Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3285

Effiziente parallele Implementierung von hierarchischen N-Body-Algorithmen auf Multicore-Systemen mit GPU-Beschleunigung

Hendrik Hochstetter

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer:	Dipl.-Inf. Hannes Hannak PD Dr. Wolfgang Blochinger
begonnen am:	11. Januar 2012
beendet am:	12. Juli 2012
CR-Klassifikation:	D.1.3, I.6.3, J.2

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

Verzeichnis der Listings

1 Einleitung

Um zu verstehen, wie sich das Universum über die Zeit entwickelt oder wie sich die Dynamik von Makromolekülen auf atomaren Ebene darstellt, müssen sehr komplexe Systeme von wechselseitig interagierenden Körpern betrachtet werden. Bereits für sehr kleine Systeme aus nur drei Körpern können analytische Lösungen lediglich für Spezialfälle angegeben werden, weshalb im Allgemeinen auf näherungsweise Lösungsverfahren ausgewichen werden muss. Mit Hilfe der Simulation dieser N-Body-Probleme lassen sich in verschiedenen naturwissenschaftlichen Disziplinen, wie der Astrophysik oder der Biochemie, Erkenntnisse gewinnen, die allein durch Theorie und Experiment nicht zugänglich wären. Simulationen lassen sich in diesem Zusammenhang als virtuelle Experimente betrachten, die entweder dazu eingesetzt werden können, bestimmte Szenarien vorauszuberechnen oder nachzustellen.

Besonders in astrophysikalischen Anwendungen dienen Simulationen dazu, theoretische Modelle zu validieren, da Experimente aufgrund der betrachteten Größenordnungen nicht zu realisieren sind. Gleichermaßen sind Beobachtungen in der Natur nur sehr eingeschränkt möglich, da die betrachteten Zeitskalen von Vorgängen, wie der Kollision von Galaxien, meist im Bereich vieler Millionen von Jahren liegen. Mit Hilfe von virtuellen Experimenten lässt sich diese Lücke allerdings zunehmend besser schließen.

Die stetig zunehmende Rechenleistung von Computersystemen lässt sich dazu einsetzen, Simulationen mit immer größeren Zahlen von Körpern durchzuführen, um möglichst detailgetreue Szenarien nachzubilden. Allein unsere Milchstraße besitzt Schätzungen zufolge über $200 \cdot 10^9$ Sterne.

1.1 Problemstellung

Bei der Entwicklung von Rechnersystemen hat sich in den letzten Jahren gezeigt, dass Leistungssteigerungen allein durch Erhöhung der Prozessorgeschwindigkeit nicht mehr zu realisieren sind. Stattdessen werden heute Hauptprozessoren mit mehreren Prozessorkernen eingesetzt, die durch die parallele Ausführung von Berechnungen Simulationen beschleunigen können. Zudem lassen sich aktuelle Generationen von Graphikkarten dazu einsetzen, Berechnungen zu übernehmen. Sie zeichnen sich besonders durch ihr Potential aus, reguläre Probleme massiv parallel berechnen zu können und besitzen dazu mehrere hundert Prozessorkerne. Die Ausnutzung dieser Parallelität von Multicore-Systemen mit

GPU-Beschleunigung zum bestmöglichen Ausschöpfen der Rechenleistung geht mit besonderen Anforderungen an den Entwurf und die Implementierung von Algorithmen einher, die beide im Mittelpunkt der vorliegenden Arbeit stehen. Auf die genauen Zielsetzungen soll im folgenden Abschnitt detaillierter eingegangen werden.

1.2 Zielsetzungen dieser Arbeit

Die Simulation von N-Body-Problemen ist sehr rechenintensiv. Mittels hierarchischer Ansätze lassen sich approximative Verfahren zur Lösung angeben. In dieser Diplomarbeit soll die Frage erörtert werden, inwieweit sich hierarchische N-Body-Algorithmen mit Hilfe von Multicore-Prozessoren und dem Einsatz von Graphikkarten beschleunigen lassen. Als Verfahren zur Simulation wurde der Barnes-Hut-Algorithmus gewählt. Dieser lässt sich in unterschiedliche Teilprobleme zerlegen, die nacheinander abgearbeitet werden können.

Um festzustellen, welche Teilprobleme des Barnes-Hut-Algorithmus' sich besser für die Ausführung auf dem Hauptprozessor oder der Graphikkarte eignen, werden unterschiedliche Implementierungen zur Lösung der einzelnen Teilprobleme entworfen und vorgestellt. Die Teilproblemlösungen werden dabei als Module entworfen, sodass der Gesamtalgorithmus aus einzelnen Modul-Implementierungen frei zusammengestellt werden kann. Es soll daraufhin untersucht werden, welche Kombinationen der einzelnen Implementierungen für unterschiedliche Problemgrößen am besten geeignet sind.

Werden Implementierungen von Modulen miteinander kombiniert, die zu einem Teil auf der Graphikkarte und zu einem anderen Teil auf dem Hauptprozessor ausgeführt werden, müssen an den Schnittstellen zwischen den entsprechenden Teilproblemen Daten zwischen Arbeitsspeicher und Graphikkarte übertragen werden. Es soll dabei untersucht werden, wie stark sich Speicherlatenzen auf die Ausführungsgeschwindigkeit auswirken und welche Möglichkeiten sich ergeben, Speicherlatenzen zu verstecken.

1.3 Gliederung der Arbeit

Im Weiteren ist die Arbeit auf folgende Weise in Kapitel untergliedert:

Kapitel 2 Theoretische Grundlagen des N-Body-Problems gibt die physikalischen und theoretischen Grundlagen dieser Arbeit und alle nötigen Definitionen und Formeln wieder, die im weiteren Verlauf zum Verständnis der vorgestellten Konzepte und Algorithmen erforderlich sind. Zudem werden die den Algorithmen zugrundeliegenden Ideen erläutert.

Kapitel 3 Parallele Programmierung stellt unterschiedliche Parallelrechnerarchitekturen vor und vergleicht diese miteinander. Es folgt die Einordnung der verwendeten Architekturen in die eingeführten Klassifikationen. Anschließend wird darauf eingegangen, wie die unterschiedlichen parallelen Rechnerarchitekturen programmiert werden können. Zuletzt werden verschiedene Kostenmaße für Algorithmen vorgestellt, anhand derer sich die Leistungsfähigkeit paralleler Algorithmen quantitativ erfassen lässt.

Kapitel 4 Verwandte Arbeiten soll einen kurzen Überblick über Arbeiten anderer Autoren geben, die sich mit verwandten Themen befassen

Kapitel 5 Effiziente parallele Implementierung des Barnes-Hut-Algorithmus' zeigt auf, wie sich der Barnes-Hut-Algorithmus effizient auf Multicore-Systemen mit GPU- Beschleunigung implementieren lässt. Dazu wird der Algorithmus in mehrere Teilprobleme zerlegt, die getrennt voneinander nacheinander gelöst werden können. Insbesondere können die Teillösungen dabei auf unterschiedlichen Hardware-Plattformen ausgeführt werden, wobei sich zeigt, dass bestimmte Teilprobleme sich besser für die Ausführung auf Graphikkarten eignen, während andere auf dem Hauptprozessor bessere Leistungen erzielen.

Kapitel 6 Zusammenfassung und Diskussion der Ergebnisse fasst die Teilergebnisse aus dem vorangegangen Kapitel der Arbeit zusammen.

Kapitel 7 Zusammenfassung und Ausblick fasst die gesamte Arbeit kurz zusammen und soll einige Anknüpfungspunkte aufzeigen, auf denen zukünftige Entwicklungen fußen könnten.

2 Theoretische Grundlagen des N-Body-Problems

In diesem Kapitel sollen die theoretischen Grundlagen zur Beschreibung von N-Body-Problemen beleuchtet und verschiedene algorithmische Lösungsansätze zur effizienten Simulation vorgestellt werden. N-Body-Probleme können in den verschiedensten physikalischen, chemischen und biologischen Modellen auftreten. Besonders hervorzuheben ist dabei die Astrophysik, da auf diesem Forschungsgebiet Simulationen deswegen von besonderer Bedeutung sind, weil sich Experimente aufgrund der extremen Größenordnungen in Zeit und Raum kaum realisieren lassen.

Abschnitt 2.1 wird zunächst die physikalischen Gesetze beschreiben, die N-Body-Problemen zugrundeliegen, bevor in Abschnitt 2.2 Diskretisierungsverfahren vorgestellt werden sollen, mit deren Hilfe Simulationen realisiert werden können. Die Abschnitte 2.3 und 2.4 gehen schließlich auf unterschiedliche algorithmische Ansätze zur Lösung von N-Body-Problemen ein. Da für die nachfolgenden Beschreibungen stets Systeme kollisionfreier Körper unendlich kleiner räumlicher Ausdehnung, d.h. Punktmassen, betrachtet werden, wird im Weiteren, synonym zur Punktmasse, der Begriff Partikel dem Begriff Körper bevorzugt verwendet.

2.1 Physikalische Beschreibung des N-Body-Problems

Für eine analytische Lösung von N-Body-Problemen müssen für jedes Partikel i Differentialgleichungen gelöst werden, die zu jedem Zeitpunkt von den Eigenschaften aller anderen betrachteten Partikel abhängig sind. Die Bewegungen der einzelnen Partikel berechnen sich dann nach den physikalischen Gesetzen der Kinematik und der Dynamik, die in den Gleichungen 2.1 – 2.3 angegeben sind. Dabei bezeichnet $\vec{F}_i$ für ein Partikel i die physikalische Größe der Kraft, $\vec{a}_i$ die Beschleunigung, $\vec{v}_i$ die Geschwindigkeit und m_i die Masse. Mit $\vec{x}_i$ wird die Position des Partikels im Raum identifiziert.

$$\vec{F}_i(t) = m_i \cdot \vec{a}(t) \tag{2.1}$$

$$\vec{a}_i(t) = \dot{\vec{v}}_i(t) \tag{2.2}$$

$$\vec{v}_i(t) = \dot{\vec{x}}_i(t) \tag{2.3}$$

Die Kraft auf ein einzelnes Partikel resultiert zu jedem Zeitpunkt aus der Wirkung von Potentialen, die aus der Verteilung der anderen Partikel im Raum entstehen. Das Potential lässt sich interpretieren als die Fähigkeit von Partikeln, Kräfte aufeinander auszuüben [BZBP09]. Die Kraft, die auf ein Partikel i durch ein anderes Partikel j ausgeübt wird, kann nach Gleichung 2.4 aus dem negativen Gradienten des Potentials zwischen den beiden Partikeln bestimmt werden.

$$\vec{F_{ij}} = -\nabla U(r_{ij}) \tag{2.4}$$

Werden alle Kräfte aus den Paarpotentialen aufsummiert, erhält man nach Gleichung 2.5 die Kraft, die insgesamt auf ein Partikel i wirkt.

$$\vec{F_i} = \sum_{j \neq i} -\nabla U(r_{ij}) = \sum_{j \neq i} \vec{F_{ij}} \tag{2.5}$$

Je nachdem, was für Szenarien nachgestellt werden sollen, können unterschiedliche Potentialfunktionen betrachtet werden. In astrophysikalischen Anwendungen wird das Gravitationspotential betrachtet, das sich dadurch auszeichnet, dass es stets eine Anziehung zwischen Körpern bewirkt und eine sehr lange Reichweite besitzt. Molekulardynamiksimulationen dagegen werden meist mit kurzreichweitigen Potentialen, wie dem Lennard-Jones-Potential, durchgeführt.

Für die vorliegende Arbeit wollen wir uns schwerpunktmäßig auf astrophysikalische Anwendungen konzentrieren, bei denen die Wechselwirkung zwischen den N Körpern durch die Gravitation vermittelt wird. Deshalb wollen wir das Gravitationspotential im folgenden Abschnitt etwas genauer betrachten.

2.1.1 Das Gravitationspotential

Das Gravitationspotential ist das einzige Potential, das zwischen Körpern stets anziehende Kräfte bewirkt. Zudem nimmt die Gravitation nur umgekehrt proportional zum Abstand ab, was zu sehr langreichweitigen Kräften führt. In Gleichung 2.6 ist das Gravitationspotential angegeben und Abbildung 2.1 zeigt den Graphen eines Potentials in Abhängigkeit vom Abstand r_{ij} zweier Partikel i und j.

$$U_{\text{grav}}(r_{ij}) = -G\frac{m_i m_j}{r_{ij}} \tag{2.6}$$

Die Konstante G wird als *Gravitationskonstante* bezeichnet. Die Kraft, die aus dem Potential zwischen einem Paar von Partikeln resultiert ist in Gleichung 2.7 angegeben. Es gilt dabei $\vec{r}_{ij} = \vec{x}_j - \vec{x}_i$.

$$\vec{F_{ij}} = G\frac{m_i m_j}{r_{ij}^3}\vec{r}_{ij} \tag{2.7}$$

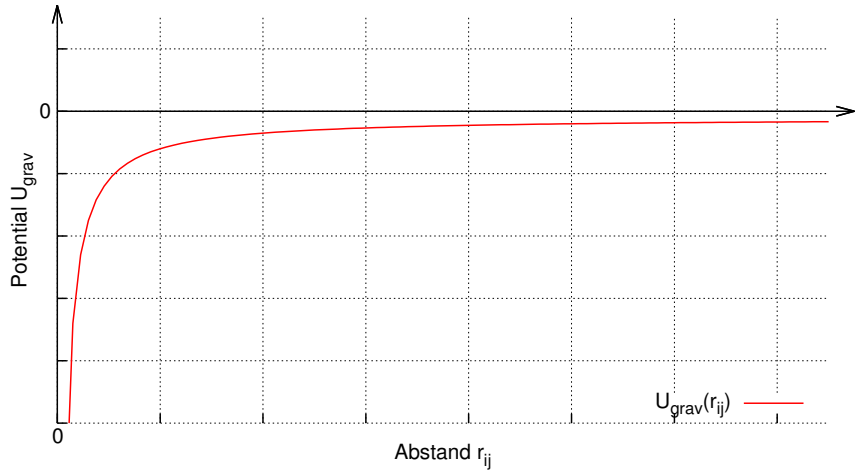


Abbildung 2.1: Verallgemeinertes Gravitationspotential in Abhängigkeit des Abstands zwischen Massen

Im Gegensatz zum Gravitationspotential können andere Potentiale mit dem Abstand zwischen Partikeln deutlich schneller abfallen. Optimierte Algorithmen zur Lösung von N-Body-Problemen mit kurz- und langreichweitigen Potentialen sollen in Abschnitt 2.3.2 vorgestellt werden.

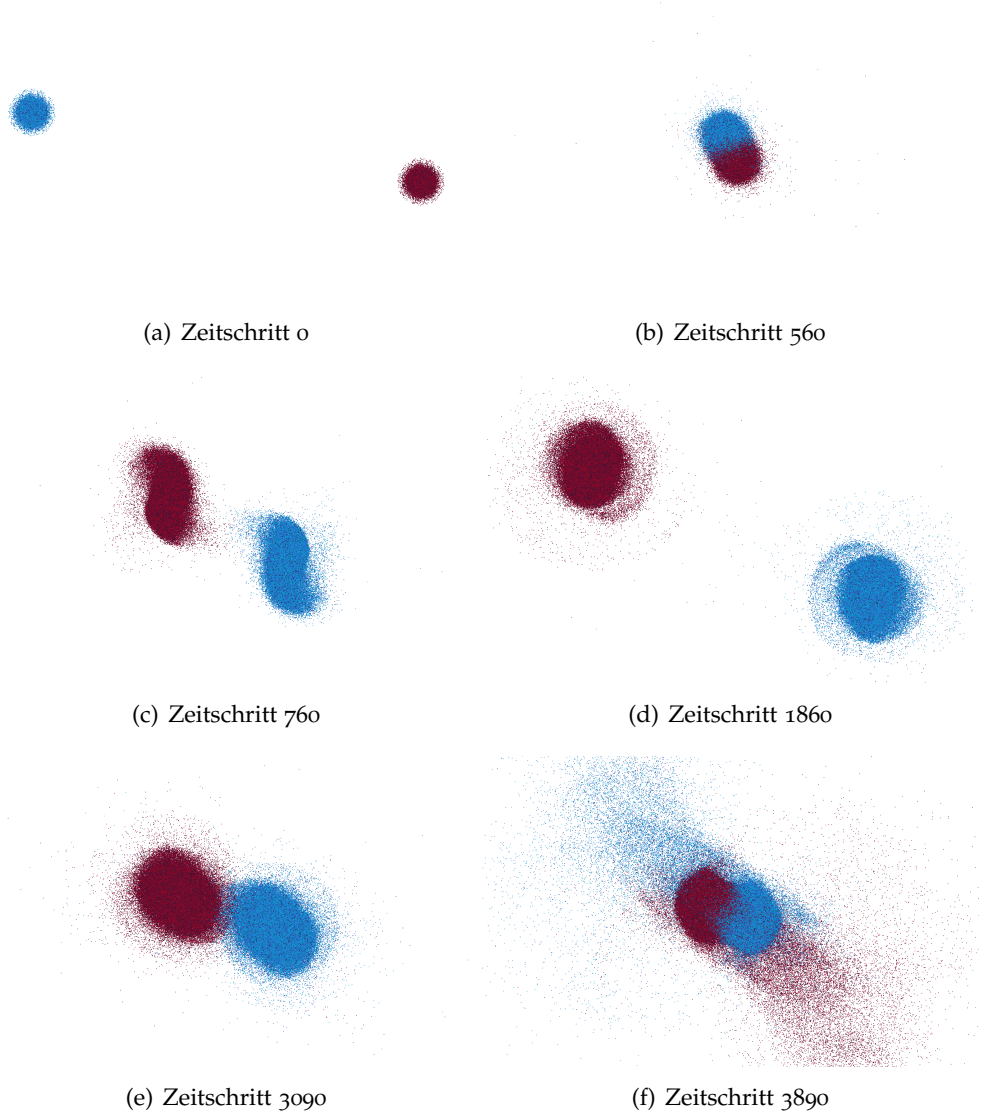
2.1.2 Größenordnungen der simulierten astrophysikalischen Probleme

Je nachdem, ob Simulationen auf kosmologischer Ebene durchgeführt werden — dann entspricht jedes simulierte Partikel in der Regel einer Galaxie — oder auf der Ebene von Galaxien — dann wird mit jedem Partikel eine Menge von Sternen beschrieben — , werden sehr unterschiedliche Größenordnungen betrachtet. Für die vorliegende Arbeit wurden ausschließlich Simulationen auf der Ebene von Galaxien durchgeführt.

Es ist in der Astrophysik üblich, Einheiten im CGS-System — in cm, g und s — anzugeben statt im MKS-System, in dem m und kg als Einheiten für Länge und Gewicht dienen. Die wichtigsten Bezugsgrößen im CGS-System sollen im Folgenden angegeben werden.

- Zur Angabe von Entfernungen werden die *Astronomische Einheit*, d.h. der mittlere Abstand zwischen Sonne und Erde mit $1\,\mathrm{AU} = 1{,}496 \cdot 10^{13}\,\mathrm{cm}$, und das *Parsec* mit $1\,\mathrm{pc} = 3{,}086 \cdot 10^{18}\,\mathrm{cm} = 3{,}2615668\,\mathrm{Lj}$ verwendet (Lj = Lichtjahre)
- Massen werden typischerweise als Vielfaches der Sonnenmasse mit $1\,\mathrm{M}_\odot = 1{,}989 \cdot 10^{33}\,\mathrm{g}$ angegeben
- Zeitangaben erfolgen in Jahren mit $1\,\mathrm{yr} = 3{,}156 \cdot 10^{7}\,\mathrm{s}$

Die in dieser Arbeit durchgeführten Simulationen stellen die Kollision zweier Galaxien nach, wie sie in Abbildung 2.2 dargestellt ist. Die Größen richten sich dabei in etwa nach denen der Milchstraße und der Galaxie Andromeda, von denen beiden erwartet wird, dass sie innerhalb der nächsten $6 \cdot 10^9$ yr kollidieren werden [Sch06, GKZC03].



(a) Zeitschritt 0

(b) Zeitschritt 560

(c) Zeitschritt 760

(d) Zeitschritt 1860

(e) Zeitschritt 3090

(f) Zeitschritt 3890

Abbildung 2.2: Simulationsverlauf einer Kollision zweier Galaxien mit je 262144 Partikeln mit $\theta = 0{,}75$. Es sind nur die jeweils 87382 Partikel sichtbarer Materie der beiden Galaxien dargestellt. Die restlichen Partikel bestehen aus dunkler Materie und werden nicht angezeigt.

Die beiden Galaxien des Simulationslaufs aus Abbildung 2.2 hatten zum Zeitpunkt $t = 0$ einen Abstand von ca. $5 \cdot 10^5$ pc, dies entspricht in etwa der Entfernung zwischen Andromeda und der Milchstraße. Die Kollision erfolgte nach 560 Zeitschritten, was in physikalischen Einheiten eine Dauer von $1{,}36881 \cdot 10^9$ yr bedeuten würde. Die beiden simulierten Galaxien enthalten jeweils $3{,}35238 \cdot 10^{10}\,M_\odot$ sichtbare Masse und $1{,}34095 \cdot 10^{12}\,M_\odot$ Gesamtmasse. In der Literatur werden ähnliche Werte für die Massen von Andromeda und der Milchstraße angenommen [WEA10]. Für alle anderen Simulationsläufe, die im Zuge der vorliegenden Arbeit durchgeführt wurden, wurden ähnliche Anfangsbedingungen, wie für den oben beschriebenen verwendet. Ausschließlich die Anzahl von Partikeln wurde variiert.

2.2 Diskretisierung der Bewegungsgleichungen

Um N-Body-Probleme am Rechner simulieren zu können, müssen die kontinuierlichen Newton'schen Bewegungsgleichungen diskretisiert werden. Dazu wird das Zeitintervall $[t_{\text{Start}}, t_{\text{Ende}}]$, für das eine Lösung gesucht ist, in gleich große Teilintervalle, jeweils mit Abstand δt, zerlegt. Es wird dadurch ein Gitter auf der Zeitachse definiert und nur für die Gitterpunkte $t = t_{\text{Start}} + n \cdot \delta t$ werden die Differentialgleichungen dann ausgewertet.

Wir gehen im Folgenden davon aus, dass die Kräfte $\vec{F}_i$, die auf die einzelnen Partikel wirken, zum Zeitpunkt t gegeben sind. Zu den gegebenen Kräften lassen sich dann neue Partikelpositionen und -geschwindigkeiten nach unterschiedlichen Verfahren bestimmen. Mit Hilfe der Taylor-Entwicklung kann für eine gewählte Schrittweite δt eine Diskretisierung der Bewegungsgleichungen angegeben werden.

2.2.1 Zeitdiskretisierung mit Hilfe des Euler-Verfahrens

Werden bei der Taylorentwicklung alle Terme höherer Ordnung vernachlässigt, erhält man das einfachste Diskretisierungsverfahren, das auch als Euler-Verfahren bekannt ist.

$$\vec{x}_i(t + \delta t) = \vec{x}_i(t) + \delta t \cdot \vec{v}_i(t) \tag{2.8}$$

$$\vec{v}_i(t + \delta t) = \vec{v}_i(t) + \delta t \cdot \frac{\vec{F}_i(t)}{m_i} \tag{2.9}$$

Aufgrund des Vernachlässigens aller Terme höherer Ordnung handelt es sich um ein Diskretisierungsverfahren erster Ordnung. Der Diskretisierungsfehler liegt damit in der Größenordnung von $\mathcal{O}(\delta t)$, d.h. um den zeitlichen Diskretisierungsfehler zu halbieren, muss die Schrittweite δt ebenfalls halbiert werden. Genauere Diskretisierungsverfahren können daraus entstehen, dass auch Terme höherer Ordnung der Taylorentwicklung berücksichtigt werden.

2.2.2 Störmer-Verlet-Verfahren

Ein weit verbreitetes Zeitintegrationsverfahren höherer Ordnung ist das Störmer-Verlet-Verfahren mit einem Diskretisierungsfehler der Größenordnung $\mathcal{O}(\delta t^2)$. Es gibt von diesem Verfahren mehrere verbreitete Varianten. In den Gleichungen 2.10 und 2.11 ist das Geschwindigkeits-Störmer-Verlet-Verfahren angegeben.

$$\vec{x}_i(t+\delta t) = \vec{x}_i(t) + \delta t \cdot \vec{v}_i(t) + \frac{\delta t^2}{2} \cdot \frac{\vec{F}_i(t)}{m_i} \tag{2.10}$$

$$\vec{v}_i(t+\delta t) = \vec{v}_i(t) + \delta t \cdot \frac{\vec{F}_i(t) + \vec{F}_i(t+\delta t)}{m_i} \tag{2.11}$$

Der Nachteil des Geschwindigkeits-Störmer-Verlet-Verfahrens liegt darin, dass Kräfte bzw. Beschleunigungen aus zwei Zeitschritten bekannt sein müssen, um neue Partikelpositionen und -geschwindigkeiten bestimmen zu können.

