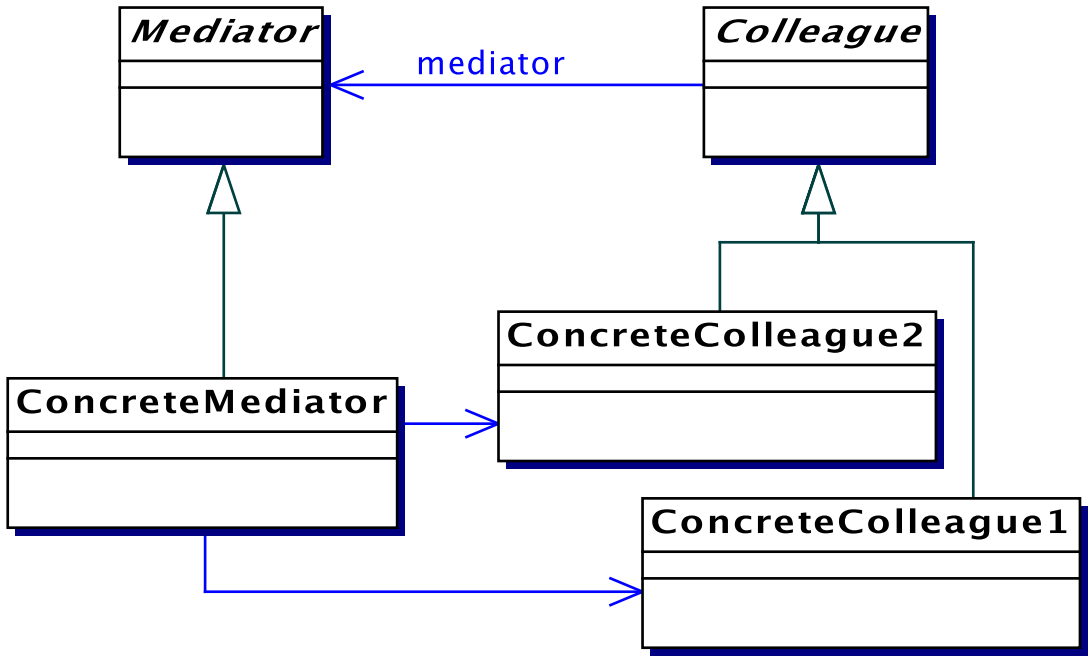



Abbildung A.15: Das GoF-Mediator-Pattern.

### A.4.5 Builder-Pattern (GoF)

*Name:* **Builder**

*Problem:* Die Erzeugung eines Objekts ist ein komplexer Vorgang, der von der Darstellung des Objekts getrennt werden sollte.

*Lösung:* Programmcode für Erzeugung und Repräsentation eines Objekts werden getrennt.

*Konsequenz:*

- Builder fördert *Low Coupling/High Cohesion*.
- Die interne Darstellung eines Produkts (Objekts) kann variiert werden.
- Der Konstruktionsprozess für ein Objekt kann besser strukturiert und kontrolliert werden.

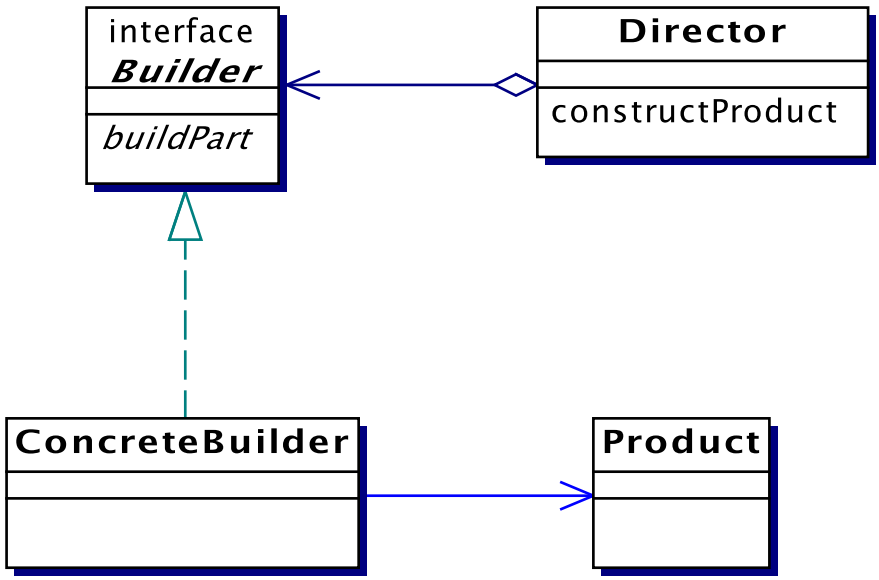



Abbildung A.16: Das GoF-Builder-Pattern.