Eine in exakter Arithmetik äquivalente Formulierung des Geschwindigkeits-Störmer-Verlet-Verfahrens kann mit dem Leapfrog-Verfahren nach den Gleichungen 2.12 und 2.13 angegeben werden.

$$\vec{v}_i\left(t+\frac{\delta t}{2}\right) = \vec{v}_i\left(t-\frac{\delta t}{2}\right) + \delta t \cdot \frac{\vec{F}_i(t)}{m_i} \tag{2.12}$$

$$\vec{x}_i(t+\delta t) = \vec{x}_i(t) + \delta t \cdot \vec{v}_i\left(t+\frac{\delta t}{2}\right) \tag{2.13}$$

Neben den oben vorgestellten Diskretisierungsverfahren existieren allerdings viele weitere Verfahren höherer Ordnung, z.B. verschiedene Predictor-Corrector-Verfahren, die zur Diskretisierung der Bewegungsgleichungen herangezogen werden können [GKZC03, BZBP09].

Nachdem wir für die Beschreibung der Diskretisierung davon ausgegangen sind, die Kräfte, die auf einzelne Partikel wirken, bereits zu kennen, soll in den folgenden Abschnitten beschrieben werden, wie die Berechnung der Kräfte algorithmisch realisiert werden kann und wie unter Verwendung der Kraftberechnung und Zeitdiskretisierung schließlich vollständige Simulationsalgorithmen entworfen werden können.

2.3 Algorithmische Ansätze zur Lösung des N-Body-Problems

Zur näherungsweisen Lösung verschiedener Klassen von N-Body-Problemen lassen sich unterschiedliche Algorithmen angeben, die sich in Hinblick auf Komplexität und Implementierungsaufwand stark unterscheiden können. Wir wollen nun zunächst eine naive Lösung vorstellen, die komplett ohne problemspezifische Optimierungen auskommt. Anschließend sollen in den Abschnitten 2.3.2 und 2.4 dann unterschiedliche optimierte Verfahren vorgestellt werden.

2.3.1 Naive Implementierung

Die einfachste Methode N-Body-Probleme zu simulieren, besteht darin, die zwischen jedem Paar von Körpern wirkenden Kräfte zu bestimmen, die Kräfte für jeden Körper aufzusummieren und die einzelnen Körper schließlich, den auf sie wirkenden Kräften entsprechend, zu verschieben. Die Verschiebung der Partikel erfolgt dann nach einem der in Abschnitt 2.2 vorgestellten diskretisierten Beschreibungen der Newton'schen Bewegungsgleichungen.

Bei dieser naiven Herangehensweise sind für jeden der N Körper die Krafteinwirkungen durch jeden der $N-1$ übrigen Körper zu berechnen, was zu einer Komplexität von $\mathcal{O}(N^2)$ führt und damit nur für sehr kleine Problemgrößen praktikabel ist.

Algorithmus 2.1 Der naive $\mathcal{O}(N^2)$-Algorithmus für N-Body-Probleme

```
t ← t_Start
while t < t_Ende do
    for all Partikel i do
        F_i ← 0
        for all Partikel j ≠ i do
            Kraft F_ij nach Gleichung 2.4 für gewähltes Potential bestimmen
            F_i ← F_ij + F_i
        end for
    end for
    Positionen und Geschwindigkeiten der Partikel aktualisieren
    t ← t + δt
end while
```

Eine Pseudocodeversion dieser naiven Lösung ist in Algorithmus 2.1 angegeben. Durch Ausnutzung von Symmetrieeigenschaften — nach dem Wechselwirkungsgesetz gilt $F_{ij} = -F_{ji}$ — lässt sich die Hälfte der Berechnungen zwar noch einsparen, an der asymptotischen Komplexität von $\mathcal{O}(N^2)$ ändert dies allerdings nichts.

Um nun Simulationen mit Partikelzahlen durchführen zu können, die für praktische Anwendungen erforderlich sind, um z.B. physikalische Modelle zu validieren, sind daher Verfahren nötig, die mit einer deutlich geringeren Laufzeitkomplexität auskommen.

2.3.2 Ausnutzung von Eigenschaften der Potentialfunktion

Je nach Problemstellung lassen sich Algorithmen entwerfen, deren Komplexität sich bis auf $\mathcal{O}(N)$ verringern lässt. Dazu unterscheidet man zunächst nach der Art der Potentiale, die die Kräfte auf die einzelnen Körper vermitteln. Wie bereits in Abschnitt 2.1 angesprochen, gibt es kurzreichweitige und langreichweitige Potentiale. Für beide können jeweils unterschiedliche optimierte Algorithmen angegeben werden.

Kurzreichweitige Potentiale

Werden in der Simulation ausschließlich kurzreichweitige Potentiale betrachtet, deren Betrag mit dem Abstand zwischen zwei Körpern sehr schnell gegen 0 konvergiert, kann recht einfach ein $\mathcal{O}(N)$-Algorithmus angegeben werden.

Man definiert dazu eine Größe r_{cut} — den Abschneideradius — und verwendet anstatt der ursprünglichen Potentialfunktion eine Näherungsfunktion, die für alle Abstände zwischen Körpern, die größer sind als r_{cut}, den Wert 0 annimmt. Partitioniert man das Simulationsgebiet zudem in gleichgroße Bereiche, mit Kantenlänge $\geq r_{cut}$, müssen Interaktionen für alle Partikel i nur mit anderen Partikeln innerhalb derselben Zelle und deren direkten Nachbarn berücksichtigt werden, da nur dort Partikel innerhalb des Abschneideradius' von Partikel i liegen können.

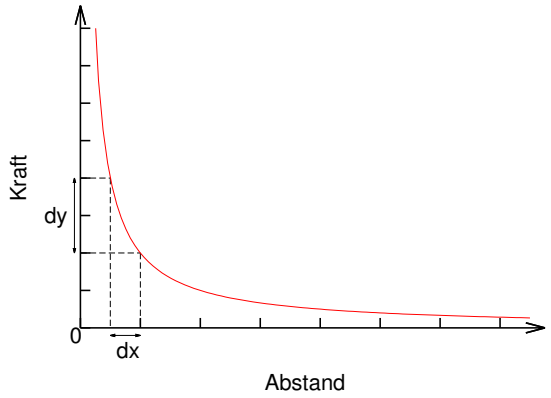
Die beschriebene Idee ist in Pseudocode in Algorithmus 2.2 abgefasst. Genauere Details zu diesem sogenannten *Linked-Cell-Algorithmus* und dessen Parallelisierung werden in [GKZC03] beschrieben.

Algorithmus 2.2 Pseudocode für Auswertung kurzreichweitiger Kräfte mit Abschneideradius r_{cut}

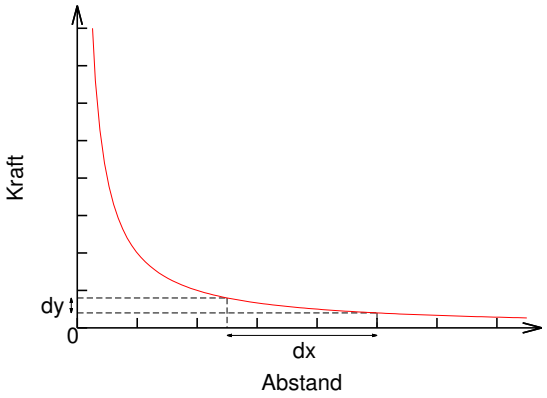
```
t ← t_Start
while t < t_Ende do
    for all Zellen z do
        for all Partikel i in Zelle z do
            F_i ← 0
            for all Partikel j ≠ i in ({z} ∪ direkte Nachbarzellen von z) do
                if ||x_i − x_j|| < r_cut then  // Partikel j innerhalb des Abschneideradius' von i
                    Kraft F_ij nach Gleichung 2.4 für gewähltes Potential bestimmen
                    F_i ← F_ij + F_i
                end if
            end for
        end for
    end for
    Positionen und Geschwindigkeiten der Partikel aktualisieren
    t ← t + δt
end while
```

Langreichweitige Potentiale

Werden in einer N-Body-Simulation hingegen langreichweitige Potentiale wie das Gravitationspotential betrachtet, würde das einfache Abschneiden der Potentialfunktion zu



(a) Wird ein Partikel im Nahfeld eines anderen Partikels nur wenig verschoben (dx), kann sich dies in einer deutlichen Veränderung der resultierenden Kraft (dy) niederschlagen.



(b) Wird ein Partikel im Fernfeld eines anderen Partikels verschoben (dx), bewirkt dies nur eine geringe Veränderung der resultierenden Kraft (dy).

Abbildung 2.3: Veränderung der Kraftwirkung (dy) auf ein Partikel (an $x = 0$), wenn das wechselwirkende Partikel verschoben wird (dx). Nach [GKZC03].

unvertretbar großen Abweichungen von einer exakten Lösung führen. Um dennoch optimierte Algorithmen angeben zu können, nutzt man eine spezielle Eigenschaft langreichweitiger Potentiale aus.

Betrachtet man das Potential zwischen zwei Partikeln, die nahe beieinander liegen, führt bereits eine kleine Änderung des Abstands zwischen beiden Partikeln zu einer großen Änderung des Potentials und der daraus resultierenden Kräfte. Für Paare von Partikeln, die dagegen einen großen Abstand voneinander besitzen, muss eine große Änderung des Abstands herbeigeführt werden, um eine Änderung des resultierenden Potentials zu bewirken. Abbildung 2.3 soll die beiden Sachverhalte graphisch verdeutlichen.

Alle Partikel, die, wie in Abbildung 2.3(a) dargestellt, nahe bei einem Partikel i liegen, werden zusammenfassend als dessen *Nahfeld* bezeichnet. Dagegen werden alle anderen, von i weiter entfernt liegenden Partikel als dessen *Fernfeld* bezeichnet, was Abbildung 2.3(b) veranschaulichen soll. Nah- und Fernfeld unterschiedlicher Partikel können dementsprechend auch deutlich unterschiedlich sein.

Algorithmisch werden nun bei der Kraftauswertung Nahfeld und Fernfeld aller Partikel unterschiedlich behandelt. Mit allen Partikeln, die sich im Nahfeld von i befinden, werden direkt Interaktionen, wie im naiven $\mathcal{O}(N^2)$-Algorithmus bestimmt. Eng benachbarte Partikelgruppen im Fernfeld von i dagegen können zu *Pseudopartikeln* oder *Multipolen* zusammengefasst werden. Dazu werden für jede Gruppe von Partikeln deren Massenschwerpunkt und Gesamtmasse bestimmt. Statt die Wechselwirkungen zwischen i und allen Partikeln der Gruppe zu bestimmen, wird dann näherungsweise auf Partikel-Pseudopartikel-Interaktionen

ausgewichen. Dadurch lässt sich die Berechnung einer sehr großen Anzahl von Interaktionen vermeiden.

Um diese Unterscheidung in Nah- und Fernfeld zu realisieren und Gruppen von Partikeln durch Pseudopartikel zu approximieren, wurden unterschiedliche baumbasierte Verfahren entworfen. Diese sollen im folgenden Abschnitt beleuchtet werden.

2.4 Baumbasierte Verfahren zur Lösung des N-Body-Problems

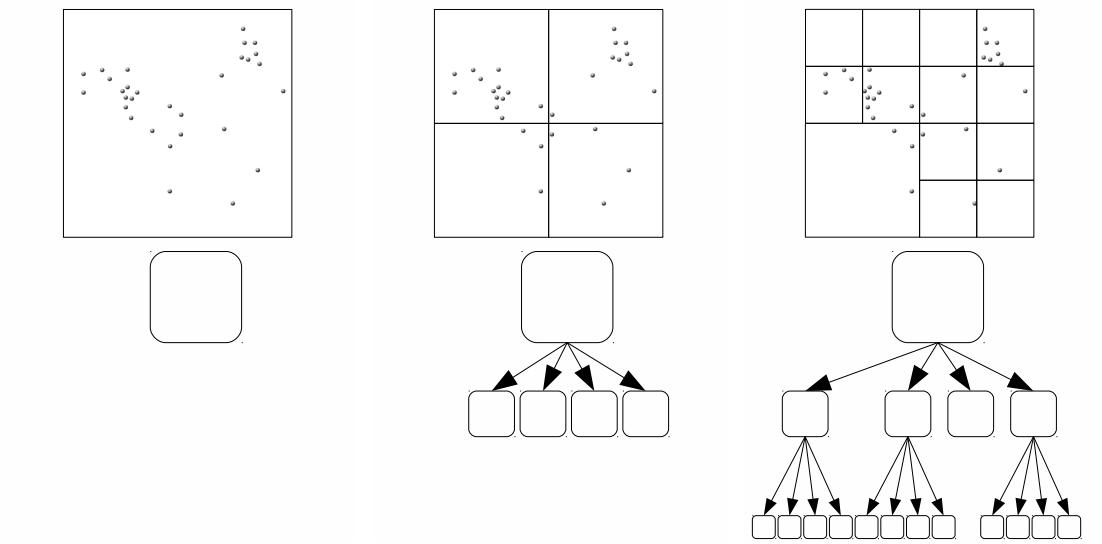
Allen baumbasierten Verfahren gemeinsam ist die namensgebende hierarchische Gebietszerlegung mittels Bäumen als Datenstrukturen. Durch die hierarchische Aufteilung des Simulationsgebiets wird eine effiziente Bestimmung von Nah- und Fernfeld von Partikeln ermöglicht. Die Komplexität der Kraftauswertung kann dann durch Approximation der Kräfte im Fernfeld von Partikeln deutlich reduziert werden. Die Details der Kraftauswertung unterscheiden sich allerdings zwischen den unterschiedlichen baumbasierten Verfahren. Auf die Details zur Kraftauswertung beim Barnes-Hut-Verfahren werden wir in Abschnitt 2.4.3 eingehen. Als Baumstrukturen werden, wie auch für die vorliegende Arbeit, zumeist Octrees eingesetzt, auf die daher im folgenden Abschnitt genauer eingegangen wird.

2.4.1 Hierarchische Gebietsunterteilung mittels Octrees

Octrees werden eingesetzt, um das Simulationsgebiet rekursiv in immer kleiner werdende Raumabschnitte zu zerlegen. Die Zerlegung beginnt auf der obersten Ebene mit einem Würfel, der das gesamte Simulationsgebiet und damit alle Partikel umfasst. Der Würfel wird anschließend in acht gleichgroße, direkt aneinander angrenzende Kinderwürfel zerlegt, die zusammen wieder denselben Raum ausfüllen wie der Elternwürfel. Dieser Prozess der Zerlegung von Würfeln in jeweils acht Kinder, auch *Oktanten* genannt, wird rekursiv fortgesetzt, bis die Menge der im Simulationsgebiet eines Würfels enthaltenen Partikel, eine vorgegebene Höchstzahl von Partikeln nicht mehr überschreitet. Die Kinderwürfel entstehen dabei immer durch Halbierung der Kantenlänge ihrer Eltern.

Die Datenstruktur zur Repräsentation dieser rekursiven Zerlegung des Raums durch Würfel ist ein Baum. Jeder Würfel wird mit einem Knoten des Baums identifiziert, der verschiedene Attribute trägt. In Abbildung 2.4 ist das Prinzip der Octrees anhand eines Quadtrees — der Entsprechung des Octrees in zwei Dimensionen — veranschaulicht, der von 2.4(a) bis 2.4(e) mit steigenden Unterteilungstiefen dargestellt ist.

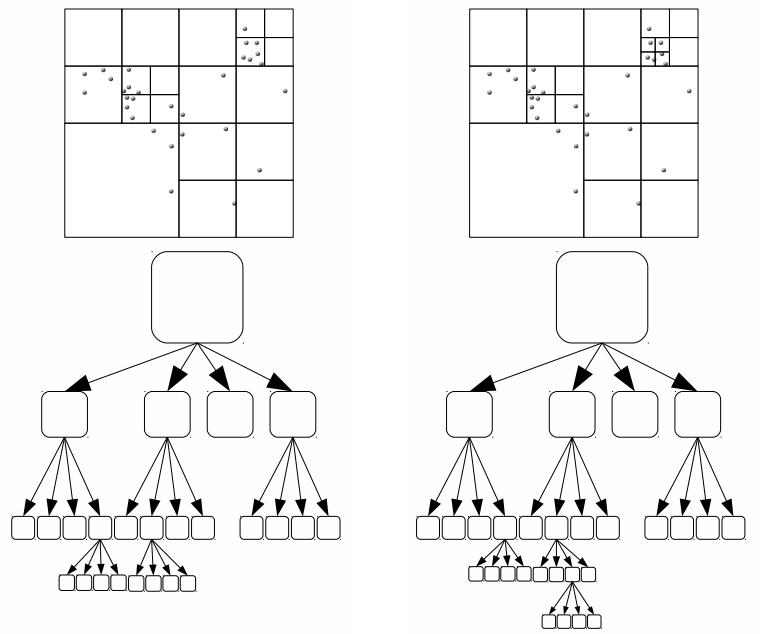
Wie der Aufbau von Octree-Strukturen für gegebene Simulationsprobleme algorithmisch realisiert werden kann, welche Attribute die einzelnen Knoten tragen und wie mit Hilfe dieser Baumstrukturen vollständige, optimierte Simulationsalgorithmen entworfen werden können, soll in den folgenden Abschnitten beschrieben werden.



(a) Nur Wurzelknoten

(b) Wurzel und erstes Level

(c) Quadtree mit Unterteilung bis zum zweiten Level



(d) Quadtree bis zum dritten Level

(e) Vollständiger Baum

Abbildung 2.4: Verteilung von Partikeln im Simulationsgebiet und hierarchische Gebietsunterteilung mittels eines Quadtrees mit höchstens vier Partikeln pro Blatt

2.4.2 Allgemeiner Ablauf von baumbasierten N-Body-Algorithmen

Der Ablauf von baumbasierten Verfahren zur näherungsweisen Lösung von N-Body-Problemen lässt sich allgemein in mehrere Teilprobleme untergliedern. Algorithmus 2.3 soll einen Eindruck davon vermitteln, wie ein solches allgemeines baumbasiertes Verfahren grundsätzlich abläuft.

Algorithmus 2.3 Ein allgemeiner Baumalgorithmus zur Lösung des N-Body-Problems

```
t ← t_Start
while t < t_Ende do
    Aufbauen der Baumstruktur der hierarchischen Gebietszerlegung
    Berechnung der Pseudopartikel für die Baumknoten
    Berechnung der Kräfte, die auf die einzelnen Partikel wirken
    Zeitintegration der Geschwindigkeiten und Aktualisierung der Partikelpositionen
    t ← t + δt
end while
```

In den folgenden Abschnitten soll nun darauf eingegangen werden, wie sich die einzelnen Teilschritte des obigen Algorithmus' berechnen lassen.

Aufbauen der rekursiven Baumstruktur

Um die Geometrie des Würfels für den Wurzelknoten bestimmen zu können, müssen zunächst einmal Minimum und Maximum aller Partikelkoordinaten bestimmt werden. Daraus werden dann der Mittelpunkt und die Kantenlänge des Würfels bestimmt. Blätter des Baums enthalten als Attribute die in ihnen enthaltenen Partikel. Innere Knoten dagegen besitzen als Attribute jeweils Zeiger auf ihre acht direkten Kinderknoten.

Der Aufbau der Baumstruktur erfolgt durch sukzessives Einfügen der Partikel ausgehend vom Wurzelknoten über einen rekursiven Abstieg zu den Blättern des Baums. Dabei wird in jedem inneren Knoten anhand der Lage des Partikels im zugehörigen Würfel entschieden, in welchen Oktanten abgestiegen werden muss. Der Abstieg in Kinderknoten im Baum wird fortgesetzt, bis ein Blatt erreicht wird. Enthält dieses Blatt beim Einfügen eines Partikels bereits die maximal zulässige Anzahl von Partikeln, wird das Blatt zu einem inneren Knoten. Dazu werden für den Knoten acht neue Blätter mit jeweils halbierter Kantenlänge erzeugt und die Partikel entsprechend ihrer Lage in diese neuen Blätter eingeordnet.

Das Einfügen kann dabei mittels eines rekursiven Algorithmus' bewerkstelligt werden, lässt sich aber auch iterativ lösen. In Algorithmus 2.4 ist die rekursive Variante zu sehen. Zusätzliche Attribute für die Knoten des Baums sind die Pseudopartikeldaten, mit deren Hilfe später die Kraftauswertung effizient approximiert werden kann, deren Berechnung im folgenden Abschnitt beschrieben wird.

Algorithmus 2.4 Rekursiver Baumaufbau durch sukzessives Einfügen von Partikeln

```
function BAUMAUFBAU
    Abmessungen des Simulationsgebiets bestimmen und
    Wurzelknoten erzeugen
    for all Partikel i do
        FügePartikelEin (i, Wurzelknoten)
    end for
end function

function FÜGEPARTIKELEIN(Partikel i, Knoten k)
    if IstBlatt (k) then
        if HatPlatz (k) then
            k.Partikel ← {i} ∪ k.Partikel
        else
            ErzeugeKinder (k)
            for all Partikel j ∈ {i} ∪ k.Partikel do      // Partikel auf neue Kinder verteilen
                FügePartikelEin (j, k)
            end for
        end if
    else
        l ← Kindknoten von k für x_i                      // Abstieg in den nächstkleineren Oktanten
        FügePartikelEin (i, l)
    end if
end function
```

Berechnung der Pseudopartikel

Die Berechnung der Pseudopartikel für die einzelnen Baumknoten erfolgt am einfachsten durch eine rekursive Traversierung der Baumstruktur. Die Rekursion läuft zunächst bis zu den Blättern, deren Pseudopartikel als erstes zu berechnen sind. Beim Wiederaufstieg werden anschließend die Pseudopartikel der inneren Knoten aus den Werten der Pseudopartikel der Kinderknoten berechnet. Algorithmus 2.5 soll dieses Vorgehen veranschaulichen.

Der Baumaufbau und die Bestimmung der Pseudopartikel sind den baumbasierten Verfahren gemein. Die Verfahren unterscheiden sich allerdings deutlich in der Art, wie die auf die Partikel wirkenden Kräfte bestimmt werden. Im nächsten Abschnitt soll zunächst auf die Kraftberechnung des Barnes-Hut-Algorithmus eingegangen werden.

Algorithmus 2.5 Berechnung der Pseudopartikel für Baumknoten

```
function PseudopartikelBerechnen
    PseudopartikelBerechnen (Wurzelknoten)
end function

function PseudopartikelBerechnen(Knoten k)
    if IstBlatt (k) then
        Massenschwerpunkt, Gesamtmasse und Pseudopartikeldurchmesser bestimmen
    else
        for all Kinderknoten l von k do
            PseudopartikelBerechnen (l)
        end for
        Massenschwerpunkt, Gesamtmasse und Pseudopartikeldurchmesser bestimmen
    end if
end function
```

2.4.3 Barnes-Hut-Algorithmus

Im Gegensatz zum Klassischen N-Body-Algorithmus, der keinerlei Problemeigenschaften bezüglich der Verteilung der Partikel im Raum verwendet, wird beim Barnes-Hut-Algorithmus diese Information genutzt, um die Rechenkomplexität zu reduzieren. Dazu wird das Simulationsgebiet räumlich rekursiv unterteilt und für jedes Gebiet ein Pseudopartikel berechnet, das durch die Summe der Massen der einzelnen Partikel, den Massenschwerpunkt als Position und einen Pseudopartikeldurchmesser definiert ist. Durch diese Pseudopartikel, mit deren Hilfe die Komplexität der Kraftauswertung reduziert werden kann, werden jeweils alle Einzelpartikel eines Teilgebiets zusammengefasst.

Bestimmen der Kräfte, die auf einzelne Partikel wirken

Für jedes Partikel wird eine Baumtraversierung durchgeführt, während der alle auf das Partikel einwirkenden Kräfte berechnet und aufsummiert werden. In Abhängigkeit des Abstands des Partikels von den Pseudopartikeln der Baumknoten und deren Durchmessern müssen dann Knoten entweder geöffnet und deren Kinder traversiert werden oder die Kraft wird zwischen Partikel und Pseudopartikel ausgewertet und die Kinder des Knotens müssen dann nicht weiter berücksichtigt werden. In Algorithmus 2.6 wird die Kraftauswertung für einzelne Partikel ausgehend vom Wurzelknoten beschrieben.