# Anhang B

# Ausgewählte Algorithmen und Programmcodes

### Code für Nachbarschaftssuche auf Cray T3E

Code-Rahmen des Nachbarschaftssuche-Algorithmus für die Implementierung auf Cray T3E. Siehe Abschnitt 2 auf Seite 91. Der Quellcode zeigt die Version für eine zweidimensionale Problemstellung.

```
1   static void mpp_build_neighbour_list(int idx)
2        /* the list of cells is searched for interaction partners of
3         * every particle in list */
4   {
5       int j;
6       int xzelle,yzelle,zzelle;
7       int minxzelle,maxxzelle,minyzelle,maxyzelle,minzzelle,maxzzelle;
8       double ix[DIM];
9       int idx1,idx2;
10      int dim;
11
12      /* Anfaenglich keine WW-Partner */
13      local.param.partinfos[idx].partners = local.tmp_WWPartner;
14
15      for(dim=0;dim<DIM;dim++)
16        ix[dim] = local.param.particles[idx].x[dim];
17
18      /* alle Nachbarzellen mit potentiellen WW-Partnern durchsuchen */
19      idx1 = pos2gridindex(ix,-global.h);
20      idx2 = pos2gridindex(ix, global.h);
21
22      minxzelle = (idx1>>GRIDSHIFTX) & GRIDMASKX;
23      minyzelle = (idx1>>GRIDSHIFTY) & GRIDMASKY;
24      minzzelle = (idx1>>GRIDSHIFTZ) & GRIDMASKZ;
25      maxxzelle = (idx2>>GRIDSHIFTX) & GRIDMASKX;
26      maxyzelle = (idx2>>GRIDSHIFTY) & GRIDMASKY;
```

```
27      maxzzelle = (idx2>>GRIDSHIFTZ) & GRIDMASKZ;
28
29      for(zzelle = minzzelle; zzelle <= maxzzelle; zzelle++) {
30        for(yzelle = minyzelle; yzelle <= maxyzelle; yzelle++) {
31          for(xzelle = minxzelle; xzelle <= maxxzelle; xzelle++) {
32              /* Liste aller Teilchen dieser Zelle durchsuchen */
33              j = indexgrid_p[(xzelle<<GRIDSHIFTX)|
34                     (yzelle<<GRIDSHIFTY)|(zzelle<<GRIDSHIFTZ)];
35              /*
36               * Es werden alle Wechselwirkungspartner des Teilchens idx
37               * gesucht. Damit treten WW-Paare doppelt auf. Die
38               * Eigenwechselwirkung ist uninteressant.
39               */
40              while(j >= 0) {     /* alle WW-Partner */
41                 if(j != idx) {     /* keine Eigenwechselwirkung */
42                      /* aufgerollte Schleife ueber DIMENSION */
43                      double d = local.param.particles[j].x[X_COORD]
44                                 - ix[X_COORD];
45                      double q = d * d;
46                      d = local.param.particles[j].x[Y_COORD]
47                          - ix[Y_COORD];
48                      q += d * d;
49                      if(q <= local.hsqr) {
50                          if(local.tmp_WWPartner >= local.WWPartner
51                             + global.num_ww_partners)
52                          {
53                                mpp_printf("* FATAL ERROR *\n"...);
54                                exit(1);
55                          }
56                          local.tmp_WWPartner->d2 = q;
57                          local.tmp_WWPartner->index = j;
58                          local.tmp_WWPartner++;
59                      }
60                  }
61                  /* next particle in this cell */
62                  j = local.particle_list[j];
63              } /* while(..) */
64          } /* for(xzelle..) */
65        } /* for(yzelle...) */
66      } /* for(zzelle...) */
67      local.param.partinfos[idx].num_partners =
68          local.tmp_WWPartner - local.param.partinfos[idx].partners;
69      local.all_partners += local.param.partinfos[idx].num_partners;
70  }
```

**Code zur Gittererstellung auf Cray T3E**

Coderahmen zur Erstellung des Gitters auf Cray T3E. Siehe Abschnitt 2 auf Seite 90. Der Quellcode zeigt die Version für eine zweidimensionale Problemstellung.

```
1   static void mpp_build_grid(struct mpp_particle *p,int32 *list)
2   {
3     int i;
4     int divval;
5     struct mpp_partinfo *pi = local.param.partinfos;
6     static int32 tmpgrid_p[GRIDLENFULL];
7
8     memset(tmpgrid_p  ,-1,GRIDLENFULL*sizeof(int32));
9     memset(countgrid_p, 0,GRIDLENFULL*sizeof(int32));
10    memset(indexgrid_p,-1,GRIDLENFULL*sizeof(int32));
11
12    divval = GRIDLENFULL;
13    divval /= linfo.num_pes;
14    if(divval * linfo.num_pes != GRIDLENFULL)
15      divval++;
16
17    for(i=local.param.start;i<local.param.end;i++) {
18      int cellno=0;
19      int old;
20
21      if(!pi[i].magic)
22        continue; /* this particle is 'deactivated' */
23      cellno |= (int)floor((p[i].x[Y_COORD] - grid.min[Y_COORD])
24                * grid.inv_cell_size);
25      cellno <<= (GRIDSHIFTY-GRIDSHIFTX); /* Y-X is OK here */
26      cellno |= (int)floor((p[i].x[X_COORD] - grid.min[X_COORD])
27                * grid.inv_cell_size);
28      cellno <<= GRIDSHIFTX;
29
30      if(tmpgrid_p[cellno] < 0) {
31        old = shmem_short_swap(&indexgrid_p[cellno],
32                               (int32)i,cellno/divval);
33        list[i] = old;
34        tmpgrid_p[cellno] = i;
35      }
36      else {
37        list[i] = list[tmpgrid_p[cellno]];
38        list[tmpgrid_p[cellno]] = i;
39      }
40      countgrid_p[cellno]++;
41    }
42    sph_barrier(6);
```

```
43     /* note: divval*linfo.num_pes can be larger than GRIDLENFULL! */
44     shmem_fcollect32(indexgrid_p,indexgrid_p+divval*linfo.pe,
45                      divval,0,0,linfo.num_pes,pSyncCol[0]);
46
47     shmem_collect32(list,list+local.param.start,
48                     local.param.count,0,0,
49                     linfo.num_pes,pSyncCol[1]);
50     shmem_short_sum_to_all(countgrid_p,countgrid_p,GRIDLENFULL,0,0,
51                            linfo.num_pes,local.pWrkGrid,
52                            pSyncReduce[0]);
53     sph_barrier(6);
54   }
```