Für die Entscheidung, ob ein Pseudopartikel die erforderliche Genauigkeit erfüllt, um alle Partikel-Partikel-Interaktionen eines Baumknotens durch eine einzige Partikel-Pseudopartikel-Interaktion zu ersetzen, wird das sogenannte θ-Kriterium (auch MAC, multi-

Algorithmus 2.6 Traversierung der Baumstruktur zur Kraftauswertung für ein Partikel

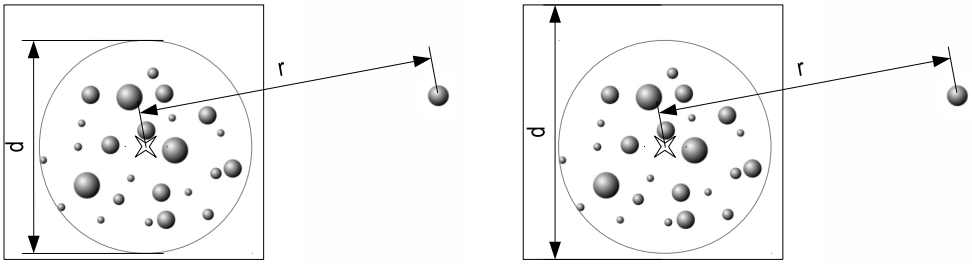
```
function KraftauswertUng(Partikel i)
    F_i ← 0
    KraftauswertungRekursiv (i, Wurzelknoten)
    return F_i
end function

function ThetaKriteriumErfüllt(Abstand r, Pseudopartikeldurchmesser d, θ)
    return { wahr, wenn r/d ≥ θ
           { falsch, sonst
end function

function KraftauswertungRekursiv(Partikel i, Baumknoten k)
    if ThetaKriteriumErfüllt (Abstand (i, k), k.Durchmesser, θ) then
        j ← k.Pseudopartikel
        F_i ← F_i + F_ij                    // Partikel-Pseudopartikel-Interaktion berechnen
    else
        if IstBlatt (k) then                // In den Blättern sind die Partikel enthalten
            for all Partikel j ≠ i in k do
                F_i ← F_i + F_ij            // Partikel-Partikel-Interaktion berechnen
            end for
        else                                // k ist ein innerer Knoten
            for all Kinderknoten l von k do
                KraftauswertungRekursiv (i, l)   // Rekursiver Abstieg zu Kinderknoten
            end for
        end if
    end if
end function
```

pole acceptance criterion, genannt) betrachtet. Im Algorithmus wird dieses durch den Aufruf der Funktion ThetaKriteriumErfüllt ausgewertet. Ist der Abstand r zwischen Partikel und Pseudopartikel im Verhältnis zum Pseudopartikeldurchmesser d ausreichend groß, wird die Partikel-Pseudopartikel-Interaktion als genau genug akzeptiert, wobei der Parameter θ die Genauigkeit steuert.

Um den Aufwand für die Auswertung des θ-Kriteriums und die Berechnung der Pseudopartikel zu reduzieren, werden allerdings statt des tatsächlichen Durchmessers eines Pseudopartikels meist Näherungen eingesetzt. Abbildung 2.5(a) zeigt die exakte Bestimmung von Abstand zwischen Partikel und Pseudopartikel und Pseudopartikeldurchmesser



(a) Durchmesser entspricht maximalem Abstand eines Partikels vom Schwerpunkt

(b) Durchmesser durch Kantenlänge des Octree-Würfels angenähert

Abbildung 2.5: Unterschiedliche Varianten zur Bestimmung des Pseudopartikeldurchmessers d zur Auswertung des θ-Kriteriums

und Abbildung 2.5(b) die näherungsweise Angabe des Pseudopartikeldurchmessers durch die Kantenlänge des Baumknotens.

Durch die näherungsweise Berechnung der Kräfte zwischen Partikeln durch Partikel-Pseudopartikel-Wechselwirkungen kann die Komplexität des ursprünglichen $\mathcal{O}(N^2)$-Algorithmus auf $\mathcal{O}(N \log N)$ reduziert werden [GKZC03].

Abbildung 2.6 zeigt den Verlauf der Simulation aus Abschnitt 2.1.2 auf Seite 17 in Bezug auf die Anzahl der Interaktionen. Es ist dabei für jeden der 4000 Zeitschritte die durchschnittliche Anzahl von Interaktionen pro Partikel dargestellt. Im Vergleich dazu würde ein $\mathcal{O}(N^2)$-Algorithmus unter Ausnutzung des Wechselwirkungsgesetzes in jedem Zeitschritt pro Partikel 262143 Kräfte zwischen Paaren von Partikeln bestimmen müssen. Abbildung 2.6 lässt gut erkennen, dass die Gesamtzahl von Interaktionen stark von der aktuellen Partikelverteilung einer Simulation abhängig ist. Zu den Zeitpunkten der Kollisionen zwischen beiden Galaxien sind deutlich mehr Interaktionen zu berechnen, als zu Zeitpunkten, in denen beide Galaxien getrennt voneinander im Simulationsgebiet liegen.

Genauigkeit des Barnes-Hut-Verfahrens

Neben den in Abschnitt 2.2 beschriebenen Diskretisierungsverfahren, die unterschiedliche Diskretisierungsfehler in Abhängigkeit der Schrittweite δt in die Simulation einführen, wird die Genauigkeit des Barnes-Hut-Algorithmus' maßgeblich durch die Wahl von θ bestimmt.

Je nachdem welche Werte für θ eingesetzt werden, variiert die Genauigkeit des Verfahrens, wobei mit einem Wert von $\theta = 0$ das Verfahren in Hinblick auf die Genauigkeit zu einem klassischen $\mathcal{O}(N^2)$-Algorithmus wird. Je größer der Wert θ gewählt wird, desto mehr Interaktionen zwischen Partikeln werden durch Partikel-Pseudopartikel-Wechselwirkungen ersetzt und desto ungenauer , aber dafür schneller, wird das Verfahren im Vergleich zum

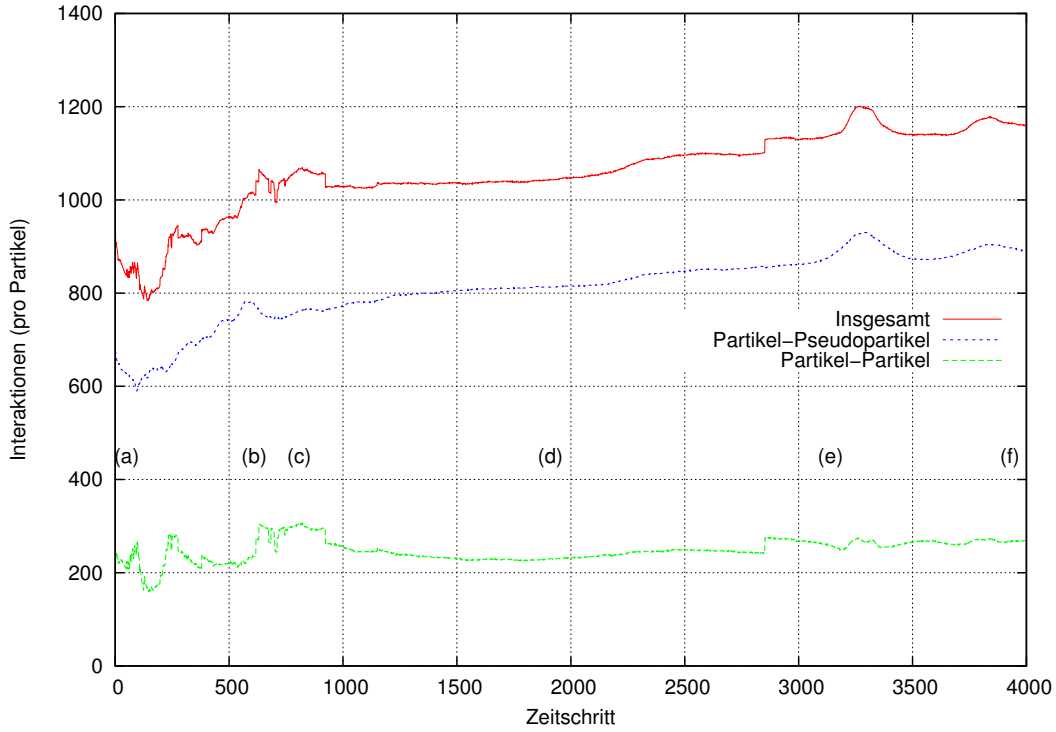


Abbildung 2.6: Anzahl von Interaktionen pro Partikel und Zeitschritt im Verlauf der Simulation aus Abbildung 2.2 aufgeschlüsselt nach Partikel-Partikel- und Partikel-Pseudopartikel-Interaktionen. Die Zeitschritte, die in Abbildung 2.2 auf Seite 18 abgebildet sind, sind mit (a)-(f) hervorgehoben. Die Simulation erfolgte mit $\theta = 0{,}75$ und einer Gesamtzahl von 524288 Partikeln.

$\mathcal{O}(N^2)$-Algorithmus. Genau genommen lässt sich eine Zeitkomplexität von $\mathcal{O}(\frac{N \log N}{\theta^3})$ in Abhängigkeit vom Steuerparameter θ für den Gesamtalgorithmus nachweisen [GKZC03]. In Abbildung 2.7 ist das Laufzeitverhalten einer Implementierung des Barnes-Hut-Verfahrens für unterschiedliche Werte von θ und unterschiedliche Problemgrößen dargestellt.

2.4.4 Schnelle Multipol-Methode

Werden Kräfte partikelweise berechnet und anhand der Unterscheidung zwischen Nah- und Fernfeld Partikel-Partikel-Wechselwirkungen näherungsweise durch Partikel-Pseudopartikel-Wechselwirkungen ersetzt, kann, wie oben beschrieben, die Komplexität von N-Body-Problemen von $\mathcal{O}(N^2)$ auf $\mathcal{O}(N \log N)$ reduziert werden. Die logische Fortsetzung dieses Gedankens besteht nun darin, anstatt Partikel-Pseudopartikel-Wechselwirkungen zu betrachten, Wechselwirkungen zwischen Paaren von Pseudopartikeln zu berechnen. Durch diesen

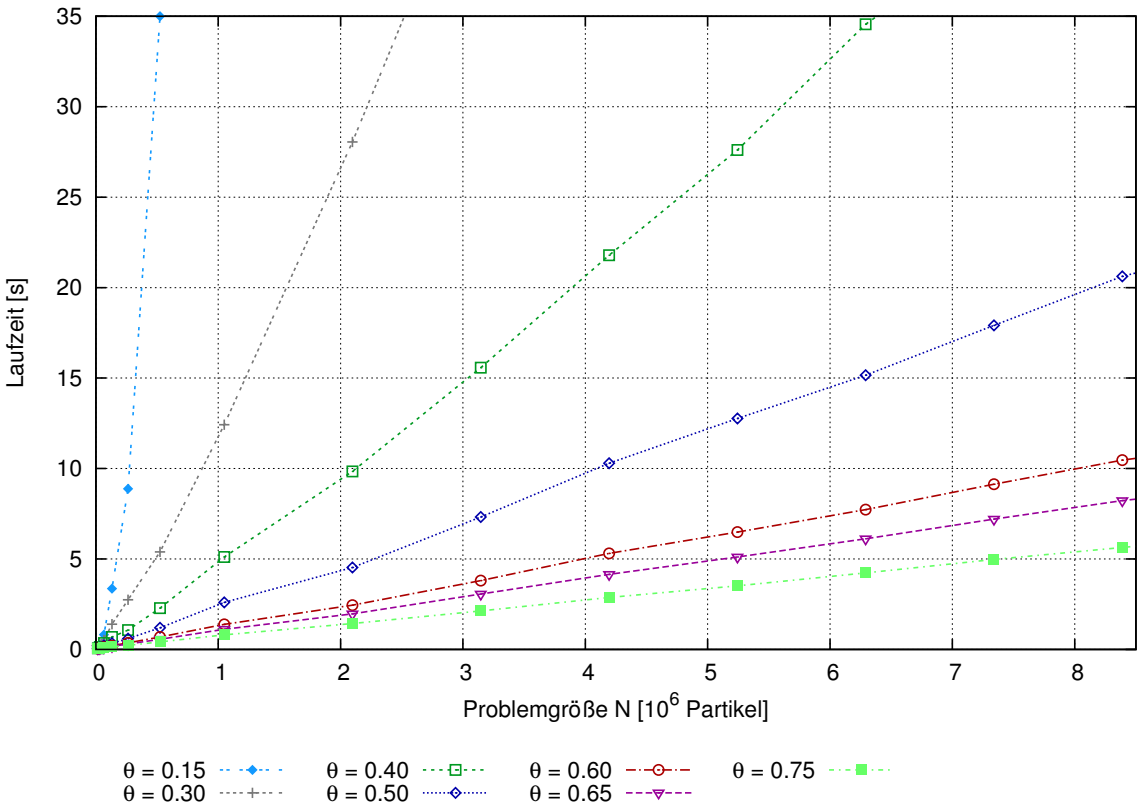


Abbildung 2.7: Laufzeiten des Barnes-Hut-Verfahrens für unterschiedliche Größen von θ für einen Zeitschritt, berechnet jeweils aus dem Mittelwert von 16 Zeitschritten.

Ansatz der sogenannten *Schnellen Multipol-Methode* lässt sich die Komplexität noch weiter reduzieren. Es resultiert ein Algorithmus der Zeitkomplexität $\mathcal{O}(N)$ [GKZC03, GR87].

3 Parallele Programmierung

Aktuelle Rechner besitzen zumeist Prozessoren, die über mehrere Prozessorkerne verfügen, die parallel Aufgaben bearbeiten können. Meistens teilen sich diese Prozessorkerne große Teile der Speicherhierarchie und können so ohne großen Aufwand dazu eingesetzt werden, gemeinsam Probleme zu lösen, indem jeder Prozessorkern ein — in Bezug auf die Ausführungszeit — möglichst gleichgroßes Teilproblem löst.

Mit der Einführung frei programmierbarer Graphikkarten, die zur Lösung allgemeiner Berechnungen eingesetzt werden können, sind zusätzlich weitere Recheneinheiten verfügbar, die zur Beschleunigung von Programmen eingesetzt werden können.

Es soll nun in Abschnitt 3.1 zunächst beschrieben werden, nach welchen Kriterien sich parallele Rechnerarchitekturen klassifizieren lassen, bevor in den Abschnitten 3.2 und 3.3 genauer auf die verwendeten Prozessoren und Graphikkarten sowie deren Programmierung eingegangen wird. Anschließend wollen wir uns in den Abschnitten 3.4 und 3.5 der Frage zuwenden, wie das Laufzeitverhalten unterschiedlicher paralleler Programme auf verschiedenen Plattformen erfasst und miteinander verglichen werden kann. Zuletzt werden in Abschnitt 3.6 die beiden für die Ausführung von Simulationen eingesetzten Testsysteme vorgestellt.

3.1 Klassifizierung paralleler Rechnerarchitekturen

Nach Flynn lassen sich Rechnerarchitekturen danach unterscheiden, ob die einzelnen Recheneinheiten jeweils über eigene Daten- und Befehlsströme verfügen oder ob diese von den Recheneinheiten gemeinsam genutzt werden. Es werden so vier verschiedene Rechnermodelle denkbar, die als **SISD** (single instruction, single data), **SIMD** (single instruction, multiple data), **MISD** (multiple instruction, single data) und **MIMD** (multiple instruction, multiple data) bezeichnet werden.

3.1.1 Beschreibung der Architekturmodelle der Klassifikation nach Flynn

Das einfachste Architekturmodell der Flynn'schen Klassifikation ist das **SISD**-Modell. SISD-Rechner entsprechen einem von-Neumann-Rechner, der genau einen Prozessor besitzt und über je einen Programm- und Datenstrom verfügt.

Das **SIMD**-Modell beschreibt Rechnerklassen, die zwar über mehrere Prozessoren verfügen, allerdings auf diesen in jedem Taktzyklus nur dieselbe Operation synchron ausführen können, da sie nur über einen gemeinsamen Befehlsstrom verfügen. Dagegen hat jeder Prozessor einen eigenen Datenspeicher, sodass die einzelnen Prozessoren auf unterschiedlichen Daten operieren können. Dadurch, dass alle Prozessoren synchron dieselbe Operation ausführen, müssen if-then-else-Anweisungen immer in zwei Teile zerlegt ausgeführt werden. Zunächst wird der if-Zweig und erst anschließend der else-Zweig bearbeitet. Bei komplexen Programmen, deren Verlauf stark von den Eingabedaten abhängt und die viele bedingte Anweisungen beinhalten, kann dies zu sehr schlechter Auslastung der einzelnen Prozessoren führen.

MISD-Rechner sind Rechner, deren mehrere Prozessoren über einen gemeinsamen Datenstrom aber getrennte Befehlsströme verfügen.

Zur Klasse der **MIMD**-Rechner werden all jene Rechner gezählt, die über mehrere Prozessoren verfügen, von denen jeder mit einem eigenen Daten- und Befehlsstrom versorgt wird. Es kann also von allen Prozessoren ein unterschiedlicher Programmverlauf auf unterschiedlichen Eingabedaten verfolgt werden. Die meisten aktuellen Parallelrechner lassen sich dieser Klasse zuordnen [RR07].

Neben den eben beschriebenen vier Architekturmodellen sind allerdings auch Mischformen unterschiedlicher Modelle denkbar. So wird z.B. mit dem **MSIMD**-Modell (multiple SIMD) ein Architekturmodell beschrieben, das aus mehreren SIMD-Prozessoren besteht, die untereinander wie ein MIMD-Rechner mit unterschiedlichen Befehlsströmen operieren können [JD11].

Die Architekturmodelle der Flynn'schen Klassifikation lassen sich noch feiner in Klassen unterteilen, je nachdem, wie die Speicherorganisation realisiert ist. Darauf soll im folgenden Abschnitt eingegangen werden.

3.1.2 Unterscheidung paralleler Architekturmodelle nach deren Speicherorganisation

Prozessoren können sich entweder denselben Speicher direkt teilen oder jeder Prozessor verfügt über eigenen Speicher, auf den nur vom jeweiligen Prozessor direkt zugegriffen werden kann. Die Prozessoren sind dann über ein Netzwerk verbunden, über welches Daten ausgetauscht werden können.

Die Speicheranbindung der einzelnen Prozessoren hat starken Einfluss auf die Speicherlatenzen und führt deshalb auch dazu, dass sich die Algorithmen, die für die jeweiligen Parallelrechnermodelle entworfen werden, deutlich voneinander unterscheiden können.

Parallelrechner mit physikalisch verteiltem Speicher

Bei physikalisch verteiltem Speicher kann nur auf Daten, die im lokalen Speicher eines Prozessors liegen, von diesem direkt zugegriffen werden. Daten die im Speicher anderer Prozessoren abgelegt sind, müssen explizit ausgetauscht werden, wenn sie für den Programmverlauf benötigt werden. Dadurch ergibt sich eine starke Abhängigkeit von der Geschwindigkeit des Verbindungsnetzwerks, über das die unterschiedlichen Recheneinheiten Daten übertragen.

Rechner mit physikalisch gemeinsamem Speicher

Bei Rechnern mit mehreren Prozessoren, die über gemeinsamen Speicher verfügen, stehen jedem Prozessor dieselben Daten direkt zur Verfügung. Alle Prozessoren teilen sich also die Daten und auch einen gemeinsamen Adressraum, sodass auch Zeigerstrukturen parallel bearbeitet werden können. Hier ist allerdings darauf zu achten, dass kein Prozessor Daten überschreibt, die von einem anderen Prozessor berechnet wurden. Um dies zu verhindern, können gemeinsame Variablen zur Kommunikation und Koordination der einzelnen Prozessoren untereinander eingesetzt werden. Darauf wird in Abschnitt 3.2.3 genauer eingegangen werden.

Aus Sicht des Programmierers lassen sich auch Mischformen aus Systemen mit verteiltem und gemeinsamem Speicher realisieren. Rechner mit virtuell gemeinsamem Speicher können programmiert werden, als verfügten die Prozessoren über gemeinsamen Speicher. Tatsächlich handelt es sich allerdings um Systeme mit verteiltem Speicher, der per Software in einem gemeinsamen Adressraum zusammengefasst wird. Wird auf einen Speicherbereich zugegriffen, der in einem fremden Speicherbereich liegt, werden die Daten im Hintergrund über das Verbindungsnetzwerk transferiert, um Speicheranfragen zu befriedigen. Es können so für unterschiedliche Speicheranfragen deutlich unterschiedliche Antwortzeiten auftreten, weshalb bei solchen Rechnern auch von **NUMA**-Systemen (non-uniform memory access) gesprochen wird.

Für die vorliegende Arbeit wurden ausschließlich Parallelrechner mit physikalisch gemeinsamem Speicher eingesetzt, auf deren Aufbau und Programmierung in den folgenden Abschnitten eingegangen werden soll.

3.2 Multicore-Systeme mit physikalisch gemeinsamem Speicher

Die Entwicklung der letzten Jahre hat gezeigt, dass Leistungssteigerungen allein durch Erhöhung des Prozessortaktes, aufgrund des gleichzeitig steigenden Stromverbrauchs und der damit verbundenen Wärmeentwicklung, nicht mehr möglich sind. Stattdessen werden

heute meist mehrere Prozessorkerne auf einem Prozessorchip untergebracht, die nach dem in Abschnitt 3.1.1 beschriebenen MIMD-Modell arbeiten. Die Prozessoren teilen sich dabei physikalisch gemeinsamen Speicher und einen gemeinsamen Adressraum.

Aufgrund der Tatsache, dass nicht mehr nur der Prozessortakt, sondern insbesondere die Anzahl der Recheneinheiten wächst, kann die theoretisch mögliche Rechenleistung aktueller Computer nur dann voll ausgeschöpft werden, wenn parallele Programmiertechniken zum Einsatz kommen und alle Prozessorkerne möglichst gut ausgelastet werden.

Neben der Prozessorentwicklung wurde auch der Hauptspeicher ständig weiterentwickelt. Zum einen wurde die Größe des Speichers stetig erhöht und zum anderen wurden die Zugriffszeiten immer weiter verringert. Allerdings erfolgte die Verringerung der Zugriffszeit auf den Hauptspeicher in deutlich kleineren Schritten als die Entwicklung der Rechenleistung von Prozessoren. Während z.B. für einen Intel i486 um 1990 Hauptspeicherzugriffe noch in 6-8 Taktzyklen befriedigt werden konnten, sind heutzutage mehrere hundert Taktzyklen Wartezeit üblich, bis angeforderte Daten aus dem Hauptspeicher im Prozessor verfügbar sind [RR07]. Speicherzugriffe werden so leicht zum Flaschenhals für eine effiziente Programmausführung.

3.2.1 Einsatz von Cache-Speichern zur Reduzierung der mittleren Zugriffszeit

Um Speicherzugriffe im Mittel zu beschleunigen, können Cache-Speicher eingesetzt werden, die deutlich schnellere Zugriffszeiten besitzen als der Hauptspeicher. Die Cache-Speicher werden zwischen Prozessor und Hauptspeicher eingefügt und bilden Teile des Hauptspeichers ab, wobei die Cache-Speicher deutlich weniger Speicherplatz besitzen als der Hauptspeicher.

Greift der Prozessor auf Daten zu, die im Cache verfügbar sind, können Speicheranfragen in wenigen Taktzyklen aus dem Cache beantwortet werden. Sind die Daten dagegen nicht im Cache abgelegt, müssen sie mit entsprechend längerer Zugriffszeit aus dem Hauptspeicher abgerufen werden.

Besonders schnelle Caches liegen meist direkt auf der Chipfläche des Prozessors, um möglichst kurze Leitungswege zu ermöglichen. Meist werden mehrere Stufen von Cache-Speichern zwischen Prozessor und Hauptspeicher eingefügt, sodass ganze Hierarchien von Speichern entstehen.

Aufbau von Speicherhierarchien aktueller Multicore-Systeme

In aktuellen Multicore-Systemen sind meist mehrstufige Cache-Hierarchien zu finden, wobei die Caches auf der obersten Ebene meist lokal, ein Cache pro Prozessorkern, direkt auf der Chipfläche liegen. L1-Caches, die die oberste Ebene der Hierarchie bilden, sind die

schnellsten und kleinsten Caches und umfassen in der Regel nur wenige Kilobytes Speicher. Je weiter man in der Hierarchie absteigt, desto größer aber dafür langsamer werden die Speicher. Zudem sind nur die Caches in den oberen Ebenen pro Kern lokal verfügbar. Die unteren Ebenen der Speicherhierarchie werden von allen Kernen gemeinsam verwendet.

Die Effektivität des Caching lässt sich durch den Entwurf von Algorithmen und Datenstrukturen stark beeinflussen.

Effiziente Nutzung des Caching durch räumliche und zeitliche Lokalität der Speicherzugriffe

Um Cache-Hierarchien besonders effizient ausnutzen zu können, sollten Speicherzugriffe stets so organisiert werden, dass möglichst in aufeinanderfolgenden Taktzyklen auch auf direkt benachbarte Speicherzellen zugegriffen wird. Man spricht dabei von der *räumlichen Lokalität* der Speicherzugriffe. Wird dagegen auf ein und dieselbe Speicherzelle mehrfach in kurzer Abfolge zugegriffen, wird von *zeitlicher Lokalität* gesprochen.