**Zellensortierung auf Cray T3E**

Dieser Codeausschnitt zeigt den halb parallelen Mergesort zum Sortieren der Gitterzellen. Siehe Abschnitt 2 auf Seite 90. Der Quellcode zeigt die Version für eine zweidimensionale Problemstellung.

```
1    static int sort_cells(int32 *clist)
2    {
3      int i;
4      int ccount,level,mask;
5      static int32 tmp[GRIDLENFULL];
6      int count0,count1,rest,count,start;
7      int ready = 0;
8
9      for(ccount=0,i=0;i<GRIDLENFULL;i++) {
10       if(countgrid_p[i] > 0) {
11         clist[ccount] = i;
12         ccount++;
13       }
14     }
15
16     /* we use only a power-of-2 # of PEs */
17     if(linfo.pe >= local.pow2num) {
18       ready = 2;
19     }
20
21     rest   = ccount & (local.pow2num-1);
22     count0 = count = ccount >> local.pow2shift;
23     count1 = 0;
24
25     if(linfo.pe < rest) {
26       count0++;
27       start = linfo.pe * count0;
```

```
28      }
29      else {
30        start = linfo.pe * count0 + rest;
31      }
32
33      /* every PE sorts its subband with a simple qsort */
34      if(!ready)
35        qsort( (clist+start) ,count0,sizeof(int32),sort_cell_cmp);
36
37      /* merge bands */
38      mask = 1; level = 0;
39      while(mask < local.pow2num) {
40
41        sph_barrier(18);
42
43        if(!ready && (linfo.pe & mask) ) {
44          ready = 1;
45          shmem_put32(clist+start,clist+start,count0+count1,
46                      linfo.pe-mask);
47        }
48        sph_barrier(18);
49
50        level++; mask <<= 1;
51        if(!ready) {
52          int pe1,cnt0,cnt1;
53          int32 *tmp0,*tmp1,*pnt;
54          count0 += count1;
55          pe1 = linfo.pe + mask;
56          if(pe1 < rest)
57            count1 = pe1 * (count + 1)  - start - count0;
58          else
59            count1 = pe1 * count + rest - start - count0;
60
61          pnt = clist+start;
62          memcpy(tmp,pnt,count0*sizeof(int32));
63          tmp0 = tmp;    tmp1 = clist+start+count0;
64          cnt0 = count0; cnt1 = count1;
65
66          while( cnt0 && cnt1 ) {
67            if( countgrid_p[*tmp0] > countgrid_p[*tmp1] ) {
68              *pnt++ = *tmp0++;
69              cnt0--;
70            }
71            else {
72              *pnt++ = *tmp1++;
73              cnt1--;
74            }
```

```
75          }
76          if(cnt0)
77            memcpy(pnt,tmp0,cnt0*sizeof(int32));
78        } /* !ready */
79      } /* while(..) */
80      sph_barrier(18);
81      return ccount;
82    }
```

**SHMEM/SPH-Code auf Cray T3E**

Der Code entfernt Teilchen, die einen zu großen Abstand zum Zentrum haben. In der eigentlichen Berechnung ist der Code zur Parallelisierung vermischt.

```
1   /* Teilchen mit zu grossem Abstand vom Zentrum werden entfernt */
2   for(i = mpparam.start; i < mpparam.end; i++) {
3     double a1,a2;  /* quadrierter Abst. zum Zentrum */
4
5     if(!mpp_active(i))
6       continue;
7
8     a1 = dist2(param.R2,P.parts.p[i].x);
9     a2 = dist2(param.R1,P.parts.p[i].x);
10    if(a1 < min_sqr || a2 < min_sqr || (a1 > max_sqr && a2 > max_sqr)) {
11      fprintf(stderr,"(2) Teilchen entfernt %e %e %e %e!\n",
12              a1,a2,min_sqr,max_sqr);
13      mpp_remove_particle(i);
14      remove_cnt++;
15    }
16  }
```

**SHMEM/SPH-Code auf Cray T3E**

Berechnung der Kraft auf SPH-Teilchen im Keplerpotential.

```
1   /* Berechnung der Kraft im Keplerpotential & Addition aller Kraefte */
2   for(i = mpparam.start; i < mpparam.end ; i++) {
3   #ifdef USE_KEPLERPOT
4     double mass = param.M1;
5   #else
6     const double mass = 0.0;
7   #endif
8     double r_abs;
9     double tmp,d[DIM];
10    /*
11     * Kraft durch Gestirn 1
```

```
12      */
13     if(!mpp_active(i))
14       continue;
15     tmp = d[X_COORD] = P.parts.p[i].x[X_KOORD] - param.R1[X_KOORD];
16     r_abs = tmp * tmp;
17   # if (DIM >= 2)
18     tmp = d[Y_COORD] = P.parts.p[i].x[Y_KOORD] - param.R1[Y_KOORD] ;
19     r_abs += tmp * tmp;
20   # endif
21   # if (DIM >= 3)
22     tmp = d[Z_COORD] = P.parts.p[i].x[Z_KOORD] - param.R1[Z_KOORD] ;
23     r_abs += tmp * tmp;
24   # endif
25
26     r_abs = r_abs * r_abs * r_abs;
27     r_abs = param.G * mass / sqrt(r_abs);
28
29     L.parts.b[i-mpparam.start][X_KOORD] = - d[X_COORD] * r_abs;
30   # if (DIM >= 2)
31     L.parts.b[i-mpparam.start][Y_KOORD] = - d[Y_COORD] * r_abs;
32   # endif
33   # if (DIM >= 3)
34     L.parts.b[i-mpparam.start][Z_KOORD] = - d[Z_COORD] * r_abs;
35   # endif
36   }
```

**SPH-Code auf NEC SX-4**

SPH-Code mit Multithreading auf der NEC SX-4.

```
1      /*
2       * Aus der Grainsize und der Teilchenzahl errechnen sich die
3       * Anzahl der parallelen Threads zu:
4       * Ganz = (param.tanz%P.s ? param.tanz/P.s+1 : param.tanz/P.s);
5       */
6      P.s = param.tanz / P.Grain;
7      P.s = ( P.s > 0 ? P.s : 1 );
8      for(P.c = 0, g = 0; (P.c + P.s) <= param.tanz; P.c += P.s)
9        dts_id[g++] = fork_KeplerVel(P.s, &(part[P.c]), rkstart, param);
10     P.s = param.tanz - P.c;
11     if( P.s > 0)
12       dts_id[g++] = fork_KeplerVel(P.s, &(part[P.c]), rkstart, param);
13
14     /* collect results */
15     for(g = 0; g < P.Grain; g++)
16       join_KeplerVel(dts_id[g]);
```

**Konfigurationsdatei für SPH-Design-Pattern**

Eine Erläuterung der Konfigurationsdatei ist durch ein Beispiel in Abschnitt 5.4.1 auf Seite 120 gegeben. Die Implementierung der SPH-Design-Pattern wird hier als `sph2000` bezeichnet. Siehe auch [27] und [20].

```
1  ########## Explanation ##########
2  # The metasymbol # is used for comments;
3  # the 'parser' ignors everything after it.
4  # All parameters are set with the following syntax, where the
5  # key describes the parameter:
6  # Key : Value
7  # e.g.:
8  # Kernel : BetaSpline
9  #
10
11 ###################################################################
12 ########## starting SPH2000: sph2000 <prefix> <#><#> ##########
13
14 ### The filename (Prefix without dot) of this simulation
15 # configuration is given
16 # as program argument for the sph2000 executable.
17 # This prefix must be used for all files needed in a
18 # simulation, (e.g. sph2000)
19 # - the config file        (sph2000.config)
20 # - the particle file      (sph2000.data)
21 #   (from which makeparticles generates the sph2000.0.data
22 # and sph2000.jet.data files)
23 # - the particle info file (sph2000.info)
24 # The suffix is hard-coded as shown in the previuos lines.
25 # The other two command line parameters
26 # <#><#> are the filestartnumber and fileendnumber.
27 # e.g.: sph2000 sph2000 0 80
28 # which means, that sph2000.0.data is the initial particle
29 # distribution,
30 # sph2000.80.data will be the last calculated and saved
31 # particle distribution.
32
33 ###################################################################
34 ########## Parallelization and machine architecture ##########
35
36 ### Machine architecture, set which kind of memory the
37 # hardware uses.
38 # Single:      a single processor with the whole memory for
39 # its own usage
40 # Distributed: every node has its own memory
41 Memory : Distributed
```

```
42
43  # Remark: the current code needs TPO++ to compile, although
44  # only the 'Distributed'
45  # case really uses it. This will be change...
46
47  ################################################################
48  ## Choose one of different strategies for the simulation ######
49
50  ### Choose the kind of simulation.
51  # More about injection and fluids see below.
52  # Possible values are:
53  #    Injection: Diesel injection into a air filled internal-
54  # combustion engine
55  Simulation : Injection
56
57  ### Choose a kernel function.
58  #
59  # Possible values are:
60  #   BetaSpline, Box, Cusp, Spline35.
61  #  (Do not use the Gauss kernel, it is unsteady)
62  Kernel : Cusp
63
64  # Should the kernel be calculated exactly, every time it is
65  # needed, (choose No)
66  # or should a table of values be initialized and an
67  # interpolation returns a value (choose Yes)?
68  # The TabGap gives the 'distance between two values in the
69  # table;
70  # unit of measurement is m.
71  TabularKernel : No
72  TabGap        : 1e-7
73
74  ### Choose an integrator.
75  # More about Timing see below.
76  # Possible values are:
77  #  Euler:      Runge Kutta of first order
78  #  Heun:       Runge Kutta of second order
79  #  RungeKutta: *The* Runge Kutta method, 4.th order
80  Integrator : Heun
81
82  ### Choose the shape of the subareas.
83  # This shape is for the domain decomposition. The simulation
84  # area can be rectangular.
85  # Possible values are:
86  #   Quadratic   # that means a square in 2D and a cube in 3D.
87  Area : Quadratic
88
```

```
89  ################################################################
90  ########## The extends and dimensions of the simulation #######
91
92  ### Set the dimension of the simulation space.
93  # Value is an integer in [2;3]:
94  Dimension : 2 ### 3D is not fully implemented!
95
96  ### Set the extensions of the simulation space.
97  # Unit of measurement is m.
98  # We need a base length; the extensions in the three
99  # directions are given as factors of this base length. The
100 # factors must be integers.
101 BaseLength : 1.0e-3
102
103 # The domain decomposition will split this area in
104 # squares/cubes. If you choose a crazy relation, the the
105 # magnitude of the subarea size can't fit the smoothing
106 # length. This means the simulation won't be balanced; it needs
107 # much more time for administration.  So we only accept
108 # integers here.
109 xFactor    : 1
110 yFactor    : 1
111 zFactor    : 1
112
113 ### Set the smoothing length h:
114 # Unit of measurement is m.
115 SmoothingLength : 5.0e-5
116
117 ### How many smoothing lengths should fit in a subarea?  Value
118 # is a rational number >= 1.0 or auto.  NEW! For single
119 # machines, sph2000 uses one big subgroup if auto is set, With
120 # tpo++, auto checks, how many processes are running and tries
121 # to build as much subgroups (-1).
122 SmoothingLengthPerArea : auto
123
124 # Additional to these geometric parameters look for
125 # InletPosition and InletWidth below!
126
127 ################################################################
128 ########## Handling of the boundary conditions ##########
129
130 ### Boundary condition: What does a particle, when it crosses
131 # the boundary?  Reflecting: it reflects back into the
132 # simulation area.  Open: it leaves the simulation area
133 # forever. ### not implemented yet!  Periodic: it leaves this
134 # side.  ### not implemented yet!  On the opposite of the
135 # simulation area, we feed it in as a new particle, with the
```

```
136  # old values.
137  Boundary : Reflecting
138
139  ### Ghosts or direct manipulation of the particles?  If you use
140  # ghosts, than the boundary conditions act on the particles via
141  # so called ghost particles. For every particle which is near
142  # the boundary, a ghost is created to effect the particle so it
143  # sees also a force back inside.  If you say No here, the
144  # particle is manipulated directly, via a potential, which
145  # forms the boundary condition.  Say Yes or No.
146  UseGhosts : Yes
147
148  ##################################################
149  ########## The timing of the simulation ##########
150
151  ### The saving time step.  Say after which time you want to
152  # save the particles to a file.  This period must be greater
153  # than the (initial) timestep of the integrator!  Unit of
154  # measurement is s.  (The simulating time is given by the
155  # program arguments on the command line of sph2000.  There you
156  # tell, which is the starting particle distribution number,
157  # e.g. 0 , and which is the last distribution number to
158  # save. The difference multiplied with the SavingPeriod is the
159  # time, the simulation will run.)
160  SavingPeriod : 4e-9    ### 1e-8
161
162  ### The initial integration time step.  For non-adaptive
163  # integrators, this is the ever lasting timestep.  For adaptive
164  # integrators, this is the first (trying) timestep.  Unit of
165  # measurement is s.
166  Timestep : 1e-9    # 1e-9
167
168  ################################################
169  ########## Fluid-specific parameters ##########
170
171  ### Set the number of fluids Remember to set the fluid id in
172  # the particle data file, when you use more than one fluid
173  # kind.  Value is an integer.
174  Fluids : 2
175
176  ### Set the average molecular mass for all fluids Syntax: mju0
177  # : 0.5 # counting of mju begins with fluid id count 0!  Often
178  # used values are: 0.5 # molecular mass of ionized hydrogen
179  # 1.24 # molecular mass of solar gas 28.8 # molecular mass of
180  # air mju is needed for the ideal gas equation. If the second
181  # fluid has another equation of state which doesn't need the
182  # mju value (e.g. the given diesel equation), we set it 0.
```

```
183   mju0 : 28.8
184   mju1 : 0
185
186   ### The adiabatic exponent gamma = c_p / c_v
187   # We need this to calculate the sound velocity.
188   # See Bergmann/Schaefer Bd.1, p. 648 for some values
189   # Unit of meassurement is 1 (dimensionless number)
190   #    1.4 # gamma_air
191   gamma0 : 1.4
192   gamma1 : 0
193
194   ######################################################
195   ########## Injection-specific parameters ##########
196
197   ### Set the position of the nozzle/inlet
198   # If you want to simulate the whole jet, then choose 'Middle';
199   # if you want to simulate the half jet, generating the whole
200   # by mirroring (like in sph98), then choose 'Corner'.
201   # By default, the inlet is on the y-axis, injecting into the
202   # x-direction.
203   # Possible values are:
204   # Corner # the x-y-Corner, x=0, y=0 # the full inlet width
205   # will be taken!
206   # Middle # in the middle of the y-axis, x=0
207   InletPosition : Middle
208
209   ### Set the extend of the nozzle/inlet
210   # This is the length of the used inlet slit in 2D,
211   # respectively the 'diameter' of the (quadratic) inlet in 3D.
212   # Unit of measurement is m.
213   InletWidth : 1.4e-4
214
215   ### Additional parameters for the initial jet particles are
216   # found in the 'makeparticles'
217   # section, at the end of this file.
218
219   ######################################################
220   ########## Quantity calculation variants ##########
221
222   ### calculation of the density
223   # calculate rho directly, \rho = ...
224   # or via continuity equation, \frac{d\rho}{dt} = ...
225   # Say Direct to use Speith (3.65)
226   # or ContinuityEquation for Speith (3.64)
227   DensityCalculation : ContinuityEquation
228
229   ### Discontinuous density for the fluid-fluid-boundary
```

```
230  # a (correctly) discontinuous density allows a physical pressure
231  # in the contact area of two fluids.
232  # Say Continuous if you want the standard calculation,
233  # Speith (3.64) or (3.65)
234  # or Discontinuous for the calculation after Ott (3.15)
235  # instead of Speith (3.65),
236  # respectively Ott (3.16) instead of Speith (3.64).
237  DensityFluctuation : Discontinuous
238
239  ### Variants of the Euler equation
240  # Calculate SPH-Euler 'normaly', like Speith (3.69)
241  # describes it (Parameter: Normal)
242  # or calculate it after the derivation of Flebbe, Speith
243  # formula (3.80) (Parameter: Flebbe)
244  # Say Flebbe or Normal
245  # If you say Normal, we use (3.69) for DvDt, (3.88) for DeDt.
246  # If you say Flebbe, we use (3.80) for DvDt, (3.94) for DeDt
247  EulerEquation: Normal
248
249  ### Diesel equation of state, p(rho, T)
250  # Use an analytic term of the DEA summer diesel oil or
251  # use a tabulated equation of state for this diesel.
252  # Say Analytic or Tabulated
253  DieselEquation: Analytic     ### Tabulated not implemented
254
255  ### The handling of the temperature
256  # Do you want an isothermal simulation, with a constant T,
257  # say Isothermal.
258  # Or do you want an adiabatic T, using the temperature
259  # equation, then say Adiabatic.
260  Temperature : Isothermal ### Adiabatic not fully implemented!
```