Beide Arten der Lokalität der Speicherzugriffe können die Programmausführung deutlich beschleunigen, da die Wahrscheinlichkeit, dass sich der entsprechende Speicherbereich, auf den zugegriffen werden soll, bereits im Cache befindet, sehr groß ist. Programme mit unregelmäßigen Speicherzugriffsmustern können dagegen nur deutlich weniger effizient ausgeführt werden, da sehr viel häufiger Wartezeiten zum Nachladen von Daten aus dem Hauptspeicher auftreten.

Cache-Kohärenz und False sharing

Werden Speicherbereiche von mehreren Prozessoren gleichzeitig in deren lokalen Caches gehalten, muss sichergestellt werden, dass alle Prozessoren stets die aktuellen Daten erhalten. Dies kann hardwareseitig durch sogenannte Cache-Kohärenz-Protokolle realisiert werden. Greift einer der Prozessoren schreibend auf Daten zu, müssen die Speicher aller anderen Prozessoren konsistent gehalten werden, indem deren Cache-Speicher als nicht mehr aktuell markiert und vor dem nächsten Zugriff nachgeladen werden.

Cache-Speicher bilden die höheren Ebenen der Speicherhierarchie nicht Byte-weise ab sondern in größeren Blöcken. Ändert ein einziger Thread Daten eines Cache-Blocks, muss dieser Block bei allen anderen Prozessoren aktualisiert werden, selbst wenn diese nur auf getrennte Bereiche des Blocks zugreifen, die nicht geändert wurden. Dieses sogenannte *False sharing* kann die Effizienz von Caching stark beeinträchtigen.

3.2.2 Multithreading und Hyperthreading

Neben dem Einsatz von Caches lassen sich auch auf andere Weise Speicherzugriffszeiten effektiv verstecken. Durch Multithreading auf einzelnen Prozessoren kann die Prozessorauslastung bei Anwendungen, die häufig auf langsame Speicherzugriffe angewiesen sind, stark erhöht werden. Dies gilt insbesondere für Zugriffe auf Daten von der Festplatte oder auf Daten, die über ein Netzwerk übertragen werden müssen und mit sehr hohen Latenzen verbunden sind. In [BKWb98] wird z.B. eine Multithreadingbibliothek für Parallelrechner mit verteiltem Speicher entworfen, die nach diesem Prinzip vorgeht. Für Anwendungen mit wenigen langsamen Speicherzugriffen bewirkt Multithreading allerdings keine Beschleunigung, da Kontextwechsel zwischen Threads durch das Betriebssystem mit einem gewissen Aufwand verbunden sind.

Eine hardwarebasierte Möglichkeit durch Multithreading Latenzen zu verstecken, besteht darin, alle Bestandteile, die den Prozessorzustand abspeichern, zu duplizieren. Es können so gleichzeitig zwei Prozessorzustände gespeichert werden. Der physikalische Prozessor wird dem Betriebssystem dann als Zusammenschluss mehrerer *logischer Prozessoren* angezeigt, die bereitstehen, gleichzeitig unterschiedliche Threads auszuführen. Alle anderen Ressourcen des Prozessorkerns — lokale Caches, Steuer- und Rechenwerke — werden nicht dupliziert sondern von den logischen Prozessoren gemeinsam genutzt. Man spricht bei dieser Technik auch von *Hyperthreading*. Wird bei der Ausführung eines Programms z.B. ein Cache-Fehlzugriff verursacht, kann nahezu verzögerungsfrei die Ausführung eines Threads fortgesetzt werden, der in einem anderen logischen Prozessor zur Ausführung bereitsteht. Unter geringer Vergrößerung der Transistorzahl zum Abspeichern des zweiten Prozessorzustandes können so einzelne Prozessorkerne deutlich besser ausgelastet werden [RR07].

Stehen mehrere Prozessorkerne auf dem Rechner zur Verfügung, können Programmausführungen dadurch beschleunigt werden, dass auf jedem Prozessorkern Threads des Programms parallel ausgeführt werden. Eine Kommunikation und Koordination zwischen den Threads kann durch gemeinsame Variablen erfolgen. Darauf soll im folgenden Abschnitt 3.2.3 genauer eingegangen werden.

Wird für ein Programm mit Multithreading ein bestimmtes Verhältnis zwischen Wartezeiten auf Eingabe-/Ausgabe-Operationen und tatsächlicher CPU-Auslastung angenommen, kann mit Gleichung 3.1 grob die CPU-Auslastung des Prozessors abgeschätzt werden. Die Variable t gibt dabei an, wie viele Threads ausgeführt werden und p ist die Wahrscheinlichkeit, dass ein Thread gerade auf die Antwort auf einen E-/A-Zugriff warten muss.

$$\text{Prozessorauslastung}(p,t) \quad = \quad 1 - p^t \tag{3.1}$$

Abbildung 3.1 zeigt den Zusammenhang zwischen Prozessorauslastung und Grad des Multithreading für unterschiedlich stark speichergebundene Programme und Anzahlen von

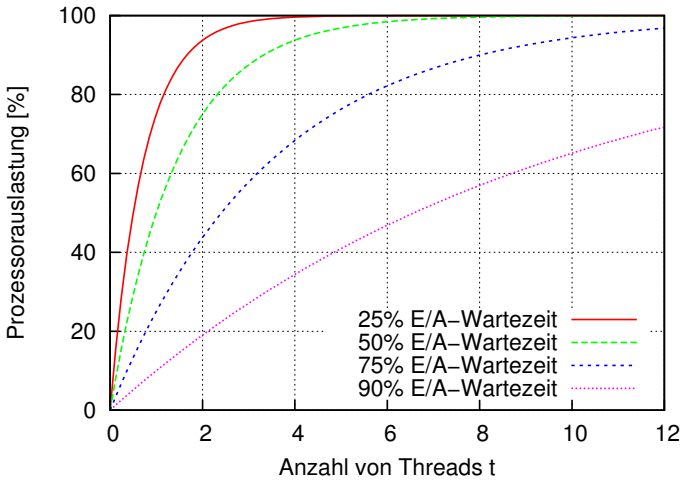


Abbildung 3.1: Abschätzung der erreichbaren Prozessorauslastung für unterschiedlich stark speichergebundene Anwendungen mit unterschiedlichen Graden von Multithreading

parallel ausgeführten Threads, wobei der Zusammenhang nur eine recht grobe Näherung der tatsächlich zu erreichenden Prozessorauslastung durch Multithreading ist [Tan09].

Besonders auf Graphikkarten ist das hardwareseitige Multithreading von zentraler Bedeutung in Hinblick auf das Verstecken von Latenzen. Darauf soll später in Abschnitt 3.3.3 genauer eingegangen werden.

3.2.3 Koordination unterschiedlicher Threads über gemeinsame Variablen

Da sich alle Prozessorkerne eines Multicore-Systems denselben Speicher teilen, muss beim Ändern von Datenstrukturen sichergestellt sein, dass alle Prozessoren stets eine Sicht auf konsistente Daten im Speicher erhalten. Dazu kann es nötig sein, dass allen Threads außer einem einzigen der Zugriff auf bestimmte Daten verwehrt wird, wenn dieser Thread Änderungen an den Daten vornehmen möchte.

Wird z.B. beim Baumaufbau ein Partikel in ein Blatt k eingefügt, das bereits die maximal zulässige Zahl von Partikeln enthält, müssen acht neue Blätter als Kinderknoten von k eingefügt werden. k selbst wird dadurch zu einem inneren Knoten. Anschließend müssen die Partikel auf die neuen Blätter verteilt und aus k entfernt werden. Haben andere Threads als derjenige, der das Partikel einfügt, jederzeit Zugriff auf die Daten, können die Daten in einem inkonsistenten Zustand ausgelesen oder sofort überschrieben werden. Man spricht dabei von *Race Conditions* oder *zeitkritischen Abläufen* [RR07].

Listing 3.1 Aufruf einer mit OpenMP parallelisierten Schleife

```
#pragma omp parallel for num_threads (6)
for (int i = 0; i < j; i++) { // Parallele Ausfuehrung des Schleifenrumpfs
    doStuffInParallel ();       // durch sechs Threads
}
```

Damit die Sicht der anderen Threads auf die gemeinsamen Daten stets konsistent bleibt, werden gemeinsame Variablen eingesetzt, über die die einzelnen Threads kommunizieren können. Es können so *Sperrvariablen* oder *Locks* für den Zugriff auf Datenstrukturen eingesetzt werden. Wenn ein Thread schreibend auf die Datenstruktur zugreifen möchte, muss er versuchen die Sperrvariable zu setzen. Gelingt dies, darf er die Daten ändern und kein anderer Thread erhält währenddessen Zugriff. Sind die Änderungen abgeschlossen, wird die Sperrvariable wieder zurückgesetzt.

Änderungen an Sperrvariablen müssen für alle anderen Threads sofort sichtbar sein, sodass viele Codeoptimierungen für den Zugriff auf solche Variablen nicht eingesetzt werden können. Der Zugriff auf Sperrvariablen ist dementsprechend teuer. Wollen mehrere Threads gleichzeitig auf Daten zugreifen, die durch Sperrvariablen geschützt werden, kann stets nur einer der Threads Zugriff erhalten. Müssen andere Threads auf das Freiwerden der Sperrvariable warten, spricht man von *Lock-contention*. Lock-contention kann die Effizienz paralleler Anwendungen maßgeblich beeinträchtigen.

Zur Programmierung von Parallelrechnern mit gemeinsamem Speicher stehen verschiedene Schnittstellen zur Verfügung, mit denen die Verwaltung von Threads, Synchronisation und Kommunikation realisiert werden können. In den folgenden Abschnitten soll auf die Schnittstellen OpenMP und Pthreads eingegangen werden, die zur Implementierung verschiedener paralleler Codes für die vorliegende Arbeit eingesetzt wurden.

3.2.4 Programmieren von Parallelrechnern mit gemeinsamem Speicher mit OpenMP

OpenMP ist eine Schnittstelle zur Programmierung von Anwendungen für Parallelrechner mit gemeinsamem Speicher. Es werden dem Entwickler dazu Konstrukte an die Hand gegeben, mit denen die Aufteilung von Arbeit auf mehrere Threads und die Synchronisation und Koordination dieser Threads umgesetzt werden können.

Die Programmausführung wird durch einen *Master-Thread* gesteuert, der alle sequentiellen Bestandteile des Codes ausführt. Alle Bereiche, die von mehreren Threads gleichzeitig bearbeitet werden sollen, werden durch `parallel`-Direktiven entsprechend gekennzeichnet. Listing 3.1 soll dies anhand einer parallelisierten Schleife demonstrieren. Die parallelen Blöcke werden von einem Team von Threads, deren Anzahl über den Parameter `num_threads`

gesteuert wird, nach dem fork-join-Modell ausgeführt. Am Ende eines jeden parallelen Blocks erfolgt dann automatisch eine Synchronisation. Der Master-Thread setzt also erst dann seine Ausführung fort, wenn alle anderen Threads des Teams ihre Berechnungen im parallelen Block beendet haben.

Alle Threads eines parallelen Blocks sind eindeutig durch ihre jeweiligen IDs unterscheidbar. Sowohl die Thread-ID als auch die Anzahl von Threads im Team lassen sich innerhalb paralleler Blöcke durch Funktionsaufrufe abfragen und in Abhängigkeit von ID und Threadanzahl lassen sich unterschiedliche Pfade im Programm einschlagen. Die Programmierung mit OpenMP erfolgt also nach dem SPMD-Programmiermodell (single-program multiple-data) [RR07].

3.2.5 Parallele Programmierung mit Pthreads

In Pthreads wird jeder Thread als ein Objekt repräsentiert, das als Parameter in verschiedenen Bibliotheksfunktionen verwendet wird. Jeder Thread führt genau eine Funktion aus, die bei der Thread-Erzeugung als Funktionszeiger übergeben wird. Alle Variablen, die in Bezug auf die Thread-Funktionen global definiert sind, werden von allen Threads gemeinsam verwendet und können zur Kommunikation zwischen unterschiedlichen Threads verwendet werden.

Wird bei der Programmierung mit OpenMP am Ende eines parallelen Abschnitts jeweils automatisch eine Synchronisation zwischen den Threads durchgeführt, muss bei Pthreads explizit auf die Beendigung von Threads gewartet werden, wenn die berechneten Daten für den weiteren Programmverlauf benötigt werden. Zum Warten auf die Beendigung von Threads gibt es, genauso wie für die Threaderzeugung, eine eigene Funktion, die ebenso das entsprechende Thread-Objekt als Parameter verwendet.

3.3 Einsatz von Graphikkarten zur Beschleunigung von Berechnungen

Neben der Verbesserung von Hauptprozessoren wurden auch Graphikkarten stetig weiterentwickelt. Mit aktuellen Generationen von Graphikkarten, die im Gegensatz zu ihren Vorgängern statt einer festen Graphikpipeline eine frei programmierbare Recheneinheit darstellen, lassen sich allgemeine Anwendungen beschleunigen. Graphikkarten verfügen über eigenen Arbeitsspeicher, der nicht direkt von der CPU aus gelesen und geschrieben werden kann und einen eigenen Adressraum besitzt. Hauptprozessor und Graphikprozessor eines Rechners zusammen können also als ein MIMD-System mit physikalisch verteiltem Speicher betrachtet werden.

Zur Beschleunigung verschiedener Teilaufgaben des Barnes-Hut-Verfahrens wurden für die vorliegende Arbeit nVidia-Graphikkarten eingesetzt. Diese zeichnen sich dadurch aus, dass sie sehr viele Prozessorkerne besitzen, die allerdings, verglichen mit aktuellen Hauptprozessoren, einen recht eingeschränkten Funktionsumfang besitzen. Mit CUDA steht eine Schnittstelle zur Verfügung, die es ermöglicht, allgemeine Berechnungen auf nVidia-Graphikkarten auszuführen.

Sowohl auf den Aufbau als auch die Programmierung von Graphikkarten wird in den folgenden Abschnitten genauer eingegangen. Im Folgenden wird die Graphikkarte auch synonym als *Device* und der Hauptprozessor als *Host* bezeichnet.

3.3.1 Überblick über den Aufbau von nVidia-Graphikkarten

Bezogen auf die theoretisch mögliche Anzahl von Fließkommaoperationen pro Zeiteinheit wurden in der Weiterentwicklung von Graphikkarten in den letzten Jahren deutlich größere Fortschritte gemacht als bei Hauptprozessoren. Jede Graphikkarte besitzt eine Menge von *Streaming Multiprozessoren (SM)*, die jeweils aus mehreren *Streaming Prozessoren (SP)* aufgebaut sind. Die SMs besitzen jeweils einen Programm- und Datencache und ein Steuerwerk, sodass alle SPs eines Multiprozessors zu jedem Zeitpunkt dieselbe Operation ausführen müssen [KH10]. Wie im SIMD-Modell kann dies allerdings auf unterschiedlichen Daten erfolgen. Da alle SMs eigene Steuerwerke besitzen, können allerdings auch unterschiedliche Operationen parallel ausgeführt werden. Daher lassen sich Graphikkarten dem MSIMD-Modell aus Abschnitt 3.1.1 zuordnen.

In Abbildung 3.2 ist stark vereinfacht der schematische Aufbau einer CUDA-fähigen Graphikkarte mit zwei Streaming Multiprozessoren und je acht Streaming Prozessoren pro SM dargestellt.

Um genau nachvollziehen zu können, wie die parallele Ausführung von Programmen auf der Graphikkarte vonstatten geht, wollen wir uns nun zunächst anschauen, wie Funktionen für die Ausführung auf der Graphikkarte in CUDA geschrieben werden können, bevor wir darauf eingehen, wie die Programmausführung in Hardware realisiert ist.

3.3.2 Parallele Programmierung von Graphikkarten mit CUDA

CUDA ist eine Spracherweiterung für C/C++, die es ermöglicht, Programmbestandteile für die parallele Ausführung auf Graphikkarten zu schreiben. Es werden dazu unterschiedliche Schlüsselwörter eingeführt, mit denen Funktionen als solche markiert werden können, die auf der Graphikkarte auszuführen sind. Funktionen, die auf der Graphikkarte parallel ausgeführt werden, werden als *Kernel* bezeichnet. Wird ein CUDA-Kernel aufgerufen, wird er auf der Graphikkarte in einem sogenannten *Grid* ausgeführt. Dieses Grid besteht aus *Blöcken*, die ihrerseits jeweils aus mehreren *Threads* bestehen. Es handelt sich bei jedem Grid

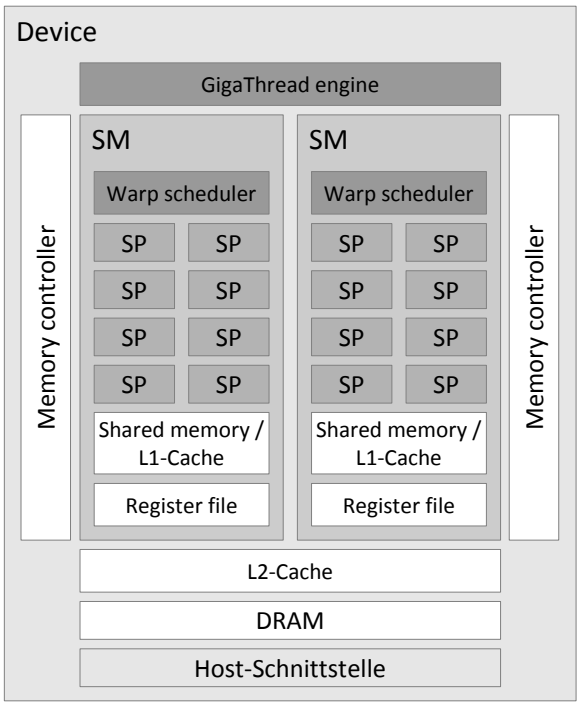


Abbildung 3.2: Schematischer Aufbau von CUDA-fähigen Graphikkarten. Frei nach [NVI09, KH10].

also um eine hierarchische Gruppierung der einzelnen Threads eines Kernels, die auf der Graphikkarte parallel ausgeführt werden. Wie viele Threads tatsächlich gestartet werden, wird beim Kernelaufruf im Host-Code angegeben.

Innerhalb eines Kernels kann über die Konstanten `threadIdx` und `blockIdx` jeder Thread und Block eindeutig identifiziert werden. Mit Hilfe der Konstanten `gridDim` und `blockDim` lassen sich zudem die Anzahl von Blöcken im Grid und die Anzahl von Threads pro Block innerhalb von Kerneln abfragen. In Listing 3.2 ist beispielhaft die Definition und der Aufruf eines CUDA-Kernels aus einem Host-Programm zu sehen.

Die Programmierung folgt dem SPMD-Modell (single-program multiple-data), nach dem auch die Programmierung in OpenMP aus Abschnitt 3.2.4 funktioniert. Das heißt, dass alle Threads dasselbe Programm ausführen, abhängig von Thread- und Block-Indizes allerdings unterschiedliche Programmverläufe möglich sind [KH10, NVI12].

Nach dieser kurzen Einführung in das CUDA-Programmiermodell wollen wir uns in den nächsten Abschnitten ansehen, wie die Programmausführung auf der Graphikkarte hardwareseitig bewerkstelligt wird.

Listing 3.2 Definition und Aufruf von CUDA-Kerneln

```
__global__ void kernel () {
       int idx = blockDim.x * blockIdx.x + threadIdx.x;
       // ... Berechnung fuer aktuellen Thread ...
       __syncthreads ();         // Barriere fuer alle Threads eines Blocks
}
void main () {
       kernel<<< blocks, threads_per_block >>> ();
       cudaThreadSynchronize (); // Warten auf Ende der Kernelausfuehrung
}
```

3.3.3 Parallele Programmausführung und Thread-Scheduling

Die Definition des Grids bei einem Kernelaufruf spiegelt sich direkt in der Ausführung auf der Graphikkarte wider: Jeder Block wird durch die GigaThread Engine einem Streaming Multiprozessor zur Berechnung zugewiesen. Die Threads eines jeden Blocks werden dann durch die einzelnen Streaming Prozessoren des Multiprozessors ausgeführt. Es besteht also eine direkte Verbindung zwischen dem hierarchischen Aufbau von Graphikkarten als Zusammenschluss mehrerer Multiprozessoren, die ihrerseits jeweils aus mehreren Streaming Prozessoren bestehen, und dem Grid des Kernels aus Blöcken von Threads.

Das Scheduling der einzelnen Threads eines Blocks erfolgt immer auf Basis von *Warps*. Warps bezeichnen eine Menge von Threads eines Blocks, die auf der Graphikkarte in einem SM von den einzelnen SPs parallel ausgeführt werden können. Es können je nach verfügbarem Speicherplatz mehrere Warps eines Blocks und mehrere Blöcke gleichzeitig auf einem Streaming Multiprozessor zur Ausführung bereitstehen, sodass das Hardware-Scheduling sehr effizient Latenzen verstecken kann, die z.B. bei langsamen Speicherzugriffen auftreten. CUDA-Threads sind sehr leichtgewichtige Threads. Erzeugung und Threadwechsel benötigen aufgrund direkter Hardwareunterstützung durch den Warp Scheduler nur einzelne Taktzyklen. Das Scheduling wird daher auch als *Zero-latency Thread Scheduling* bezeichnet [KH10].

Allerdings ist ein sehr sparsamer Umgang mit den Speicherressourcen der einzelnen Multiprozessoren Voraussetzung dafür, dass mehrere Warps und Blöcke gleichzeitig im Speicher von Multiprozessoren Platz finden. Um feststellen zu können, an welchen Stellen sich Einsparpotentiale ergeben, werden wir deshalb im nächsten Abschnitt die Speicherorganisation von Graphikkarten genauer betrachten.

3.3.4 Speicherorganisation aktueller nVidia-Graphikkarten

Auf der Graphikkarte sind unterschiedliche Speicherarten vorhanden, die sich hinsichtlich ihrer Zugriffsbereiche auf Host und Device und ihrer Zugriffsgeschwindigkeiten unterscheiden. Es wird bei den Speichern zum einen unterschieden zwischen *globalem* und *konstantem Speicher*, auf die beide von allen Threads aller Blöcke zugegriffen werden kann und zum anderen zwischen *Shared memory* und Registern, deren Zugriff auf die Threads einzelner Blöcke bzw. auf einzelne Threads beschränkt ist. Von der Graphikkarte aus ist auf den konstanten Speicher nur lesender Zugriff möglich.

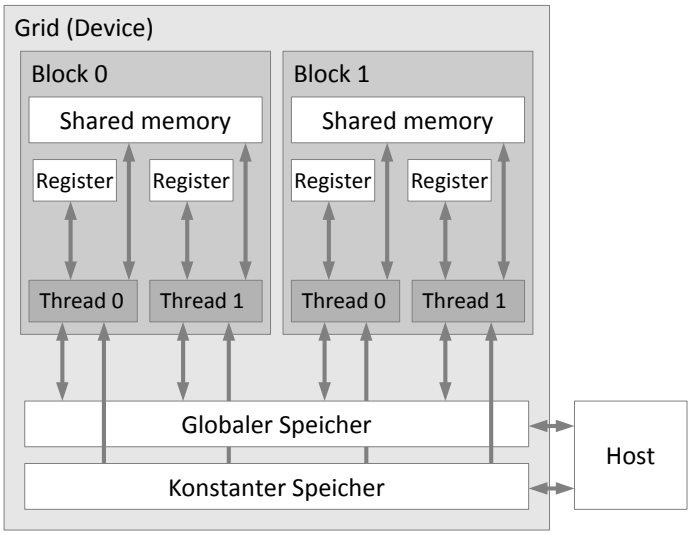


Abbildung 3.3: Überblick über die verschiedenen Speicherarten von CUDA-Graphikkarten und deren Zugriff nach [KH10].

Abbildung 3.3 soll die einzelnen Speicherarten, die auf der Graphikkarte zur Verfügung stehen, und deren Zugriffsbereiche veranschaulichen. Wie in der Abbildung zu sehen ist, lassen sich Daten zwischen Hauptspeicher des Hosts und sowohl globalem als auch konstantem Speicher der Graphikkarte übertragen.

Der Zugriff auf den globalen Speicher benötigt mehrere hundert Taktzyklen. Allerdings existieren Caches, mit deren Hilfe Speicherlatenzen versteckt werden können, solange der Zugriff auf den Speicher in angemessenen Zugriffsmustern erfolgt, wie sie in Abschnitt 3.2.1 beschrieben wurden.

Bezogen auf die Speicherzugriffszeiten bilden Register und Shared memory die schnellsten Speicherarten, auf die schreibend zugegriffen werden kann. Beide können eingesetzt werden, um Daten aus dem globalen Speicher abzulegen, die häufig verwendet werden. Es ist dabei allerdings im Auge zu behalten, dass beides stark begrenzte Ressourcen sind. Werden z.B. zu

viele Register pro Thread verwendet, kann dies die Anzahl gleichzeitig ausführbarer Threads auf dem Multiprozessor verringern. Die Effizienz von Kerneln hängt dementsprechend stark vom jeweiligen Einsatz der unterschiedlichen Speicherarten ab.