### Implementierung des Kernel-Pattern

Die Diskussion einer Beispiel-Implementierung ist in Abschnitt 5.4.2 auf Seite 122 zu finden. Hier werden die Programmtexte der Basisklasse *Kernel* sowie einer konkreten Kernel-Implementierung aufgezeigt. Siehe auch Kernel-Pattern A.1.3 auf Seite 155.

**Kernel-Basisklasse** Definition der abstrakten Basisklasse des Kernel-Patterns. Alle konkreten Kernelfunktionen müssen von dieser Klasse abgeleitet werden.

```
1  // ***** global macros (prevent multiple inclusion) *****
2  #ifndef KERNEL_HPP
3  #define KERNEL_HPP
4  // ***** system includes *****
5  #include <string>
```

```
 6  // ***** project includes *****
 7  #include "Vector.hpp"
 8  /**
 9   * Design pattern Strategy:
10   *   Defines one interface for different implementations.
11   *
12   *   Kernel for the SPH method. Calculates the kernel function and
13   *   gradient due to the distance of two particles.
14   *
15   * We first implement
16   * a) an isotropic kernel with
17   * b) an 'isotropic' smoothing length.
18   * @stereotype strategy
19   */
20  class Kernel
21  {
22   public:
23
24   /**
25    * Conctructor to set the data members.
26    * Concrete kernels will pre-calculate some factors
27    * in their constructor to save time when calling
28    * w() and gradW().
29    */
30   Kernel(unsigned long group, size_t dim, double smoothingLength);
31
32   /**
33    * Base classes with virtual member functions should have a virtual
34    * destructor.
35    */
36   virtual ~Kernel();
37
38   /**
39    * Calculates the kernel value for the given distance of two
40    * particles.
41    */
42   virtual double w(double distance) const = 0;
43
44   /**
45    * Calculates the kernel derivation for the given distance
46    * of two particles
47    */
48   virtual Vector
49     gradW(double distance, const Vector& distanceVector) const=0;
50
51  protected:
52
```

```
53    /**
54     * The id of the subgroup.
55     * Every object of a subgroup has the same number to
56     * identify his subgroup.
57     */
58    const unsigned long group;
59    /** A local copy of the dimension. */
60    size_t dim;
61    /** A local copy of the smoothing length. */
62    double smoothingLength;
63  };
64  #endif
65  // KERNEL_HPP
```