3.3.5 Kommunikation und Synchronisation zwischen Threads und Blöcken

Bei der Synchronisation innerhalb von CUDA-Kerneln muss zwischen der Synchronisation auf Block-Ebene und der Synchronisation zwischen Threads unterschiedlicher Blöcke, d.h. gridweit, unterschieden werden. Während alle Threads eines Blocks mit einem einfachen Funktionsaufruf synchronisiert werden können, der als Barriere fungiert, die erst dann überschritten werden kann, wenn alle Threads des Blocks sie erreicht haben, existiert für Threads unterschiedlicher Blöcke kein solcher Synchronisationsmechanismus [KH10]. Zwischen unterschiedlichen Blöcken kann die Synchronisation stattdessen mittels atomarer Operationen auf Variablen im globalen Graphikkartenspeicher realisiert werden, ist dann aber deutlich langsamer als die Synchronisation einzelner Blöcke.

Nach den vorangegangenen Ausführungen über parallele Rechnerarchitekturen und deren Programmierung soll in den nächsten Abschnitten darauf eingegangen werden, wie sich parallele Algorithmen bewerten und miteinander vergleichen lassen.

3.4 Kostenmaße zur Aufwandsabschätzung paralleler Programme

Zur Bewertung der Komplexität paralleler Programme werden zunächst parallele Kostenmaße eingeführt, anhand derer unterschiedliche parallele Implementierungen miteinander und mit sequentiellen Algorithmen verglichen werden können.

Im einfachsten Fall betrachtet man als Kostenmaß die Zeit, die ein Programm für seine Ausführung benötigt. Die Zeit, die ein paralleler Algorithmus zur Ausführung auf p Prozessoren für ein Problem der Größe N benötigt, wollen wir als $T_p(N)$ bezeichnen. $T_p(N)$ beschreibt dann die Dauer vom Start des Programms bis zu dessen Beendigung auf dem am längsten arbeitenden Prozessor.

Soll anstatt des reinen Zeitaufwands der tatsächliche Rechenaufwand eines parallelen Programms erfasst werden, lässt sich dieser nach Gleichung 3.2 angeben.

$$C_p(N) \quad = \quad p \cdot T_p(N) \tag{3.2}$$

$C_p(N)$ entspricht in etwa der Anzahl ausgeführter Instruktionen auf allen eingesetzten Prozessoren zusammen.

Mit Hilfe der Kostenmaße $T_p(N)$ und $C_p(N)$ lassen sich weiterhin verschiedene Maße zum Vergleich unterschiedlicher paralleler Programme angeben, die im nächsten Abschnitt vorgestellt werden.

3.5 Speedup und parallele Effizienz

Im Folgenden soll $T^*(N)$ die Zeit beschreiben, die ein optimaler sequentieller Algorithmus zur Lösung eines Problems der Größe N benötigt. Bildet man den Quotienten aus den Laufzeiten des optimalen sequentiellen Algorithmus' und des parallelen Algorithmus', erhält man nach Gleichung 3.3 den sogenannten *Speedup* einer Parallelisierung.

$$S_p(N) = \frac{T^*(N)}{T_p(N)} \tag{3.3}$$

Je näher der Wert $S_p(N)$ an p heranreicht, desto besser ist die Parallelisierung.

Um dagegen festzustellen, wie effizient eine Parallelisierung in Bezug auf die Anzahl der ausgeführten Instruktionen ist, lässt sich weiterhin die *parallele Effizienz* des Algorithmus' nach Gleichung 3.4 angeben.

$$E_p(N) = \frac{T^*(N)}{C_p(N)} = \frac{S_p(N)}{p} \tag{3.4}$$

Die parallele Effizienz kann maximal den Wert 1 erreichen. Die Effizienz lässt sich interpretieren als Verhältnis zwischen dem Anteil der tatsächlichen Berechnung des Problems und des Verwaltungsaufwands einer Parallelisierung, z.B. durch Kommunikation zwischen den Prozessoren. Der Speedup dagegen geht vom reinen Zeitaufwand aus, der bei p Prozessoren bestenfalls um den Faktor p verkleinert werden kann [RR07].

3.6 Beschreibung der eingesetzten Testsysteme

Um die verschiedenen im Laufe der Arbeit entstandenen sequentiellen und parallelen Implementierungen verschiedener Teilprobleme des Barnes-Hut-Algorithmus' bezüglich ihres Laufzeitverhaltens und ihrer Effizienz zu untersuchen, wurden zwei unterschiedliche Testsysteme eingesetzt, die im Folgenden beschrieben werden sollen.

3.6.1 Testsystem 1: TheCell

Das Testsystem TheCell verfügt über einen Hexacore-Prozessor: AMD Phenom II X6 1055T. Der Prozessortakt beträgt 2,8 GHz und der Rechner verfügt über 4 GiB Arbeitsspeicher. Die Cache-Hierarchie umfasst drei Stufen:

- L1-Cache: je Kern 64 + 64 KiB (Daten + Instruktionen)
- L2-Cache: je Kern 512 KiB mit Prozessortakt
- L3-Cache: 6 MiB mit 2 GHz, von allen Prozessorkernen gemeinsam verwendet

Die Graphikkarte ist eine nVidia GeForce GTX470 mit 1280 MiB. Sie verfügt insgesamt über 448 Streaming Prozessoren in 14 Streaming Multiprozessoren zu je 32 Streaming Prozessoren.

3.6.2 Testsystem 2: StarCluster

Das zweite Testsystem, StarCluster, besitzt einen Intel i7-2670QM Hauptprozessor mit 2,2 GHz Taktfrequenz. Der Prozessor stellt dem Betriebssystem acht logische Prozessoren durch Hyperthreading zur Verfügung und besitzt vier physikalische Prozessorkerne. Es stehen 8 GiB an Arbeitsspeicher zur Verfügung. Wie auf dem Testsystem TheCell wird eine dreistufige Cache-Hierarchie verwendet:

- L1-Cache: je Kern 32 + 32 KiB (Daten + Instruktionen)
- L2-Cache: je Kern 256 KiB mit Prozessortakt
- L3-Cache: 6 MiB mit Prozessortakt, gemeinsam von allen Prozessorkernen genutzt

Bei der verwendeten Graphikkarte handelt es sich um eine nVidia GeForce GTX570M mit 1536 MiB Arbeitsspeicher. Die Graphikkarte verfügt insgesamt über 336 SPs.

3.6.3 Eigenschaften der Softwareumgebung

Auf beiden eingesetzten Testrechnern war das Linux-Betriebssystem kubuntu 11.10 installiert. Als Übersetzer kam GCC in der Version 4.4.6 zum Einsatz. Zur Graphikkartenprogrammierung wurde auf beiden Systemen das CUDA-Toolkit in der Version 4.1 verwendet.

Messungen zur Übertragungsgeschwindigkeit von Daten zwischen Host und Device sollen später in Abschnitt 5.2.1 auf Seite 55 für beide Testsysteme vorgestellt werden.

4 Verwandte Arbeiten

In diesem Kapitel wird ein kurzer Überblick über verwandte Arbeiten anderer Autoren gegeben und es werden Bezüge zwischen diesen und der vorliegenden Arbeit hergestellt.

In [NHP07] wird eine hochoptimierte reine GPU-Implementierung des $\mathcal{O}(N^2)$-Algorithmus' zur Simulation von N-Body-Problemen in CUDA beschrieben. Zur Verringerung der Komplexität von $\mathcal{O}(N^2)$ wurde in [BH86] der dieser Diplomarbeit zugrundeliegende Barnes-Hut-Algorithmus entworfen, durch den die Komplexität auf $\mathcal{O}(N \log N)$ reduziert werden kann. Wie anhand des Simulationsbeispiels in Abschnitt 2.4.3 gezeigt wurde, hängt die Anzahl von zu berechnenden Interaktionen beim Barnes-Hut-Verfahren zu jedem Zeitpunkt von der jeweiligen Partikelverteilung ab. Es handelt sich folglich um ein Verfahren zur Lösung von Problemen mit hochgradig irregulärer Struktur, was eine effiziente Parallelisierung zu einer besonderen Herausforderung macht. In [TB07] wird mit der physikalischen Stoffsimulation ein ebenfalls hochgradig irreguläres Problem mit Hilfe des im letzten Kapitel bereits angesprochenen DOTS-Frameworks [BKWb98] parallelisiert.

4.1 Parallelisierung des Barnes-Hut-Algorithmus'

Um den Barnes-Hut-Algorithmus effizient auf Parallelrechnern umsetzen zu können, wurden eine Reihe von Parallelisierungsstrategien entwickelt. Für die Ausführung auf Vektorrechnern wird in [Bar90] ein Ansatz beschrieben, der die Baumtraversierung nicht pro Partikel ausführt sondern für Gruppen von Partikeln, die in gleichen Baumknoten liegen. Für diese Gruppen werden dann bei der Baumtraversierung zunächst keine Kräfte berechnet sondern Interaktionslisten von Pseudopartikeln und Partikeln bestimmt. Erst im nächsten Schritt werden anschließend zwischen den Partikeln der Gruppe und den Partikeln und Pseudopartikeln der Interaktionslisten Kräfte berechnet. So lässt sich die hochgradig irreguläre Baumtraversierung getrennt ausführen und die Kraftauswertung effizient in regulären Mustern parallelisieren. Eine Parallelisierung für Rechencluster wurde in [Spr05] entworfen.

4.1.1 Parallelisierungen mit GPU-Beschleunigung

Für die Ausführung des Barnes-Hut-Verfahrens auf der GPU ist es zunächst erforderlich, Baumstrukturen in einer linearen Darstellung angeben zu können, da sich zeigerbasierte

Baumstrukturen nicht zwischen CPU und GPU übertragen lassen. Die Verwendung von Octrees auf Graphikkarten mit Hilfe einer solchen linearen Darstellung wurde in [LHN05] beschrieben. Im Mittelpunkt der ersten Ansätze einer Ausführung des Barnes-Hut-Algorithmus' auf der Graphikkarte stand stets die Parallelisierung der Kraftauswertung, während die anderen Teilprobleme des Algorithmus zunächst weiterhin auf der CPU ausgeführt wurden. In [GBZ10] wird die in [Bar90] entworfene Idee aufgegriffen und eine GPU-basierte Parallelisierung der Kraftauswertung umgesetzt, die mit Hilfe stackbasierter Baumtraversierungen Interaktionslisten berechnet. Eine Auswertung der Kräfte auf der Graphikkarte durch rein iterative Baumtraversierungen mit Next- und More-Arrays, wie sie auch für die vorliegende Arbeit zum Einsatz kommen, wurde in [Nak12] umgesetzt. In neueren Arbeiten, wie in [BP11, BGZ12], konnten inzwischen Implementierungen realisiert werden, bei denen alle Teilprobleme des Barnes-Hut-Verfahrens auf der Graphikkarte ausgeführt werden können.

An geeigneter Stelle soll in späteren Kapiteln auf Details einzelner Quellen noch genauer eingegangen werden.

5 Effiziente parallele Implementierung des Barnes-Hut-Algorithmus'

Die im Zuge dieser Diplomarbeit entworfene Implementierung des Barnes-Hut-Algorithmus' ist streng modular aufgebaut. Der Algorithmus ist dazu in mehrere Teilprobleme zerlegt, die sich nacheinander, getrennt voneinander lösen lassen. Für alle Module zur Lösung der einzelnen Teilprobleme sind eindeutige Schnittstellen definiert, die es erlauben, verschiedene Implementierungen einzelner Module frei untereinander auszutauschen. Für jedes Teilproblem wurden unterschiedliche Lösungsansätze implementiert, die entweder auf der CPU oder der Graphikkarte ausgeführt werden können. Werden Modul-Implementierungen kombiniert, die auf unterschiedlichen Plattformen ausgeführt werden, müssen dabei Daten zwischen beiden Plattformen ausgetauscht werden, was mit Speichertransferlatenzen verbunden ist.

Im Folgenden sollen in Abschnitt 5.1 zunächst die einzelnen Teilprobleme beschrieben werden, in die der Gesamtalgorithmus zerlegt wird. Abschnitt 5.2 widmet sich dann der Frage, wie ein Speichermodell gestaltet werden kann, mit dem Implementierungen für Teilproblemlösungen, die auf unterschiedlichen Plattformen und verteiltem Speicher ausgeführt werden, dennoch einheitlich behandelt werden können. Zudem befasst sich der Abschnitt mit den Fragen, inwieweit Speicherlatenzen die Gesamtlaufzeit unterschiedlicher Programmkombinationen beeinflussen und ob und auf welche Weise Latenzen versteckt werden können. Anschließend wenden wir uns in Abschnitt 5.3 unterschiedlichen Datenstrukturen zur Darstellung von Bäumen zu und setzen uns mit der Frage auseinander, welche dieser Darstellungen sich für die gegebene Anwendung am besten eignet. In den Abschnitten 5.4 bis 5.9 werden schließlich die unterschiedlichen Implementierungen für die verschiedenen Teilprobleme vorgestellt und jeweils untereinander verglichen. Die Ergebnisse, die durch unterschiedliche Modul-Kombinationen erzielt wurden, werden im nächsten Kapitel in Abschnitt 6.1 vorgestellt, nachdem die einzelnen Teilproblemlösungen diskutiert wurden.

5.1 Beschreibung der einzelnen Teilprobleme

Die Teilprobleme, deren Lösungen als Module entworfen werden, entsprechen grob den Teilproblemen, die in Abschnitt 2.4.2 auf Seite 26 für allgemeine baumbasierte N-Body-

Algorithmen eingeführt wurden. Um genau untersuchen zu können, welche Teilprobleme sich auf welcher Architektur besonders effizient bearbeiten lassen, ist die Unterteilung allerdings noch etwas verfeinert worden. Insgesamt werden in jedem Zeitschritt sechs Teilprobleme bearbeitet, die im Folgenden vorgestellt werden sollen.

Hierarchische Gebietszerlegung und Baumaufbau Zunächst erfolgt der Aufbau einer hierarchischen Gebietszerlegung mittels Octrees. Die Details der unterschiedlichen Implementierungen des Baumaufbaus werden in Abschnitt 5.4 ausgeführt.

Linearisierung der rekursiven Baumstrukturen Anschließend wird der Octree, sofern er als Zeigerstruktur realisiert ist, in eine lineare Array-Repräsentation konvertiert. Dabei können unterschiedliche Array-Darstellungen zum Einsatz kommen, die in Abschnitt 5.3 beschrieben werden. Während der Linearisierung wird auch ein Indirektions-Array geschrieben, über das auf Partikel indirekt in der Reihenfolge der Baumtraversierung zugegriffen werden kann oder über das im nächsten Modul die Partikel neu sortiert werden können. Die Implementierung der Linearisierung wird in Abschnitt 5.5 behandelt.

Umsortierung der Partikel Optional kann anschließend an die Linearisierung des Octrees eine Umsortierung der Partikel nach der Depth-First-Traversierungsreihenfolge des Baums erfolgen. Erfolgt keine Umsortierung der Partikel, wird stattdessen das oben beschriebene Indirektions-Array verwendet, um auf Partikeldaten indirekt sortiert zuzugreifen. Die Sortierung kann die später folgende Kraftauswertung deutlich beschleunigen und wird in Abschnitt 5.6 beschrieben.

Berechnung der Pseudopartikel Nach erfolgtem Baumaufbau können die Pseudopartikel für die einzelnen Baumknoten berechnet werden. Details zu den unterschiedlichen Implementierungen folgen in Abschnitt 5.7.

Kraftberechnung Sind die Pseudopartikel bestimmt, stehen alle Daten zur Verfügung, die erforderlich sind, um die Kräfte, die auf die einzelnen Partikel wirken, zu bestimmen. Abschnitt 5.8 wird auf die einzelnen Implementierungen zur Kraftauswertung genauer eingehen.

Aktualisierung von Partikelpositionen und -geschwindigkeiten Im letzten Schritt jeder Iteration der Simulationsschleife werden aus den Kräften, alten Geschwindigkeiten und Partikelpositionen neue Positionen und Geschwindigkeiten für alle Partikel bestimmt. Die Eigenschaften der Implementierungen der Partikelaktualisierung werden in Abschnitt 5.9 genauer beleuchtet.

Um die Kombination beliebiger Modul-Implementierungen zu ermöglichen, müssen die Schnittstellen aller Module genau definiert sein. Die einzelnen Schnittstellen sollen im folgenden Abschnitt beschrieben werden.

5.1.1 Schnittstellen zwischen unterschiedlichen Abschnitten des Algorithmus'

Da Modul-Implementierungen sowohl auf der CPU als auch der GPU ausgeführt werden können sollen, müssen die Schnittstellen für Ein- und Ausgabedaten der einzelnen Module so geartet sein, dass die Daten direkt zwischen Hauptspeicher und Graphikkartenspeicher ausgetauscht werden können. Für Zeigerstrukturen, die auf der einen oder anderen Plattform erzeugt wurden, ist die direkte Datenübertragung auf die jeweils andere Plattform nicht möglich, da beide Plattformen getrennte Speicher und eigene Adressräume verwenden. Damit Daten an Schnittstellen zwischen CPU und GPU effizient ausgetauscht werden können, werden daher zur Definition von Schnittstellen ausschließlich Arrays verwendet.

Tabelle 5.1: Definition der unterschiedlichen Modul-Schnittstellen der Implementierung des Barnes-Hut-Verfahrens

Modul	**Eingabedaten**	**Ausgabedaten**
Baumaufbau	Partikelpositionen (Geschwindigkeiten und Massen nicht erforderlich)	
Linearisierung		Array-Darstellung der Baumstruktur, Indirektions-Array für Partikel
Partikelsortierung	Partikeldaten, Indirektions-Array	Partikeldaten
Pseudopartikelberechnung	Partikeldaten, Baum-Arrays	Pseudopartikeldaten
Kraftberechnung	Pseudopartikeldaten, Partikeldaten, Baum-Arrays	Kräfte
Partikelaktualisierung	Partikeldaten, Kräfte	Partikeldaten

In Tabelle 5.1 sind die Schnittstellen der einzelnen Module jeweils nach erforderlichen Eingabe- und zu berechnenden Ausgabedaten angegeben. Wenn der Baumaufbau auf der CPU in Form von Zeigerstrukturen erfolgt, folgt eine Linearisierung der Zeigerstrukturen als Zwischenschritt, bevor eine Array-Repräsentation des Baums vorliegt. Daher besteht keine Schnittstelle zwischen Baumaufbau und Linearisierung, die eine Kombination von CPU- und GPU-Implementierung erlauben würde. Bei Implementierungen des Baumaufbaus, die direkt auf Arrays operieren, muss keine anschließende Linearisierung erfolgen. Baumaufbau und Linearisierung sind in diesem Sinne als ein einziges Modul zu betrachten. Da Simulationen in einer Schleife über eine beliebige Anzahl von Zeitschritten ausgeführt werden, muss am Ende eines jeden Zeitschritts sichergestellt werden, dass die aktualisierten Partikeldaten für den Baumaufbau im folgenden Zeitschritt wieder bereitstehen.

5.1.2 Zur Komplexität der einzelnen Teilprobleme

Wie wir bereits wissen, ist für den Barnes-Hut-Algorithmus die Zeitkomplexität von $\mathcal{O}(N \log N)$ nachweisbar. Allerdings beantwortet dies nicht die Frage, wie es sich mit der Komplexität der oben definierten einzelnen Teilprobleme verhält. Dieser Frage wollen wir im Folgenden genauer nachgehen.

Zunächst lässt sich feststellen, dass der Baumaufbau eine Zeitkomplexität von $\mathcal{O}(N \log N)$ besitzt, sofern die Partikel gleichmäßig im Raum verteilt sind. Die Platzkomplexität dagegen ist linear in der Anzahl der Partikel. Für die Linearisierung ergibt sich aufgrund der linearen Platzkomplexität der Baumdarstellung eine lineare Zeit- und Platzkomplexität, da die Baumstruktur nur mit konstanter Häufigkeit traversiert werden muss, um die Linearisierung zu erhalten. Für die Sortierung der Partikel in der Reihenfolge der Baumtraversierung gilt nach obigem Argument ebenfalls eine Zeitkomplexität von $\mathcal{O}(N)$, ebenso wie für die Berechnung der Pseudopartikel. Da die Pseudopartikel jeweils für die einzelnen Baumknoten bestimmt werden und diese linear von der Partikelzahl abhängen, ist auch der Platzaufwand für die Pseudopartikel linear. Die Kraftauswertung, das rechenaufwändigste Teilproblem des Barnes-Hut-Algorithmus', lässt sich in der Zeit $\mathcal{O}(N \log N)$ lösen. Die anschließende Aktualisierung der Partikelpositionen und -geschwindigkeiten ist trivialerweise wieder mit $\mathcal{O}(N)$ Zeitaufwand zu bewältigen.

Zusammenfassend lässt sich sagen, dass alle Teilprobleme außer dem Baumaufbau und der Kraftauswertung nur einen linearen Zeitaufwand besitzen. Baumaufbau und Kraftauswertung dagegen haben die Zeitkomplexität $\mathcal{O}(N \log N)$. Die Platzkomplexität ist insgesamt nur linear. Dies ist insbesondere in Bezug auf die Betrachtung von Speicherlatenzen in Abschnitt 5.2.2 interessant.

5.2 Speicherverwaltung und Abstraktion von CPU- und GPU-Speicher

Wie bereits angesprochen, werden alle Daten, die zwischen unterschiedlichen Modulen ausgetauscht werden können, in Arrays gehalten. Sollen allerdings Modul-Implementierungen kombiniert werden, die auf unterschiedlichen Plattformen ausgeführt werden, müssen die Daten auch auf den jeweiligen Plattformen vorliegen. Damit die Implementierungen von Modulen für GPU und CPU einfach ausgetauscht werden können, wurde ein einheitliches Speichermodell entworfen.

Listing 5.1 zeigt, wie sich durch Templates dieselben Datenstrukturen für die Verwaltung der Simulationsdaten auf CPU und GPU verwenden lassen. Speicherallokierung auf beiden

Listing 5.1 Plattformunabhängige Implementierung der Arraystrukturen für die Simulation durch Templates

```
typedef enum {
       PlatformCPU = CPU,
       PlatformGPU = GPU
} SimulationPlatform;

template <SimulationPlatform platform>
class SimulationData {
       public:
              int nbodies;       // Anzahl von Partikeln in Simulation

              // ... Positionen, Geschwindigkeiten, Kraefte ...
              float *pos;
              // ... lineare Baumdarstellungen
              int *TreeArray;
              int *Next;
              int *More;

              // Kopierrichtung wird anhand des Templateparameters platform
              // bestimmt. Mit Hilfe von CUDA-Streams kann asynchron uebertragen
              // werden und Synchronisation verschoben werden, bis die Daten
              // wirklich auf der Gegenseite benoetigt werden.
              template <class sometype>
              void memcpy (sometype *dest, sometype *source, size_t elements,
                  cudaStream_t stream);
}
```

Plattformen und Kopieren von Daten zwischen Host und Device lassen sich so auf einheitliche Weise realisieren. Die Richtung, in der ein Kopiervorgang erfolgen muss, kann dann einfach anhand des Template-Arguments ermittelt werden.

Zudem lässt sich das Konzept der CUDA-Streams verwenden, um das Kopieren von Daten in eigenen Threads ausführen zu lassen, die dann an entsprechender Stelle explizit mit dem Rest der Anwendung synchronisiert werden können.

5.2.1 Datendurchsatz bei Übertragung zwischen beiden Plattformen und Aufwand für Speicherallokierung

Auf dem Host-Rechner kann Speicher auf zwei unterschiedliche Weisen allokiert werden. Bei der gewöhnlichen Allokierung mit Hilfe der Operatoren `new` und `delete` wird Speicher reserviert, der bei Bedarf in Sekundärspeicher, d.h. auf die Festplatte, ausgelagert werden

kann. Wird stattdessen die CUDA-Bibliotheksfunktionen `cudaMallocHost` verwendet, wird sogenanntes Page-Locked- oder Pinned-Memory allokiert, für das die Auslagerung in Sekundärspeicher durch das Betriebssystem nicht erlaubt ist.

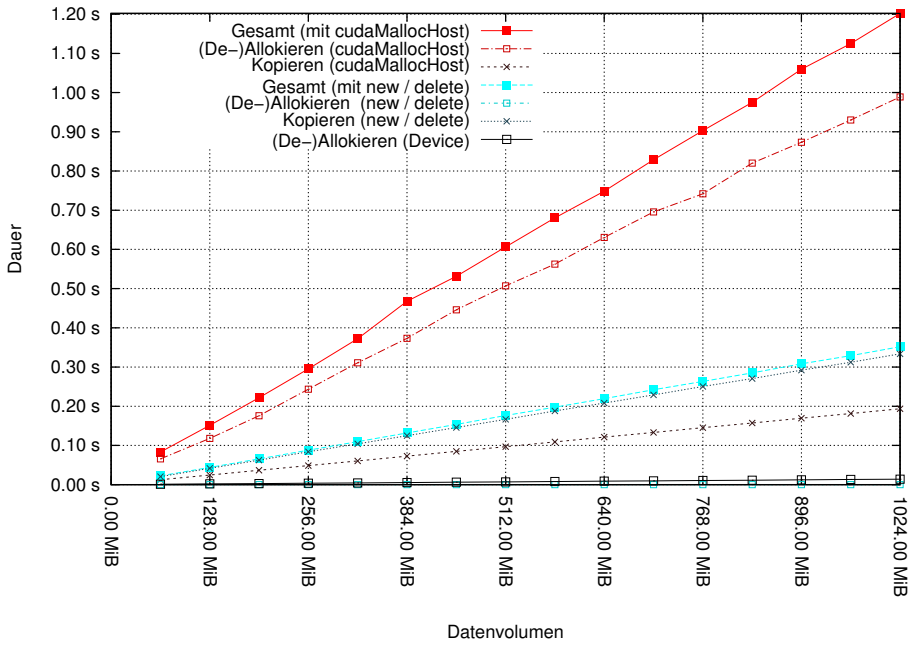
Wird Page-Locked-Memory verwendet, können Datenübertragungen zwischen der CPU und der GPU mit Hilfe des DMA-Controllers durchgeführt werden, wodurch deutlich gesteigerte Datendurchsätze erreicht werden können. Allerdings sind auch die Kosten zur Allokierung von Page-Locked-Memory im Vergleich zur gewöhnlichen Speicherallokierung deutlich höher. Um genaue Aussagen über den Aufwand von Allokierung und Kopiervorgängen zu erhalten, wurde auf TheCell und StarCluster der jeweilige Zeitaufwand von Speicherallokier-Kopier-Deallokier-Zyklen für unterschiedliche Datenvolumina experimentell ermittelt. Die Ergebnisse dazu sind in Abbildung 5.1 zu sehen.

Auf TheCell nahm das Allokieren von 1 GiB Page-Locked-Memory beinahe 1 s Zeit in Anspruch, auf StarCluster waren es immerhin ca. 800 ms. Der Zeitaufwand für die Speicherallokierung von normalem Speicher benötigte nur einen Bruchteil dessen für Page-Locked-Memory und lag auf beiden Testsystemen unterhalb von 20 ms. Bei der Messung des Zeitaufwands von Kopiervorgängen mit beiden Speicherarten zeigte sich allerdings auch, dass Page-Locked-Memory zumindest auf TheCell einen deutlich höheren Datendurchsatz erreichte. Die Übertragung von 1024 MiB konnte in 200 ms abgeschlossen werden, was eine Übertragungsgeschwindigkeit von $5\,\mathrm{GiB} \cdot \mathrm{s}^{-1}$ bedeutete. Wurde gewöhnlich allokierter Speicher verwendet, stieg die Dauer des Datentransfers auf TheCell um über 65 % auf ca. 330 ms zur Übertragung des gleichen Datenvolumens. Auf StarCluster waren die Unterschiede beim Datentransfer mit unterschiedlich allokiertem Speicher weniger stark ausgeprägt. Hier benötigte die Übertragung von 1 GiB mit Page-Locked-Memory 162 ms im Gegensatz zu 171 ms mit normal allokiertem Speicher. Die Übertragung mit Page-Locked-Memory war damit um weniger als 10 % schneller.

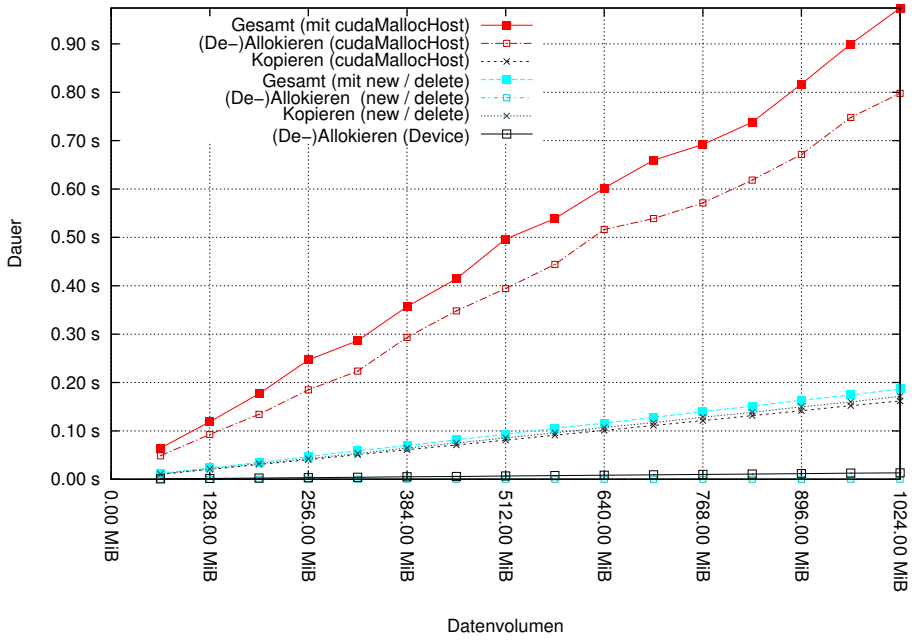
Für alle Daten, die zwischen CPU und GPU übertragen werden müssen, werden wegen des oben beschriebenen höheren Datendurchsatzes ausschließlich Arrays verwendet, die mit `cudaMallocHost` allokiert wurden. Außerdem lassen sich nur Daten, die mit `cudaMallocHost` allokiert wurden, asynchron mit Hilfe von CUDA-Streams transferieren. Aufgrund der Tatsache, dass das Allokieren von Page-Locked-Memory teuer ist, werden alle Arrays nur einmal zu Beginn allokiert und dann über den gesamten Programmverlauf wiederverwendet. Zur Zwischenablage von Daten, z.B. bei Sortiervorgängen, werden zusätzliche Arrays bereitgehalten.

5.2.2 Verstecken von Speicherlatenzen bei der Übertragung von Daten zwischen CPU und GPU

Werden Daten zwischen GPU und CPU ausgetauscht, müssen diese über den relativ langsamen PCIexpress-BUS versandt werden, wie im letzten Abschnitt dargelegt. Dies erzeugt



(a) Ergebnisse der Messungen auf TheCell



(b) Ergebnisse der Messungen auf StarCluster

Abbildung 5.1: Zeitaufwand für Speicherallokier-Kopier-Deallokier-Zyklen mit Page-Locked-Memory (cudaMallocHost) und normal allokiertem Speicher (mit Hilfe von new und delete) für unterschiedliche Datenvolumina auf den beiden eingesetzten Testsystemen.

Speicherlatenzen, die dazu führen, dass das Programm seine Ausführung unterbrechen muss, bis die Daten an der jeweiligen Zieladresse eingegangen sind.

An vielen Stellen lassen sich allerdings Daten bereits zu Zeitpunkten versenden, an denen sie noch nicht direkt auf der Gegenseite benötigt werden. In diesem Falle können die Daten asynchron, also ohne Unterbrechung der sonstigen Programmausführung, in eigenen Threads übertragen werden. Erst an expliziten Synchronisationspunkten, dort wo die Daten tatsächlich gebraucht werden, wird dann überprüft, ob die Daten bereits vollständig übertragen wurden und gegebenenfalls auf die Beendigung der Übertragung gewartet. Die Ausführung des Programms kann sich dann mit der Datenübertragung überlappen, um alle Ressourcen möglichst effizient auszunutzen.

Wie oben bereits angesprochen, wird die asynchrone Datenübertragung mittels CUDA-Streams realisiert. Für jedes einzelne Modul der Simulation wird dazu ein CUDA-Stream im Programm definiert. Wird eine Datenübertragung angestoßen, so wird als Parameter für die Kopierfunktion derjenige Stream angegeben, in dessen Modul die Daten spätestens auf der entsprechenden Zielplattform vorliegen müssen. Direkt vor der Ausführung des Zielmoduls des Kopiervorgangs findet dann die Synchronisation mit dem entsprechenden Stream statt.

In Abschnitt 5.1.1 wurden bereits die unterschiedlichen Modul-Schnittstellen beschrieben. Bei genauer Betrachtung fällt auf, dass z.B. für die Partikelsortierung nur das Indirektions-Array und die Partikeldaten vorliegen müssen. Gleichzeitig ist allerdings bereits die lineare Array-Darstellung der Baumstrukturen berechnet worden und kann dann parallel zur Sortierung der Partikel übertragen werden. Auf triviale Weise lassen sich Speicherlatenzen allerdings auch komplett verhindern, indem aufeinanderfolgende Module möglichst auf derselben Plattform ausgeführt werden.

5.3 Unterschiedliche Datentypen zur Beschreibung von Octrees

Für die rein CPU-basierte Implementierung des Barnes-Hut-Algorithmus' kamen drei unterschiedliche Datenstrukturen zum Einsatz, den Octree zu repräsentieren. So konnte festgestellt werden, welche Repräsentation sich für den Barnes-Hut-Algorithmus am effizientesten einsetzen lässt.

5.3.1 Darstellung von Octrees durch rekursive Zeigerstrukturen

Die natürlichste Repräsentation von Octrees im Rechner lässt sich mittels rekursiver Zeigerstrukturen erreichen. Jeder Baumknoten wird durch ein Objekt mit verschiedenen Attributen, wie Mittelpunkt, Kantenlänge und Pseudopartikeleigenschaften modelliert, das Zeiger auf acht weitere Objekte desselben Typs, die Kinderknoten im Baum, besitzt. In Listing 5.2 ist

Listing 5.2 Einfache Darstellung von Octrees mittels rekursiver Datentypen

```
struct OctreeNode {
       OctreeNode *children[8];
       .... Mittelpunkt und Kantenlaenge des Octree-Wuerfels
       .... weitere Attribute wie Pseudopartikeldaten
};
```

eine beispielhafte Definition für einen rekursiven Datentyp zur Repräsentation von Octrees angegeben.

Durch eine einfache Baumtraversierung kann der rekursiv definierte Baum in eine lineare Datenstruktur umgewandelt werden.

5.3.2 Einfache lineare Darstellung von Octrees in Arrays

Baumstrukturen können recht einfach in einem Baumdurchlauf linearisiert werden. Das genaue Vorgehen wird in Abschnitt 5.5 beschrieben. Hier soll nur kurz eine Erklärung folgen, wie mit Hilfe von Arrays die rekursiven Baumstrukturen dargestellt werden können.

Angenommen, der Octree stellt eine hierarchische Gebietszerlegung eines Simulationsgebiets mit N Partikeln dar, die bereits in der Reihenfolge einer Depth-First-Baumtraversierung sortiert vorliegen, dann wird ein Array mit M Elementen erzeugt, wobei $M = N +$ #Baumknoten gilt. Das Array soll im Folgenden *TreeArray* genannt werden. Alle Arrayelemente mit Index $i \geq N$, wobei wir stets nach C-Konvention die Arrayindizierung mit Index 0 beginnen, entsprechen einem Baumknoten. Jeder Knoten kann entweder ein innerer Knoten sein und besitzt dann Kinder oder er ist ein Blatt und kann dann entweder Partikel enthalten oder nicht. Die drei beschriebenen Fälle lassen sich einfach anhand der Werte im TreeArray erkennen.

Innerer Knoten Ein Knoten mit Index i im TreeArray ist daran als innerer Knoten zu erkennen, dass der Wert TreeArray$[i] = j \geq N$ ist. In diesem Falle ist j der Index des ersten Kindknotens von i. Alle acht Kinder sind fortlaufend, ausgehend von j, im TreeArray zu finden.

Blatt mit Partikeln Ein Blatt, das Partikel enthält, wird daran erkannt, dass es einen Wert TreeArray$[i] = j$ zwischen 0 und $(N-1)$ im TreeArray besitzt. Der Wert j ist der Index des ersten Partikels, das zum Blatt gehört, und an der Arraystelle j ist im TreeArray eingetragen, wie viele Partikel im Blatt zu finden sind. Alle Partikel des Blatts sind dann von j ausgehend aufsteigend indiziert.

Leeres Blatt Ein leeres Blatt im Baum lässt sich daran erkennen, dass sein Eintrag im TreeArray den Wert -1 hat.

Der Wurzelknoten besitzt stets den Index N.

Die Attribute, die im Falle der Darstellung durch Zeigerstrukturen direkt in den Knoten gespeichert werden können, werden bei Verwendung des TreeArrays in zusätzlichen Arrays abgelegt. Die Arrays, die Knotenattribute speichern, werden durch einen Offset so verschoben, dass auf sie mit denselben Indizes wie auf die Knoten im TreeArray zugegriffen werden kann.

Mit der eben beschriebenen Linearisierung von Octrees wird eine lineare Repräsentation erzeugt, die die rekursive Struktur der Octrees immer noch recht genau widerspiegelt. Für die Traversierung eines Octrees, der als TreeArray vorliegt, muss weiterhin entweder rekursiv vorgegangen werden oder stets auf einem Stack der Pfad von der Wurzel zum aktuellen Knoten zwischengespeichert werden. Neben der Darstellung durch das TreeArray gibt es jedoch weitere Linearisierungen, die eine stackfreie, vollkommen iterative Traversierung ermöglichen.

5.3.3 Lineare Darstellung von Baumstrukturen zur iterativen, stackfreien Traversierung

Eine rein iterative, stackfreie Traversierung von Baumstrukturen kann mit Hilfe einer Baumdarstellung durch zwei Arrays erfolgen. Diese werden als *Next-* und *More-Array* bezeichnet. Die beiden Arrays spiegeln dabei genau den Fortschritt wider, der bei einer rekursiven Traversierung gemacht werden kann. Entweder ein Knoten besitzt Kinder, dann kann zu diesen Kindern rekursiv abgestiegen werden, oder ein Knoten besitzt keine Kinder, dann wird bei der Traversierung das nächste Geschwister besucht oder falls der Knoten keine weiteren Geschwister besitzt, wird beim nächsten Geschwister des Vaters fortgefahren oder dem des Großvaters, bis kein einziger Vorfahr mehr Geschwister besitzt und die Traversierung endet.

Das More-Array speichert zu jedem Knoten dessen erstes nicht-leeres Kind, wohingegen im Next-Array zu jedem Knoten dessen Nachfolger zu finden ist, wie oben für den zweiten Fall beschrieben. Hat ein Knoten keine Kinder und ist also ein Blatt, steht im More-Array der Index des ersten Partikels, genau wie es für die Linearisierung im letzten Abschnitt beschrieben wurde. Abbildung 5.2 soll veranschaulichen, auf welche Knoten von den Next- und More-Arrays jeweils verwiesen wird. Knotenattribute werden, genau wie im letzten Abschnitt beschrieben, in zusätzlichen Arrays gespeichert.

Eine Traversierung des Baums mit Next- und More-Array gestaltet sich dann so, dass von der Wurzel aus so lange den Einträgen im More-Array gefolgt wird, bis ein Eintrag mit Wert $< n$ gefunden wird — also bis ein Blatt erreicht worden ist — oder das Akzeptanzkriterium greift. Anschließend wird der Nachfolger aus dem Next-Array gelesen und ein neuer Abstieg über das More-Array, ausgehend vom aktuellen Knoten, beginnt. Dies wird solange fortgesetzt,

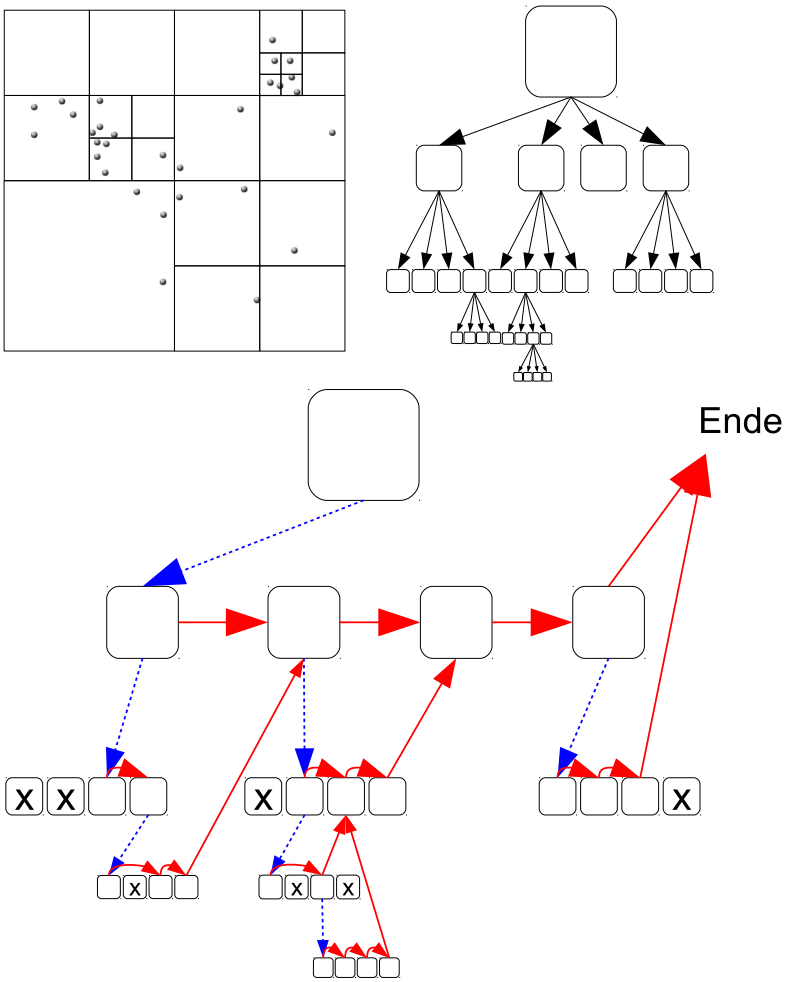


Abbildung 5.2: Darstellung der Struktur von Next- und More-Array anhand eines Beispielbaums. More-Verweise sind als blau gestrichelte und Next-Verweise als rote, durchgehende Pfeile eingezeichnet. Leere Knoten sind mit X gekennzeichnet und werden in Next- und More-Arrays direkt übersprungen.

bis im Next-Array schließlich der Wert -1 gefunden wird, mit dem signalisiert wird, dass keine weiteren Knoten zu bearbeiten sind und die Traversierung beendet ist.

5.3.4 Ergebnisse und Diskussion der drei beschriebenen Repräsentationen für Octrees

Da die einzelnen Objekte in der zeigerbasierten Baumdarstellung dynamisch allokiert und deallokiert werden, ist die Reihenfolge der Objekte im Speicher abhängig von der Rei-

henfolge, in der Partikel beim Baumaufbau in den Baum eingefügt werden. Dies kann zu schlechter räumlicher Kohärenz der Baumknoten im Speicher führen und die erreichbare Geschwindigkeit späterer Baumtraversierungen deutlich einschränken. Zudem lassen sich Zeigerstrukturen, die auf der CPU erzeugt wurden, nicht direkt auf den Graphikkartenspeicher übertragen, weshalb eine Linearisierung der Zeigerstrukturen unerlässlich ist, wenn Graphikkarten zur Beschleunigung von Berechnungen eingesetzt werden sollen.

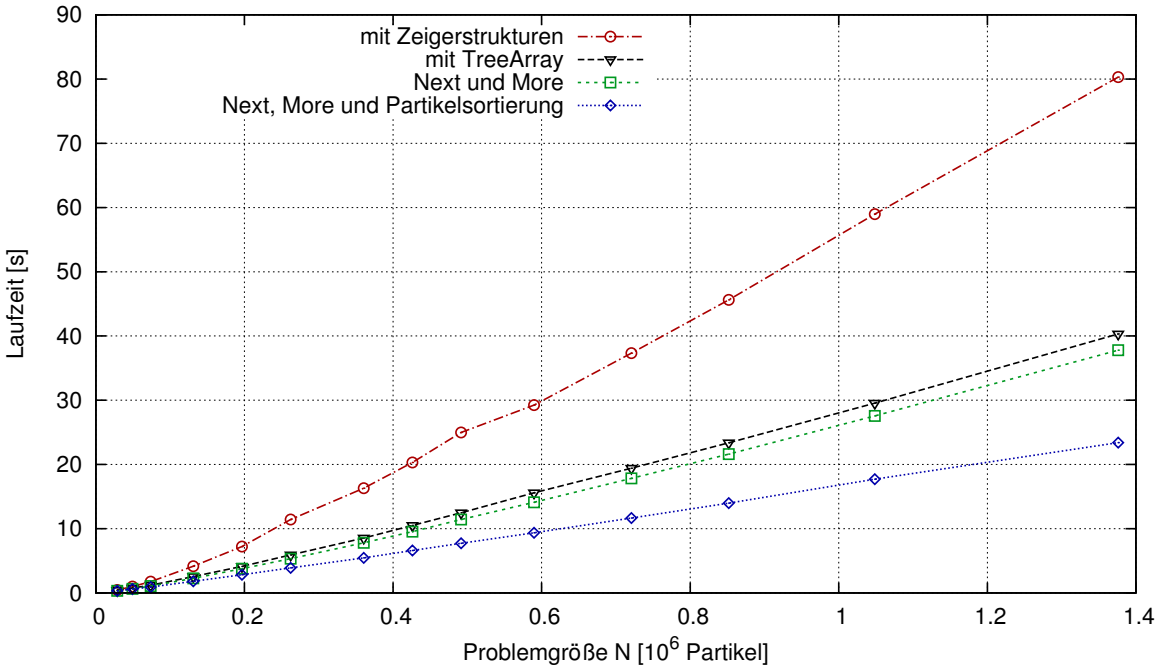


Abbildung 5.3: Laufzeitverhalten von Implementierungen des Barnes-Hut-Algorithmus' mit unterschiedlichen Baumdarstellungen. Dargestellt ist jeweils der Mittelwert aus sechs Zeitschritten mit $\theta = 0{,}75$.

Abbildung 5.3 zeigt, welche Auswirkungen die Verwendung der unterschiedlichen Octreedarstellungen im Vergleich zur Darstellung durch Zeigerstrukturen auf die Laufzeit rein sequentieller CPU-Implementierungen hatte. Es ist deutlich zu sehen, dass alle Array-basierten Baumdarstellungen deutlich schnellere Simulationsläufe ermöglichten als die zeigerbasierten Datenstrukturen, obwohl die Linearisierung einen zusätzlichen Aufwand bedeutete. Zudem lässt sich erkennen, dass sich durch die Verwendung von Next- und More-Arrays und iterativer Baumtraversierung anstatt rekursiver Traversierung die Laufzeit noch weiter reduzieren ließ als lediglich durch die Verwendung der linearen Baumdarstellung mittels des TreeArrays.

Abbildung 5.4 zeigt, welche Speedups allein durch die Verwendung linearisierter Baumstrukturen im Vergleich zur zeigerbasierten Darstellung von Bäumen erreichbar waren. Zudem ist

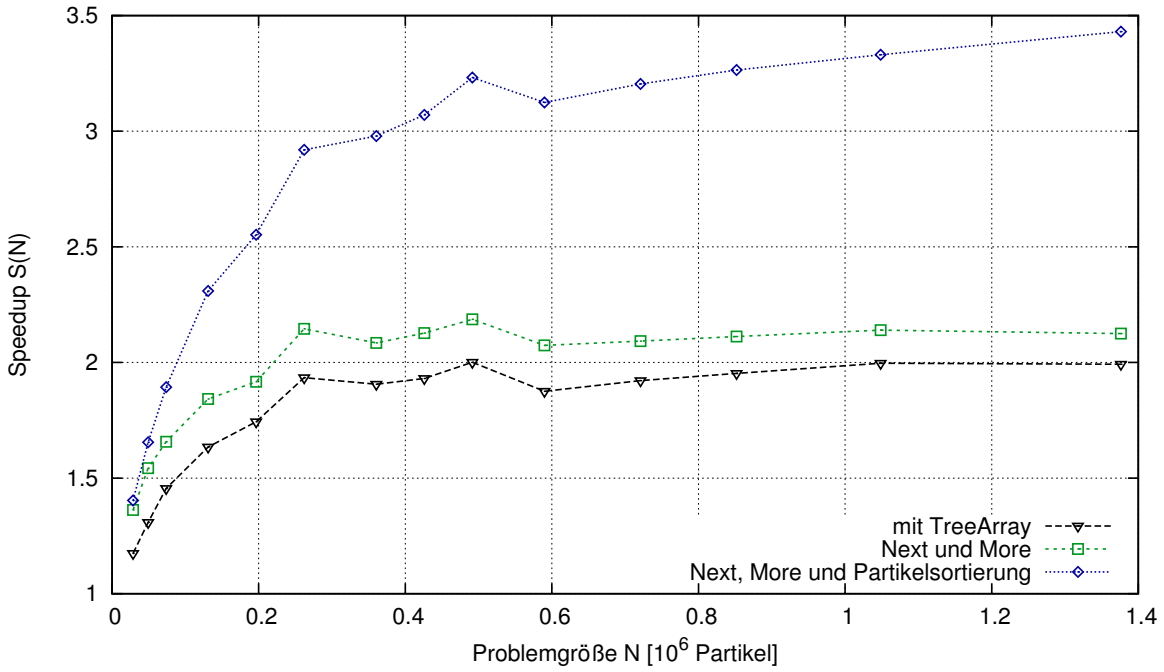


Abbildung 5.4: Speedup von Implementierungen des Barnes-Hut-Algorithmus' mit unterschiedlichen Baumlinearisierungen im Vergleich zu einer zeigerbasierten Implementierung. Gemessen jeweils am Mittelwert aus sechs Zeitschritten mit $\theta = 0{,}75$.

zu sehen, dass durch Partikelsortierung in der Reihenfolge der Baumtraversierung zusätzlich zum Speedup durch Verwendung von Next- und More-Arrays deutliche Beschleunigungen erreicht werden konnten. Der Speedup war für kleine Problemgrößen mit weniger als $3 \cdot 10^5$ Partikeln stark abhängig von der Problemgröße. Bei der Berechnung größerer Probleme näherte sich der erreichbare Speedup allerdings recht schnell einer Sättigungsgrenze an.