**Gauss-Kernel (C++-Implementierung)** Der Gauss-Kernel ist eine konkrete Implementierung einer Kernfunktion und muss daher von der Kernel-Basisklasse abgeleitet sein. Siehe Abschnitt 5.4.2.

```
1   // ***** system includes *****
2   #include <iostream>
3   // ***** project includes *****
4   #include "auxiliary.hpp"
5   #include "Gauss.hpp"
6   // ***** class Gauss : public Kernel *****
7   Gauss::Gauss(unsigned long group, size_t dim, double smoothingLength)
8       : Kernel(group, dim, smoothingLength)
9   {
10    // initialize the auxiliary factors
11    reciprocH = 1.0 / smoothingLength;
12
13    // set the dimension dependent auxiliary factors
14    factorW = pow(reciprocH, dim)
15              / pow(pi, static_cast<double>(dim) / 2.0);
16    factorGradW = + 2.0 * pow(reciprocH, dim+2)
17              / pow(pi, static_cast<double>(dim) / 2.0);
18  }
19
20  double
21  Gauss::w(double distance) const
22  {
23    // dist/smoothingLength is often needed
24    double normedDist = distance * reciprocH;
25    return factorW * exp(-normedDist * normedDist);
26  }
27
28  Vector
29  Gauss::gradW(double distance, const Vector& distanceVector) const
```

```
30  {
31    // dist/smoothingLength is often needed
32    double normedDist = distance * reciprocH;
33    //!!! check this due to the fitting offset
34    if (distance != 0.0)  {
35      return (factorGradW * exp(-normedDist * normedDist) / distance)
36              * distanceVector;
37    }
38    else  {
39      // for distance == 0, the distanceVector is a null vector
40      return distanceVector;
41    }
42  }
```

**SPH-Term-Pattern (Quantity)**

Die Klasse Quantity ist die Basisklasse für alle SPH-Term-Klassen aus dem SPH-Term-Pattern.

```
1   class Quantity
2   {
3    public:
4
5    /**
6     * Constructor to initialize the data members, all quantities need.
7     * This is the index of the particle's quantity which will be
8     * summed up (we call it the 'left hand side' lhs) and a bool,
9     * which tells, whether we need an update of the proxies
10    * before the calculation of this quantity.
11    */
12   Quantity(size_t lhs, bool requiresUpdate = false);
13
14   /**
15    * The virtual destructor for this abstract class.
16    */
17   virtual ~Quantity() = 0;
18
19   /**
20    * Über die Operationen set(), sum(), multiply() und add()
21    * werden die einzelnen SPH-Terme zu einem Ausdruck zusammengefügt.
22    * Jeder SPH-Term muss also diese Methoden implementieren.
23    * Die Methoden werden in der Reihenfolge
24    * (1) set() (2) sum(), (3) multiply(), (4) add().
25    * aufgerufen.
26    */
27   virtual void set(ParticleContainer* pc) const;
28
```

```
29    virtual void sum(Interactions* i) const;
30
31    virtual void multiply(ParticleContainer* pc) const;
32
33    virtual void add(ParticleContainer* pc) const;
34
35    /**
36     * If a Quantity needs particle quantites, which are calculated in
37     * the quantityList, then we need an update for the proxies.
38     * This member data tells the rhsCalculator, whether to update
39     * the proxies before the calculation or not.
40     */
41    const bool requiresUpdate;
42  };
```

**SPH-Term-Pattern (Quantity-Builder)**

Der Quantity-Builder ist die Schnittstelle zu den Subsimulationsgebieten. Über den Quantity-Builder werden die einzelnen SPH-Term-Objekte eingebunden.

```
1   class QuantityBuilder
2   {
3     public:
4
5     /**
6      * Durch die ParameterMap werden dem Konstruktor
7      * die benötigten Quantities mitgeteilt.
8      */
9     QuantityBuilder(unsigned long group, const ParameterMap& map,
10                    IdStore& ids);
11    /**
12     * This member function is called from the SubInitializer.
13     * It returns the quantityList for the subgroup.
14     *
15     * Über die ParameterMap wird durch die Methode build() die
16     * liste der SPH-Terme (also die DGL) zusammengebaut.
17     */
18    list<Quantity*>* build();
19
20  private:
21
22    /**
23     * Im Folgenden sind die private Hilfs-Methoden deklariert,
24     * über die die Quantity-List aufgebaut wird.
25     */
26
27    /** Build quantityList for the simulation 'Injection'. */
```

```
28      void buildInjection();
29      /** Use the orbit equation. */
30      void useDrDt(bool withUpdate = false);
31      /** Use the Euler equation. */
32      void useDvDt(bool withUpdate = false);
33      /** Add the normal Euler equation to the list. */
34      void useDvDtNormal(bool withUpdate = false);
35      /** Add the Flebbe Euler equation to the list. */
36      void useDvDtFlebbe(bool withUpdate = false);
37      /** Use artificial viscosity. */
38      void useDvDtArtVisc(bool withUpdate = false);
39      /** Use physical viscosity. */
40      void useDvDtPhysVisc(bool withUpdate = false);
41      /** Use the energy equation. */
42      void useDeDt(bool withUpdate = false);
43      /** Add the normal energy equation to the list */
44      void useDeDtNormal(bool withUpdate = false);
45      /** Add the Flebbe energy equation to the list */
46      void useDeDtFlebbe(bool withUpdate = false);
47      /** Use artificial viscosity. */
48      void useDeDtArtVisc(bool withUpdate = false);
49      /** Use physical viscosity. */
50      void useDeDtPhysVisc(bool withUpdate = false);
51      /** Use the density. */
52      void useRho(bool withUpdate = false);
53      /** Add the direct density calculation to the list. */
54      void useRhoDirect(bool withUpdate = false);
55      /** Add the density calculation to the list. */
56      void useRhoContinuityEquation(bool withUpdate = false);
57      /** Add the diesel gas equation of state */
58      void usePDiesel(bool withUpdate = false);
59      /** Add the analytic diesel gas equation of state */
60      void usePDieselAnalytic(size_t dieselFluid, bool withUpdate=false);
61      /** Add the tabulated diesel gas equation of state */
62      void usePDieselTabulated(size_t dieselFluid, bool withUpdate=false);
63      /** Add the ideal gas equation of state */
64      void usePIdeal(bool withUpdate = false);
65      /** Add the temperature equation to calculate the temperature. */
66      void useDTDt(bool withUpdate = false);
67      /** Add the constitutive equation de/dT for const rho. */
68      void useDeDT_rho(bool withUpdate = false);
69      /** Add the constitutive equation de/drho for const T. */
70      void useDeDrho_T(bool withUpdate = false);
71      /** Add the direct volume calculation to the list. */
72      void useV(bool withUpdate = false);
73
74      /** The parameter map for the builder */
```

```
75      const ParameterMap* const map;
76      /**
77       * Object to store the indices of the particles' quantities.
78       * The QuantityBuilder sets these ids, the integrator and
79       * quantiyList needs them.
80       */
81      IdStore& ids;
82
83      /** The list of quantities to build. */
84      list<Quantity*>* quantityList;
85    };
```