Wie in Abschnitt 5.3.2 beschrieben, spiegelt die linearisierte Baumrepräsentation mittels TreeArray die rekursive Struktur von Octrees genau wider. Da die Berechnung der Pseudopartikel in der Reihenfolge des Wiederaufstiegs aus einer Rekursion zu den Blättern des Baums erfolgen muss, weil die Pseudopartikelberechnung eines Baumknotens immer von den Werten der Kinderknoten abhängt, werden in allen Implementierungen der Berechnung der Pseudopartikel, die in Abschnitt 5.7 beschrieben werden, stets TreeArrays eingesetzt. Für alle Implementierungen der Kraftauswertung, die in Abschnitt 5.8 behandelt werden sollen, werden Baumtraversierungen ausschließlich über Next- und More-Arrays realisiert. Ebenso wird für alle Testläufe, außer den drei oben beschriebenen, im Weiteren eine Partikelsortierung vorgenommen.

5.4 Baumaufbau

Für den Baumaufbau wurden neben einer sequentiellen Variante unterschiedliche Parallelisierungsstrategien implementiert, die im Folgenden zunächst einzeln beschrieben und anschließend miteinander verglichen werden.

5.4.1 Sequentieller Baumaufbau

Der sequentielle Baumaufbau erfolgt durch sukzessives Einfügen von Partikeln in einen zu Beginn nur aus dem Wurzelknoten bestehenden Baum. Für jedes Partikel wird ein iterativer Abstieg im Baum bis zu einem Blatt durchgeführt. Kann das Partikel nicht in dieses Blatt eingefügt werden, weil bereits die maximale Anzahl von Partikeln pro Blatt erreicht ist, wird das Blatt zu einem inneren Knoten, indem es acht neue Blätter als Kinderknoten erhält. Die Partikel werden dann auf die neuen Kinderknoten in der nächsten Ebene verteilt.

5.4.2 Paralleler Baumaufbau mit einem Thread pro Baumknoten

Eine sehr einfache Parallelisierungsstrategie besteht darin, anstatt Partikel von der Wurzel aus einzufügen, Partikel erst ab einem bestimmten Knoten im Baum einzufügen. Im einfachsten Fall werden nur der Wurzelknoten und dessen acht Kinder erzeugt und anschließend parallel in acht Threads die Partikel direkt ab den ersten acht Kinderknoten der Wurzel einsortiert.

Vorteil dieses Parallelisierungsansatzes ist, dass sich die Threads niemals gegenseitig beeinflussen, da diese komplett unabhängig voneinander an unterschiedlichen Teilbäumen arbeiten. Da keine Sperrmechanismen eingesetzt werden müssen, geht das Einfügen einzelner Partikel schnell vonstatten. Allerdings muss dafür auch jedes Partikel von jedem Thread daraufhin untersucht werden, ob es in dem dem jeweiligen Thread zugewiesenen Oktanten einsortiert werden muss. Zudem ist die Lastverteilung nur begrenzt steuerbar, was dazu führen kann, dass bei ungleichmäßigen Partikelverteilungen ein einzelner Thread die Gesamtlaufzeit des Baumaufbaus dominiert, während die anderen Threads keine Arbeit mehr zu verrichten haben. Daher wurde dieser Ansatz nicht weiterverfolgt.

5.4.3 Baumaufbau mit Parallelisierung auf Partikelebene

Um zu verhindern, dass einzelne Threads sehr lange arbeiten, während andere nur wenig Arbeit zu erledigen haben, werden in einem flexibleren Parallelisierungsansatz nicht einzelne Teilbäume komplett getrennt bearbeitet, sondern Partikel gleichmäßig auf Threads verteilt und dann parallel in denselben gemeinsamen Baum eingefügt. Bei statischer Lastverteilung bearbeitet jeder einzelne der p Prozessoren dann $\frac{N}{p}$ Partikel. Statt statischer Lastverteilung kann auch eine dynamische Lastverteilung eingesetzt werden, bei der die Partikelpartition

aus Vorhersagen über die zu erwartende Rechenlast zum Einfügen von Partikeln in den Baum ermittelt wird. Für den Baumaufbau kann dieselbe dynamische Lastverteilung eingesetzt werden, die später in Abschnitt 5.8.3 für die Kraftauswertung beschrieben wird. Allerdings wird für den Baumaufbau dann die Partitionierung für die Kraftauswertung des vorangegangenen Zeitschritts verwendet.

Bei Parallelisierungen, die Partikel in einen gemeinsamen Baum einfügen, ist darauf zu achten, dass keine zwei Threads gleichzeitig denselben Knoten verändern können. Es wird dazu pro Blatt eine Sperrvariable verwendet, die dann gesetzt wird, wenn ein Thread Änderungen am Blatt vornehmen möchte. Ist die Sperrvariable bereits von einem anderen Thread gesetzt worden, wird der Zugriff verwehrt, und es muss zu einem späteren Zeitpunkt erneut versucht werden, Zugriff zu erhalten. Es sind nur Sperrvariablen für Blätter erforderlich, da durch das Einfügen von Partikeln keine Änderungen an inneren Knoten erfolgen können. Blätter können verändert werden, indem ihnen entweder Partikel hinzugefügt werden oder indem sie, wenn die maximale Anzahl von Partikeln überschritten würde, zu inneren Knoten werden und sie ihrerseits Blätter als Kinderknoten erhalten. Je nachdem, ob ein Partikel direkt eingefügt werden kann, oder ob neue Knoten erzeugt werden und alle Partikel in neue Blätter eingefügt werden müssen, können Blätter unterschiedlich lange gesperrt bleiben.

5.4.4 Paralleler Baumaufbau mit Listen von Partikeln, die nicht auf Anhieb eingefügt werden konnten

Schlägt das Einfügen eines Partikels in den Baum fehl, weil die Sperrvariable für den Knoten, in den das Partikel eingefügt werden sollte, bereits von einem anderen Thread gesetzt worden ist, wird im einfachsten Fall solange erneut versucht, die Sperrvariable zu setzen, bis dies gelingt. Dies kann allerdings dazu führen, dass Threads sehr lange keinen Fortschritt machen können, da sie damit beschäftigt sind auf die Sperrvariable zu warten. Man nennt diesen Zustand *busy-waiting*.

Statt auf das Freiwerden der Sperrvariablen nach dem Prinzip des busy-waiting zu warten, kann eine Liste eingesetzt werden, in die alle Partikel eingetragen werden, für die das Einfügen in ein Blatt nicht auf Anhieb erfolgreich war. Zusätzlich zum Partikel wird noch die Adresse des entsprechenden Baumknotens, dessen Sperrvariable nicht gesetzt werden konnte, mitgespeichert. Anschließend wird direkt mit dem nächsten Partikel fortgefahren und versucht dieses einzufügen. Auf diese Weise entstehen keine langen Wartezeiten auf Sperrvariablen, die nicht frei sind.

Nach einer vorgegebenen Zahl von bearbeiteten Partikeln wird anschließend erneut versucht, die Partikel, die in der Wiederholungsliste zwischengespeichert wurden, in den Baum einzufügen. Da diejenigen Baumknoten in der Liste mitgespeichert werden, bis zu denen für das Einfügen der Partikel im Baum bereits ein Abstieg durchgeführt wurde, muss der Abstieg nicht erneut von der Wurzel ausgehend begonnen werden.

5.4.5 Vorstellung von Ansätzen zum Baumaufbau auf der Graphikkarte

Alle bisher vorgestellten Implementierungen des Baumaufbaus werden entweder sequentiell oder parallel auf der CPU ausgeführt. Dabei ist auch ein Baumaufbau auf der Graphikkarte grundsätzlich möglich.

Die einfachste Möglichkeit besteht darin, wie beim parallelen Ansatz aus Abschnitt 5.4.3, Partikel parallel in den Baum einzufügen und Sperrvariablen für Blätter zu verwenden. In [BP11] wurde dieser Ansatz realisiert. Allerdings ist dort stets nur ein Partikel pro Blatt zugelassen, und es gibt kein Abbruchkriterium in Abhängigkeit von der Baumtiefe. Liegen zwei Partikel sehr nahe beieinander, müssen solange neue Blätter erzeugt werden, bis beide Partikel in unterschiedliche Blätter einsortiert werden können. Im schlimmsten Fall reicht dafür der Speicherplatz nicht aus, und die gesamte Simulation muss abgebrochen werden. Soll es dagegen möglich sein, eine variable Anzahl von Partikeln in Blättern zu erlauben, müssen komplexere Datenstrukturen auf der Graphikkarte eingesetzt werden, was sich deutlich auf die Ausführungsdauer auswirkt.

Ein gänzlich anderer Weg zur Parallelisierung des Baumaufbaus auf der Graphikkarte wurde in [BGZ12] verfolgt. In der Implementierung, die dort beschrieben wird, wird der Baum Ebene für Ebene, von der Wurzel ausgehend, aufgebaut. So kann die Verwendung von Sperrvariablen vollständig umgangen werden. Allerdings müssen dafür die Partikel bereits nach raumfüllenden Kurven sortiert vorliegen.

5.4.6 Ergebnisse und Diskussion der unterschiedlichen Implementierungen des Baumaufbaus

Für alle Implementierungen, die entworfen wurden, sind die maximal erlaubte Baumtiefe und die maximale Anzahl von Partikeln pro Blatt frei einstellbar. Für die im Folgenden beschriebenen Ergebnisse wurde, wenn nicht anders erwähnt, als maximale Tiefe des Baums stets 24 gewählt und die Höchstzahl erlaubter Partikel pro Blatt auf 32 festgesetzt. Die Begrenzung der maximalen Baumtiefe dient dazu, sicherzustellen, dass keine entarteten Bäume entstehen, wenn Partikel sehr eng benachbart im Simulationsgebiet liegen. Für alle Testläufe wurde die Partikelsortierung verwendet, die in Abschnitt 5.6 diskutiert werden, sodass die Partikeldaten für den Baumaufbau vorlagen immer nach der Sortierung aus dem letzten Zeitschritt.

Durch den Vergleich zwischen der Laufzeit der Parallelisierungen auf nur einem Prozessorkern und dem sequentiellen Baumaufbau konnte festgestellt werden, wieviel Zusatzaufwand durch Sperren und Entsperren der Blätter entstand. Auf beiden Testsystemen wurde der Baumaufbau durch die Verwendung von Sperrvariablen durchschnittlich um 5 % verlangsamt. Die Speedups für die Testläufe auf TheCell sind in Abbildung 5.5 zu sehen, die für StarCluster in Abbildung 5.6.

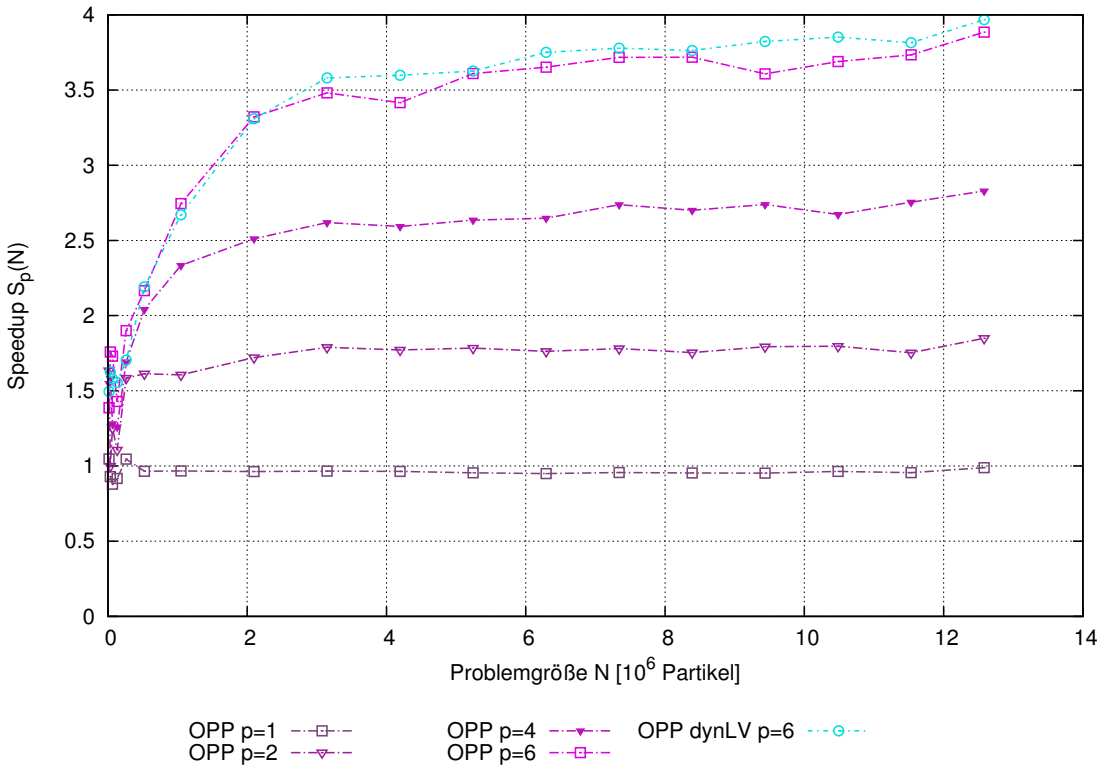


Abbildung 5.5: Speedup unterschiedlicher paralleler Baumaufbau-Implementierungen auf TheCell im Vergleich zur rein sequentiellen Implementierung. Gemessen jeweils am Mittelwert aus 16 Zeitschritten.
OPP: Baumaufbau mit OpenMP-Parallelisierung auf Partikelebene. dynLV: Implementierung mit dynamischer Lastverteilung

Unter Einsatz aller sechs Prozessorkerne konnten auf TheCell Speedups von über 3,5 erreicht werden. Auf StarCluster waren beim Baumaufbau unter Beteiligung aller acht logischen Prozessoren Speedups von über 4 festzustellen. Für Problemgrößen von über $4 \cdot 10^6$ Partikeln näherte sich der Speedup langsam einer Sättigung an, die allerdings selbst für das größte Problem noch nicht erreicht war. Bei kleineren Problemgrößen waren nur deutlich geringere Speedups erreichbar. Es zeigte sich auch, dass sich der Baumaufbau durch die dynamische Lastverteilung, auf die in Abschnitt 5.8.3 detailliert eingegangen wird, auf beiden Testsystemen deutlich beschleunigen ließ.

In den Abbildungen 5.7 und 5.8 sind die auf TheCell bzw. StarCluster erreichten parallelen Effizienzen der einzelnen Parallelisierungen dargestellt. Auf beiden Testsystemen war, bei Parallelisierung auf allen Prozessorkernen bzw. logischen Prozessoren, mit dynamischer Lastverteilung für größere Probleme stets eine Effizienz von über 50 % zu erreichen, auf The-

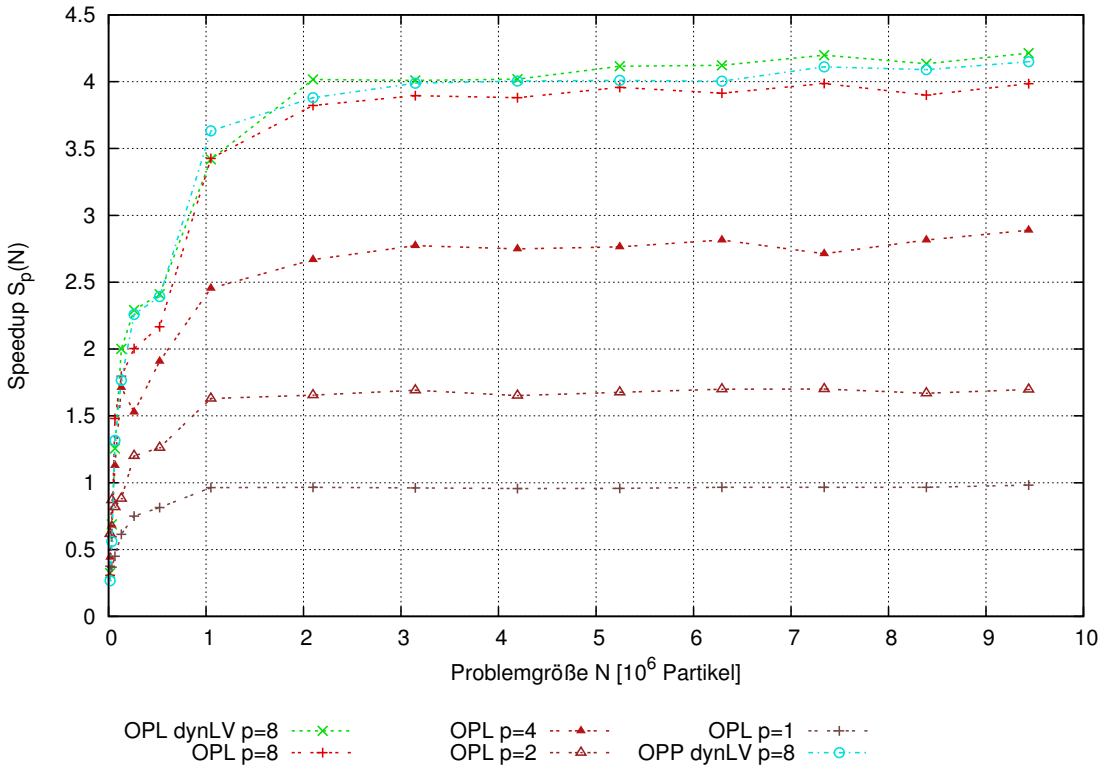


Abbildung 5.6: Speedup unterschiedlicher paralleler Baumaufbau-Implementierungen auf StarCluster im Vergleich zur rein sequentiellen Implementierung. Gemessen jeweils am Mittelwert aus 16 Zeitschritten.
OPP: Baumaufbau mit OpenMP-Parallelisierung auf Partikelebene. OPL: wie OPP, mit Wiederholungslisten. dynLV: Implementierung mit dynamischer Lastverteilung

Cell sogar eine parallele Effizienz von über 60 %. Der parallele Baumaufbau auf StarCluster war allerdings insgesamt als effizienter zu bewerten, wenn man bedenkt, dass tatsächlich nur vier physikalische Prozessorkerne zur Verfügung standen.

Unterschiede zwischen dem parallelen Baumaufbau auf Partikelbasis und der Parallelisierung mit Listen von Partikeln zur Wiederholung des Einfügevorgangs waren für die hier dargestellten Testläufe nur eingeschränkt nachweisbar, wahrscheinlich da durch die Vorsortierung der Partikel selten Konflikte beim Sperren von Blättern auftraten. Für eine größere Anzahl von 64 maximal erlaubten Partikeln pro Blatt konnte allerdings eine Beschleunigung durch Wiederholungslisten festgestellt werden, da dann die Sperren für einzelne Blätter länger bestehen bleiben. Ein ähnlicher Effekt wird sich vermutlich auch beobachten lassen, wenn der Baumaufbau auf Systemen mit mehr Prozessorkernen zum Einsatz kommt und

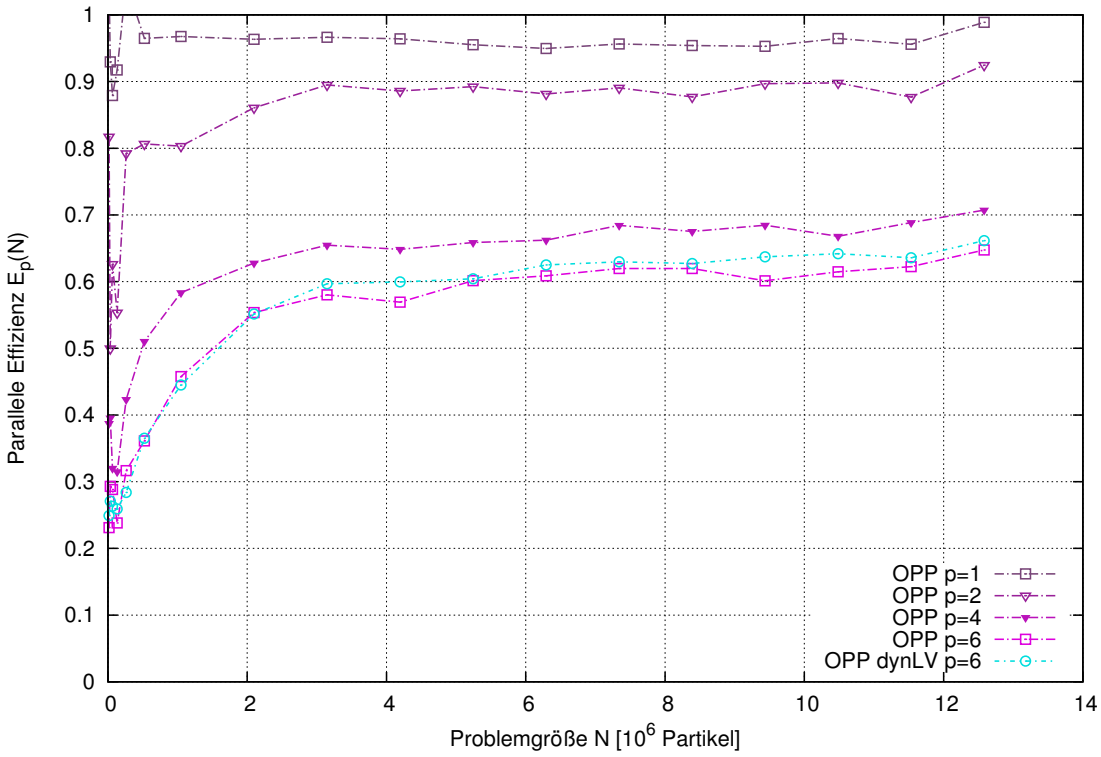


Abbildung 5.7: Parallele Effizienz unterschiedlicher Baumaufbau-Implementierungen auf TheCell im Vergleich zur rein sequentiellen Implementierung. Gemessen jeweils am Mittelwert aus 16 Zeitschritten.
OPP: Baumaufbau mit OpenMP-Parallelisierung auf Partikelebene. dynLV: Implementierung mit dynamischer Lastverteilung

die Wahrscheinlichkeit steigt, dass mehrere Threads gleichzeitig Partikel in dasselbe Blatt einzufügen versuchen.

Der in [BGZ12] berichtete Anteil des Baumaufbaus auf der Graphikkarte am Gesamtaufwand für Simulationen von etwa 10 % deckt sich in der Größenordnung mit dem Anteil, der für den schnellsten parallelen Baumaufbau auf der CPU in der optimalen Kombination von Modul-Implementierungen in der vorliegenden Arbeit ermittelt werden konnte. Für die optimale rein CPU-basierte Implementierung lag der Anteil des Baumaufbaus dagegen unter 2 %, woraus sich schließen lässt, dass sich das Teilproblem des Baumaufbaus im Vergleich zu den anderen Teilproblemen nur eingeschränkt für eine Parallelisierung auf Graphikkarten eignet.