# Index

# Literaturverzeichnis


[1] C. ALEXANDER, S. ISHIKAWA, M. SILVERSTEIN, M. JACOBSON, I. FIKSDAHL-KING, UND S. ANGEL, *A Pattern Language*, Oxford University Press, New York, 1977.

[2] M. ALLEN UND D. TILDESLEY, *Computer Simulation of Liquids*, Clarendon Press Oxford, 1987.

[3] G. S. ALMASI UND A. GOTTLIEB, *Highly parallel computing*, The Benjamin/Cummings series in computer science and engineering, Benjamin/Cummings Publ. Comp., Redwood City, Calif., 1989.

[4] S. ATLAS, S. BANERJEE, J. CUMMINGS, P. HINKER, M. SRIKANT, J. REYNDERS, UND M. THOLBURN, *POOMA: A high performance distributed simulation environment for scientific applications*, in Supercomputing 95, San Diego, CA, Dec. 1995.

[5] G. BOOCH, *Objektorientierte Analyse und Design mit praktischen Anwendungsbeispielen*, Addison-Wesley (Deutschland), 1994.

[6] T. BUBECK, *Untersuchungen zur Parallelisierung und Ausführung von Programmen in verschiedenen parallelen Speicherarchitekturen*, Dissertation, Universität Tübingen, 1998.

[7] *Cactus Code Server*. http://www.cactuscode.org, 1999.

[8] P. COAD, *Object-Oriented Patterns*, Communications of the ACM, 35 (1992).

[9] M. CONWAY, *A Multiprocessor System Design*, in AFIPS Fall Joint Computer Conference, vol. 24 of Spartan Books, 1963.

[10] G. COPELAND UND S. KHOSHAFIAN, *Object Identity*, ACM SIGPLAN Notices, 21 (1986).

[11] J. COPLIEN, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA, 1992.

[12] J. COPLIEN UND D. SCHMIDT, Eds., *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[13] R. DAVÉ, J. DUBINSKI, UND L. HERNQUIST, *Parallel TreeSPH*, Astrophy. J. e-Series, (1997).

[14] H. ENGESSER, Ed., *Duden Informatik*, Dudenverlag, 1988.

[15] O. FLEBBE, *Smoothed Particle Hydrodynamics: Modellierung von Superhump-Lichtkurven*, Dissertation, Universität Tübingen, 1994.

[16] O. FLEBBE, S. MÜNZEL, H. HEROLD, H. RIFFERT, UND H. RUDER, *Smoothed Particle Hydrodynamics: Physical Viscosity and the Simulation of Accretion Disks*, Astrophysical Journal, 431 (1994).

[17] M. FLYNN, *Some Computer Organization and their Effectiveness*, IEEE Trans. on Computers, (1972).

[18] M. FOWLER UND K. SCOTT, *UML konzentriert*, Addison-Wesley, 2 ed., 2000.

[19] E. GAMMA, R. HELM, R. JOHNSON, UND J. VLISSIDES, *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995.

[20] S. GANZENMÜLLER, *Analyse und Design einer objektorientierten SPH-Bibliothek mit Entwurfsmustern unter dem Aspekt der Parallelisierung*, Diplomarbeit, Universität Tübingen, 2000.

[21] R. GINGOLD UND J. MONAGHAN, *Smoothed particle hydrodynamics: theory and application to non-spherical stars*, mn, 181 (1977).

[22] A. GOLDBERG UND D. ROBSON, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[23] J. GOSLING, B. JOY, UND G. STEELE, *The Java Language Specification*, The Java Series, Addison-Wesley, 1997.

[24] M. GRAND, *Patterns in Java*, Wiley, 1999.

[25] R. HELM, I. HOLLAND, UND D. GANGOPADHYAY, *Contracts: Specifying Behavioural Compositions in Object-Oriented Systems*, in Conference on Object-Oriented Programming Systems, Languages, and Applications, N. K. Meyrowitz, Ed., vol. 25, Ottawa, Canada, October 1990, SIGPLAN Notices.

[26] L. HERNQUIST UND N. KATZ, *TreeSPH: A Unification of SPH with the Hierarchical Tree Method*, Astron. Astrophys. Suppl. Series, 70 (1989).

[27] F. HEUSER, *Objektorientierte Implementierung der SPH-Methode für Dieseleinspritzung mit Entwurfsmustern*, Diplomarbeit, Universität Tübingen, 2000.

[28] M. HIPP, *Smoothed Particle Hydrodynamics auf Cray T3E*, Diplomarbeit, Universität Tübingen, 1998.

[29] S. HÜTTEMANN, *Methoden zur Parallelisierung von Smoothed Particle Hydrodynamics*, Diplomarbeit, Universität Tübingen, 1996.

[30] R. E. JOHNSON, *Documenting Frameworks using Patterns*, in Conference on Object-Oriented Programming Systems, Languages, and Applications, A. Paepcke, Ed., vol. 27, Vancouver, British Columbia, Canada, October 1992, SIGPLAN Notices.

[31] B. KERNIGHAN UND D. RITCHIE, *The C Programming Language*, Prentice Hall, 1978.

[32] G. KRASNER UND S. POPE, *A cookbook for using the Model-View-Controler user interface paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, 1 (1988).

[33] T. LAIB, *Parallelisierte Monte-Carlo-Simulation des Strahlungstransports bei akkretierenden Neutronensternen*, Diplomarbeit, Universität Tübingen, 1999.

[34] L. LANDAU UND E. LIFSCHITZ, *Lehrbuch der Theoretischen Physik* `VI` *Hydrodynamik*, Akademie Verlag, 1986.

[35] LANL, *POOMA 2.3.0: Parallel Object-Oriented Methods and Applications*, 2000. URL http://www.acl.lanl.gov/pooma.

[36] D. LEA, *Concurrent Programming in Java: Design Principles and Patterns*, The Java Series, Addison Wesley Longmann, Inc., 1997.

[37] C. LIA, G. CARRARO, C. CHIOSI, UND M. VOLI, *A new parallel TreeSPH*, Astrophy. J. e-Series, (1998).

[38] L. LUCY, *A numerical approach to the testing of the fission hypothesis.*, apj, 82 (1977).

[39] M. MILLIKIN, *Object orientation: What it can do for you; from operating systems to user interfaces, commercial viability is near*, Computerworld, (1989).

[40] J. MONAGHAN, *Why Particle Methods Work*, SIAM J. SCI. STAI. COMPUT, 3 (1982).

[41] D. MUSSER UND A. SAINI, Eds., *STL Tutorial and Reference Guide : C++ Programming with the Standard Template Library.*, Addison-Wesley, 1996.

[42] N. MYERS, *A new and useful template technique: Traits.*, C++ Report, 7(5) (1995).

[43] http://www.omg.org.

[44] F. OTT, *Smoothed Particle Hydrodynamics: Grundlagen und Tests eines speziellen Ansatzes für viskose Wechselwirkungen*, Diplomarbeit, Eberhard-Karls-Universität Tübingen, 1995.

[45] F. OTT, *Weiterentwicklung und Untersuchung von Smoothed Particle Hydrodynamics im Hinblick auf den Zerfall von Dieselfreistrahlen in Luft*, Dissertation, Universität Tübingen, 1999.

[46] W. PREE, *Design Patterns for Object-Oriented Software Development*, ACM Press, Addison-Wesley, 1995.

[47] W. PRESS, B. FLANNERY, S. TEUKOLSKY, UND W. VETTERLING, *Numerical Recipes in C*, Cambridge University Press, 1990.

[48] M. QUINN, *Parallel Computing: Theory and Practice*, McGraw-Hill, 1994.

[49] M. REISER UND N. WIRTH, *Programming in Oberon: steps beyond Pascal and Modula*, Addison-Wesley, 1992.

[50] W. ROSENSTIEL, *Rechnerarchitektur* II. Vorlesung zur Rechnerarchitektur. Universität Tübingen.

[51] A. SILBERSCHATZ, L. PETERSON, UND B. GALVIN, *Operating System Concepts*, Addison-Wesley, third ed., 1991.

[52] R. SPEITH, *Untersuchung von Smoothed Particle Hydrodynamics anhand astrophysikalischer Beispiele*, Dissertation, Universität Tübingen, 1998.

[53] R. STELLINGWERF UND C. WINGATE, *Impact modeling with Smoothed Particle Hydrodynamics*, in Smooth Particle Hydrodynamics in Astrophysics, G. Bono und J. Miller, Eds., vol. 65, Società Astronomica Italiana, 1994.

[54] B. STROUTSTRUP, *Classes: An Abstract Data Type Facility for the C Language*, ACM SIGPLAN Notices, 17 (1982).

[55] B. STROUTSTRUP UND A. ELLIS, *The Annotated C++ Reference Manual*, Addison-Wesley, AT&T Bell Laboratories, Murray Hill, New Jersey, 1990.

[56] A. S. TANENBAUM, *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, N.J., 1992.

[57] T. THIEL, *MEMSY: a Modular Expandable Multiprocessor System*, tech. rep., Universität Erlangen, 1997.

[58] M. UMEMURA, T. FUKUSHIGE, J. MAKINO, T. EBISUZAKI, D. SUGIMOTO, E. L. TURNER, UND A. LOEB, *Smoothed particle hydrodynamics with GRAPE-1A*, Publ. Astron. Soc. Japan, 45 (1993).

[59] J. VLISSIDES, J. COPLIEN, UND N. KERTH, Eds., *Pattern Languages of Program Design*, vol. 2, Addison-Wesley, 1995.

[60] L. WINBLAD, D. EDWARDS, UND R. KING, *Object-Oriented Software*, Addison-Wesley, 1990.

# Lebenslauf

| | |
|---|---|
| **Name:** | Stefan Gathmann genannt Hüttemann |
| **Geburtsort:** | 02.09.1967 in Metzingen |
| **1973 – 1986** | Grundschule und Gesamtschule Tübingen. Am 25.06.1986 habe ich (mit den Leistungskursen Mathematik und Physik) an der Gesamtschule Tübingen mit der allgemeinen Hochschulreife abgeschlossen. |
| **1986 – 1988** | Zivildienst |
| **1988** | Beginn des Physikstudiums an der Eberhard-Karls-Universität Tübingen |
| **1991 – 1992** | Auslandsstudium (Physik und Computer Science) an der Louisiana State University (LSU) in Baton Rouge, LA, USA |
| **1996** | Studienabschluss Diplomphysiker. Am 29. Februar 1996 erhielt ich das Diplom von der Fakultät für Physik der Universität Tübingen. |
| **1996 – April 2001** | Wissenschaftlicher Angestellter an der Universität Tübingen. Die vorliegende Arbeit ist im Rahmen des SFB 328 *Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern* entstanden. |