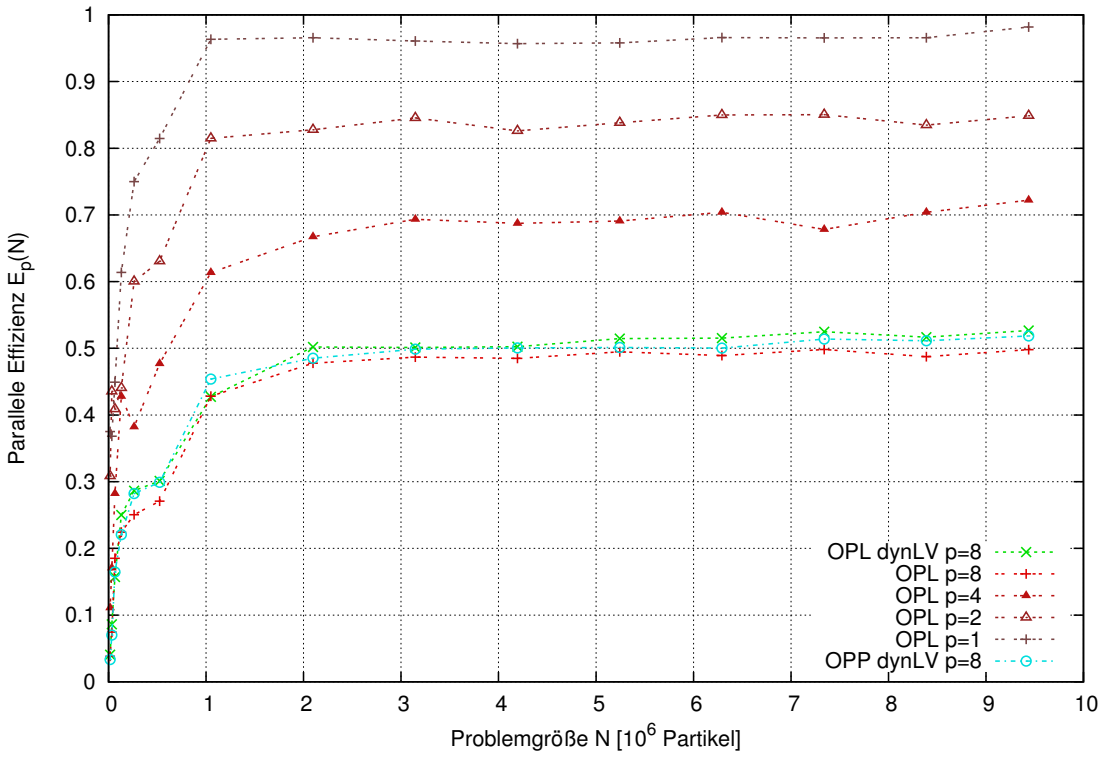


Abbildung 5.8: Parallele Effizienz unterschiedlicher Baumaufbau-Implementierungen auf StarCluster im Vergleich zur rein sequentiellen Implementierung. Gemessen jeweils am Mittelwert aus 16 Zeitschritten.
OPP: Baumaufbau mit OpenMP-Parallelisierung auf Partikelebene. OPL: wie OPP, mit Wiederholungslisten. dynLV: Implementierung mit dynamischer Lastverteilung

5.5 Linearisierung von Baumstrukturen für die Verwendung auf der Graphikkarte

Um Pseudopartikelberechnungen und Kraftauswertung auf der Graphikkarte durchführen zu können, müssen die Baumstrukturen der hierarchischen Gebietszerlegung linearisiert und in Arrays abgelegt werden. In Abschnitt 5.3 auf Seite 58 wurden bereits die beiden linearen Baumdarstellungen beschrieben, die erzeugt werden sollen.

Die Erzeugung des TreeArrays erfolgt in einem einzigen Baumdurchlauf, ausgehend von der Wurzel, die den Index N erhält. In Algorithmus 5.1 ist der Ansatz zur Linearisierung des TreeArrays dargestellt. Alle Kinder eines Knotens werden stets zusammenhängend im Array abgelegt. Die Position, die ein Knoten im TreeArray erhält, wird durch eine globale

Algorithmus 5.1 Linearisierung einer Baumstruktur mit N Partikeln in einem Baumdurchlauf

```
globaler TreeArray-Index x
function LINEARISIERUNG
    x ← N + 1                          // x ist stets das nächste freie Element im TreeArray
    Traversierung (Wurzelknoten, N)    // Aufruf der rekursiven Traversierungsfunktion
end function
function TRAVERSIERUNG(Baumknoten k, Array-Index i)
    Knotenattribute von k in Arrays an Index i schreiben
    if hatKinder (k) then
        Startindex für Kinder x_s ← x
        x ← x + 8                      // Acht neue Kinder werden in TreeArray geschrieben
        for all Kinder k_l von k do    // Index l läuft von 0 bis 7
            Traversierung (k_l, x_s + l)
        end for
    else                               // k ist Blatt und enthält entweder Partikel oder nicht
        if BlattEnthältPartikel (k) then
            Verweise auf Partikel des Blatts k schreiben
        else
            TreeArray-Eintrag an Stelle i auf -1 setzen
        end if
    end if
end function
```

Indexvariable bestimmt, die mit jedem besuchten Knoten erhöht wird. Es wird im Array also genau die Reihenfolge der Baumtraversierung widergespiegelt.

Sollen zusätzlich Next- und More-Arrays erzeugt werden, wird dies bereits unter Verwendung des TreeArrays realisiert. Die Erzeugung der beiden Arrays erfolgt ebenfalls jeweils durch eine Baumtraversierung, sodass die Knoten wieder in der Reihenfolge der Traversierung in die Arrays geschrieben werden. Die Indizes der Knoten im TreeArray entsprechen dann genau den Indizes in den Next- und More-Arrays, sodass die Arrays der Knotenattribute gemeinsam verwendet werden können. Um spätere Traversierungen durch Next- und More-Arrays zu beschleunigen, werden Verweise auf leere Blätter übersprungen und stattdessen direkt die nächsten nicht-leeren Knoten referenziert.

Im Zuge der Linearisierung wird auch das Indirektions-Array geschrieben, das die Reihenfolge, in der Partikel bei der Traversierung besucht werden, auf tatsächliche Partikelindizes abbildet. Über dieses Array kann auf Partikel indirekt sortiert zugegriffen werden, oder alternativ lassen sich über dieses Array die Partikel in $\mathcal{O}(N)$ Zeitaufwand nach raumfüllenden Kurven sortieren.

5.6 Umsortierung der Partikel nach raumfüllenden Kurven

Werden die Partikel in der Reihenfolge einer Depth-First-Traversierung des Baums neu sortiert, entsteht eine Sortierung nach einer raumfüllenden Kurve. Für alle vorliegenden Ergebnisse wurden als raumfüllende Kurven Hilbert-Kurven gewählt.

Es stehen sowohl eine sequentielle und eine parallele CPU-Implementierung der Partikelsortierung zur Verfügung als auch eine parallele GPU-Variante in CUDA. Alle Implementierungen werden auf dieselbe Weise realisiert. Es werden dazu, zusätzlich zu den Partikeldaten, weitere Arrays verwendet, in die über das im letzten Abschnitt beschriebene Indirektions-Array die Partikeldaten umsortiert geschrieben werden. Anschließend werden die Arrays vertauscht, sodass die neu sortierten Arrays anstelle der alten verwendet werden.

5.6.1 Ergebnisse und Diskussion der Partikelsortierung

Es wurde bereits in Abschnitt 5.3.4 gezeigt, dass bei Verwendung von Next- und More-Arrays durch Neusortierung der Partikel im Vergleich zum unsortierten bzw. indirekten Zugriff auf die Partikeldaten ein deutlicher Speedup erreicht werden konnte. Da die wirkliche Sortierarbeit bereits durch den Baumaufbau erledigt wird und anschließend die Partikel nur noch in linearer Zeit umgeordnet werden müssen, stellte sich heraus, dass die hohe Speicherbandbreite der Graphikkarte sehr hohe Speedups beim Umsortieren von Partikeldaten ermöglichte.

In Abbildung 5.9 sind der erreichte Speedup durch den Einsatz der Graphikkarte zum Sortieren der Partikel und der Speedup durch die parallele CPU-Implementierung gegenüber der rein sequentiellen CPU-Implementierung dargestellt. Es konnten bei Verwendung aller sechs Prozessorkerne auf dem Testsystem TheCell nur Speedups der Größenordnung $S_6(N) \leq 3{,}5$ erreicht werden, wobei sich recht schnell eine Sättigung des Speedups einstellte. Auf der Graphikkarte ließ sich die Ausführung dagegen in der Spitze beinahe um den Faktor 20 beschleunigen, wobei selbst für die größten Probleme noch keine Sättigung erreicht zu sein schien. Dieses Verhalten ließ sich damit erklären, dass das Umsortieren der Partikel in irregulären Zugriffsmustern erfolgte. Auf der Graphikkarte ließen sich dann — je mehr Threads erzeugt wurden, desto besser — Latenzen durch hardwareseitiges Multithreading verstecken. Für die CPU-basierte Implementierung war dies nicht möglich.

Die Vorteile einer Neusortierung der Partikel bestehen darin, dass für andere Module eine erhöhte räumliche Kohärenz der Speicherzugriffe auf die Partikel in Blättern erreicht werden kann, da die Partikel in derselben Reihenfolge angeordnet sind, wie sie bei einer Traversierung besucht werden. Zudem ist durch Sortierung nach raumfüllenden Kurven gewährleistet, dass Bereiche, die im Partikelarray eng benachbart sind, auch im Simulationsgebiet nahe beieinander liegen. Wird die folgende Kraftauswertung für Partikel in der Reihenfolge einer raumfüllenden Kurve durchgeführt, kann dies die räumliche und zeitliche Lokalität der

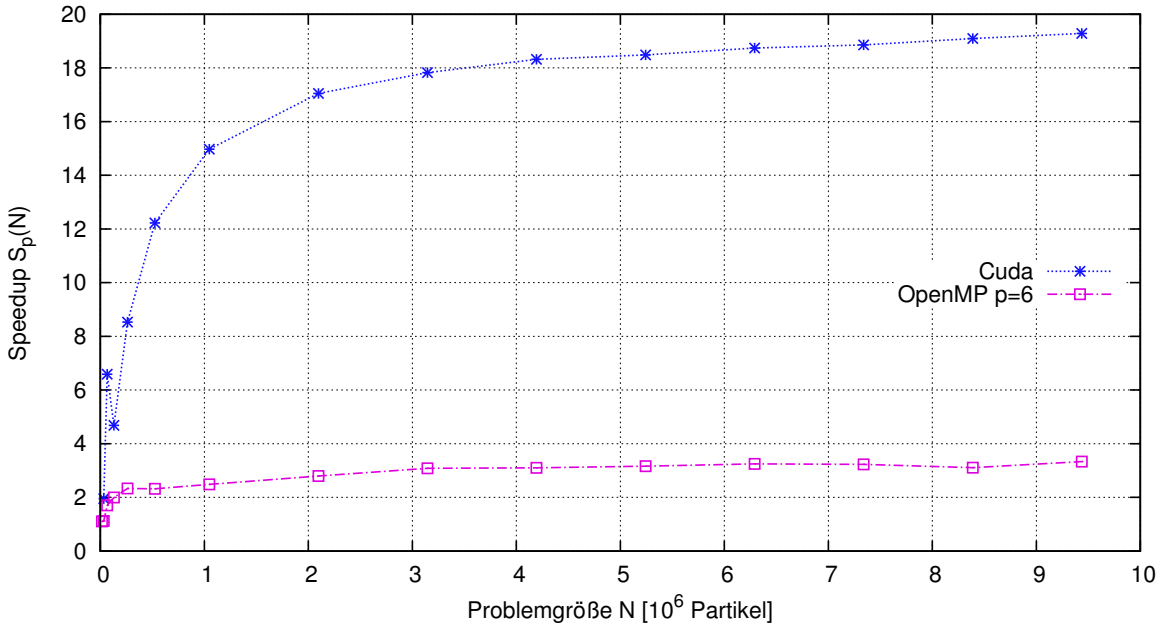


Abbildung 5.9: Speedup der Partikelsortierung mit OpenMP und CUDA im Vergleich zur sequentiellen Implementierung auf TheCell. Gemessen jeweils am Mittelwert aus 16 Zeitschritten.

Speicherzugriffe deutlich erhöhen, da für Baumtraversierungen eng benachbarter Partikel meist auch dieselben Baumknoten zur Kraftauswertung geöffnet werden müssen.

Ein hohes Maß an Kohärenz der Baumtraversierungen benachbarter Partikel ist besonders für die massiv parallele Kraftauswertung auf der Graphikkarte von Bedeutung, da Warps stets synchron dieselben Befehle bearbeiten müssen. Wird für alle Partikel eines Warps derselbe Pfad durch den Baum verfolgt, können alle Recheneinheiten optimal ausgelastet werden.

Wird Partikelsortierung verwendet, liegen die Partikel im folgenden Zeitschritt für den Baumaufbau immer in der Sortierung aus dem letzten Zeitschritt vor. Da sich Partikel in der Regel pro Zeitschritt kaum bewegen, liegen die Partikel also wieder nahezu in der Sortierung nach raumfüllenden Kurven vor. Dadurch kann der Baumaufbau stark beschleunigt werden, denn, wie bei der Kraftauswertung, ist dann ein hohes Maß an Kohärenz beim Abstieg im Baum zu den Blättern gewährleistet.

5.7 Berechnung der Pseudopartikel

Partikel von Baumknoten werden zu Pseudopartikeln zusammengefasst, da mit deren Hilfe die Berechnung von Wechselwirkungen mit weit entfernten Partikeln deutlich beschleunigt werden kann, indem die einzelnen Partikel-Partikel-Wechselwirkungen durch eine einzige Partikel-Pseudopartikel-Wechselwirkung ersetzt werden. Jedes Pseudopartikel wird durch seinen Schwerpunkt, seine Masse und seinen Durchmesser definiert. Statt den tatsächlichen Durchmesser zu berechnen, wird in den unten beschriebenen Implementierungen näherungsweise die Kantenlänge des Baumknotens verwendet.

5.7.1 Sequentielle Implementierung

Die einfachste Weise Pseudopartikel für jeden Baumknoten eines Octrees zu bestimmen, besteht darin, eine rekursive Traversierung des Baums bis zu den Blättern vorzunehmen. In den Blättern angekommen werden dann die Pseudopartikel derselben berechnet. Anschließend werden beim Wiederaufstieg die Pseudopartikel der inneren Knoten berechnet, bis die Rekursion schließlich wieder zur Wurzel zurückgekehrt ist.

Eine Parallelisierung der Pseudopartikelberechnung für die Ausführung auf dem Hauptprozessor wurde nicht vorgenommen, da die Berechnung der Pseudopartikel nur einen sehr kleinen Bruchteil der Gesamtausführungsdauer ausmacht und zudem nur von linearer Zeitkomplexität ist. Es wurde aber eine parallele Implementierung in CUDA umgesetzt.

5.7.2 Parallele Berechnung der Pseudopartikel in CUDA

Da die Berechnung der Pseudopartikel durch rekursive Funktionen auf älteren Graphikkarten nicht möglich ist und auf aktuellen Graphikkarten nur recht ineffizient durchgeführt werden kann, wird bei der Parallelisierung in CUDA auf Rekursion verzichtet. Stattdessen werden die Pseudopartikel der Baumknoten Ebene für Ebene von unten nach oben — von den Blättern zur Wurzel — in mehreren Kernelaufrufen bestimmt, bis schließlich das Pseudopartikel für den Wurzelknoten berechnet ist.

Zu Beginn werden alle Pseudopartikel als noch zu berechnen markiert, indem die Pseudopartikelmasse auf -1 gesetzt wird. Anschließend wird aus dem TreeArray in einem Kernelaufruf ein Eltern-Array berechnet, das für jeden Knoten auf dessen Elternknoten verweist. Für die Wurzel wird als Verweis -1 eingesetzt. Zuletzt wird noch ein Array erzeugt, in das für jeden Knoten k der Abstand zu den Blättern im Unterbaum von k eingetragen wird.

Mit Hilfe der beschriebenen Arrays können dann über mehrere Kernelaufrufe die Pseudopartikel der einzelnen Ebenen des Baums von den Blättern bis zur Wurzel nacheinander

bestimmt werden. Die Berechnung endet dann, wenn der Wurzelknoten einen Wert ungleich -1 annimmt. Die Anzahl von Kernelausführungen entspricht dann genau der Baumtiefe.

5.7.3 Ergebnisse und Diskussion der unterschiedlichen Verfahren zur Berechnung der Pseudopartikel

Durch die Vermeidung von Rekursion kann auf der Graphikkarte für Funktionsaufrufe zur Pseudopartikelberechnung Inline-Expansion eingesetzt werden. Dies kann die Bearbeitung deutlich beschleunigen, da keine Stackrahmen für rekursive Funktionsaufrufe in den globalen Speicher geschrieben werden müssen. Beim Einsatz von unbegrenzt rekursiven Funktionen wäre dies nicht möglich.

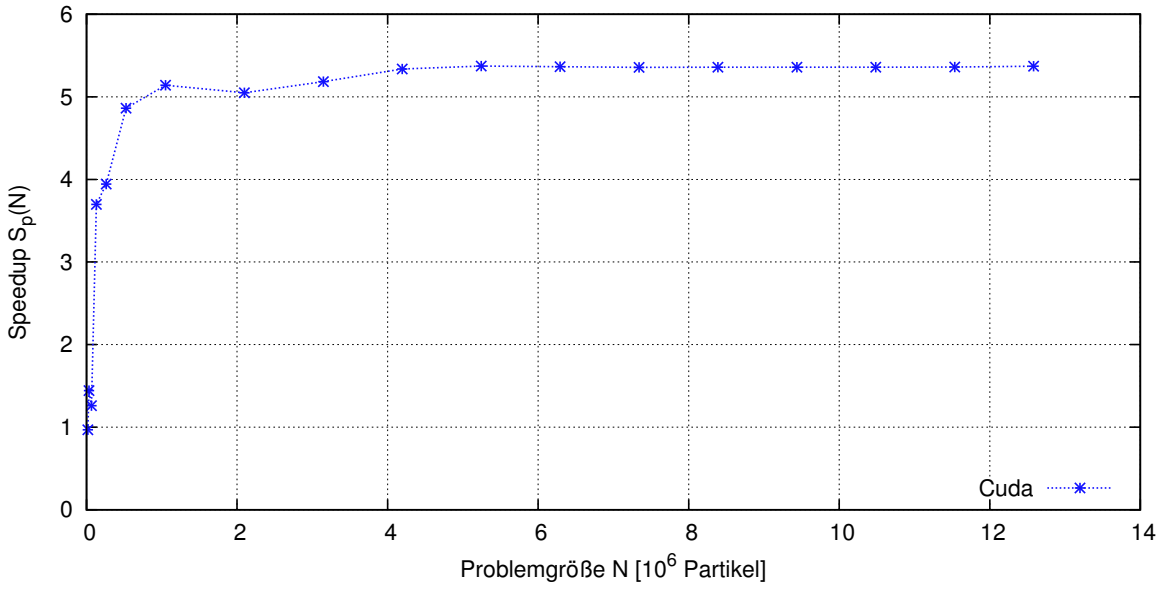


Abbildung 5.10: Speedup der Berechnung der Pseudopartikel mit CUDA im Vergleich zur rein sequentiellen Berechnung auf TheCell

Auf dem Testsystem TheCell konnten mit der Parallelisierung der Pseudopartikel auf der Graphikkarte im Vergleich zur sequentiellen CPU-Implementierung mit TreeArray Speedups von über 5 erreicht werden. Der Speedup für Probleme mit weniger als 10^6 Partikeln war dagegen deutlich geringer, stieg allerdings mit wachsender Problemgröße schnell an. Für größere Probleme mit mehr als 10^6 Partikeln war sehr schnell eine Sättigung des erreichbaren Speedups festzustellen. In Abbildung 5.10 ist der erreichte Speedup für unterschiedliche Problemgrößen dargestellt.

Insgesamt ist die Berechnung von Pseudopartikeln, aufgrund des hohen Maßes an Datenabhängigkeiten, nur begrenzt für die Parallelisierung auf Graphikkarten geeignet.

5.8 Implementierung der Kraftauswertung

Der zentrale und auch zeitaufwändigste Teilschritt des Barnes-Hut-Verfahrens ist die Auswertung der Kräfte, die auf die einzelnen Partikel einwirken. Dazu wird für jedes Partikel eine Baumtraversierung durchgeführt.

Es wurden neben der sequentiellen Variante der Kraftauswertung eine parallele CPU-Version, eine GPU-Version und eine Hybrid-Version, die gleichzeitig auf CPU und Graphikkarte Berechnungen durchführt, entworfen.

Eine Parallelisierung ist prinzipiell recht einfach zu realisieren, da die Kraftauswertungen für die einzelnen Partikel komplett unabhängig voneinander ablaufen können. Allerdings kann sich der Aufwand der Kraftauswertung für unterschiedliche Partikel, je nach deren Lage im Raum und der Verteilung der Partikel, stark unterscheiden. Daher können bei der Aufteilung der Partikel in parallelen Implementierungen verschiedene Verfahren zur Lastverteilung eingesetzt werden.

5.8.1 Sequentielle Kraftauswertung

Die sequentielle Kraftauswertung wird durch eine einfache for-Schleife über alle Partikel realisiert, in deren Rumpf für jedes einzelne Partikel eine iterative Baumtraversierung über Next- und More-Arrays durchgeführt wird.

5.8.2 Parallele Kraftauswertung mit OpenMP

Die parallele CPU-Implementierung partitioniert bei Einsatz von p Prozessoren die N Partikel im einfachsten Fall in p gleichgroße Teilmengen zusammenhängender Partikel der Größe $\frac{N}{p}$, die dann jeweils von einem Prozessor komplett unabhängig von den anderen Prozessoren bearbeitet werden. Bei ungleichmäßigen Partikelverteilungen, wie sie z.B. bei der Kollision von Galaxien auftreten, müssen für die einzelnen Partikel deutlich voneinander abweichende Anzahlen von Interaktionen berechnet werden. In diesem Fall kann die beschriebene statische Lastverteilung, wie es auch beim Baumaufbau geschehen kann, dazu führen, dass einzelne Prozessorkerne stark ausgelastet sind, während andere die ihnen zugewiesenen Partikel bereits vollständig bearbeitet haben. Es sollte daher versucht werden, die Anzahl von Interaktionen und damit die Rechenlast, die Partikel erzeugen, vorherzusagen und Partikel, der zu erwartenden Rechenlast entsprechend, auf Prozessoren zu verteilen.

5.8.3 Dynamische Lastverteilung

Anstatt einer statischen Lastverteilung, die davon ausgeht, dass pro Partikel gleichviel Zeit für die Kraftauswertung erforderlich ist, kann eine Lastverteilung eingesetzt werden, die versucht aus Daten vorheriger Iterationen oder anderer Eigenschaften der Partikel Vorhersagen darüber zu treffen, wie teuer die Kraftauswertung für einzelne Partikel sein könnte.

Eine einfache Vorhersage kann daraus getroffen werden, wie viele Interaktionen für Partikel im letzten Zeitschritt berechnet werden mussten, da sich die Partikelverteilung mit der Zeit nur langsam verändert. Das Zählen der Interaktionen erfordert allerdings pro Interaktion eine Addition, also insgesamt einen Zusatzaufwand von $\mathcal{O}(N \log N)$ Additionen. Als Alternative wird im nächsten Abschnitt ein Ansatz vorgestellt, der allein aus Eigenschaften der erzeugten Baumstruktur effizient Vorhersagen über die zu erwartende Anzahl von Interaktionen für Partikel treffen kann. Mit Hilfe dieser Vorhersagen wird anschließend die dynamische Lastverteilung der Partikel realisiert.

Abschätzung der Interaktionen der einzelnen Partikel im aktuellen Zeitschritt

Es kann für die dynamische Lastverteilung eine Heuristik verwendet werden, die Eigenschaften der einzelnen Partikel zur Lastabschätzung ausnutzt, welche allein durch den Aufbau der Baumstruktur im selben Zeitschritt bestimmt werden können. Die entscheidende Größe der Heuristik ist dabei das Baumlevel, in dem das Blatt liegt, in das ein Partikel einsortiert ist. Je tiefer im Baum ein Partikel einsortiert ist, desto größer ist die zu erwartende Anzahl von zu berechnenden Interaktionen, da das Nahfeld des Partikels viele Partikel umfassen muss und auch sehr viele Interaktionen mit kleinen Pseudopartikeln zu erwarten sind. Partikel hingegen, die in niederen Baumleveln einsortiert sind, sind in einem eher dünnbesetzten Gebiet lokalisiert und dementsprechend wenige Partikel umfasst das Nahfeld und viele Partikel im Fernfeld können durch große Pseudopartikel angenähert werden. Abbildung 5.11 veranschaulicht den Zusammenhang zwischen dem Level eines Partikels im Baum und der zu erwartenden Anzahl von Interaktionen dieses Partikels im selben Zeitschritt.

Überlegt man sich, dass das Level im Baum, auf dem ein Partikel eingefügt wird, in der Größenordnung von $\mathcal{O}(\log N)$ liegt und N Partikel bearbeitet werden, liefert die Baumlevel-Heuristik also auch für die Gesamtkomplexität von $\mathcal{O}(N \log N)$ plausible Voraussagen.

Realisierung der dynamischen Lastverteilung zur Partitionierung der Partikelmenge

Die Lastabschätzung für die einzelnen Partikel wird bereits während der Linearisierung, parallel zur Berechnung des Indirektions-Arrays für die Partikelsortierung, bestimmt. Es wird dazu ein weiteres Array geschrieben, wobei jeder Array-Eintrag mit einem der Partikel

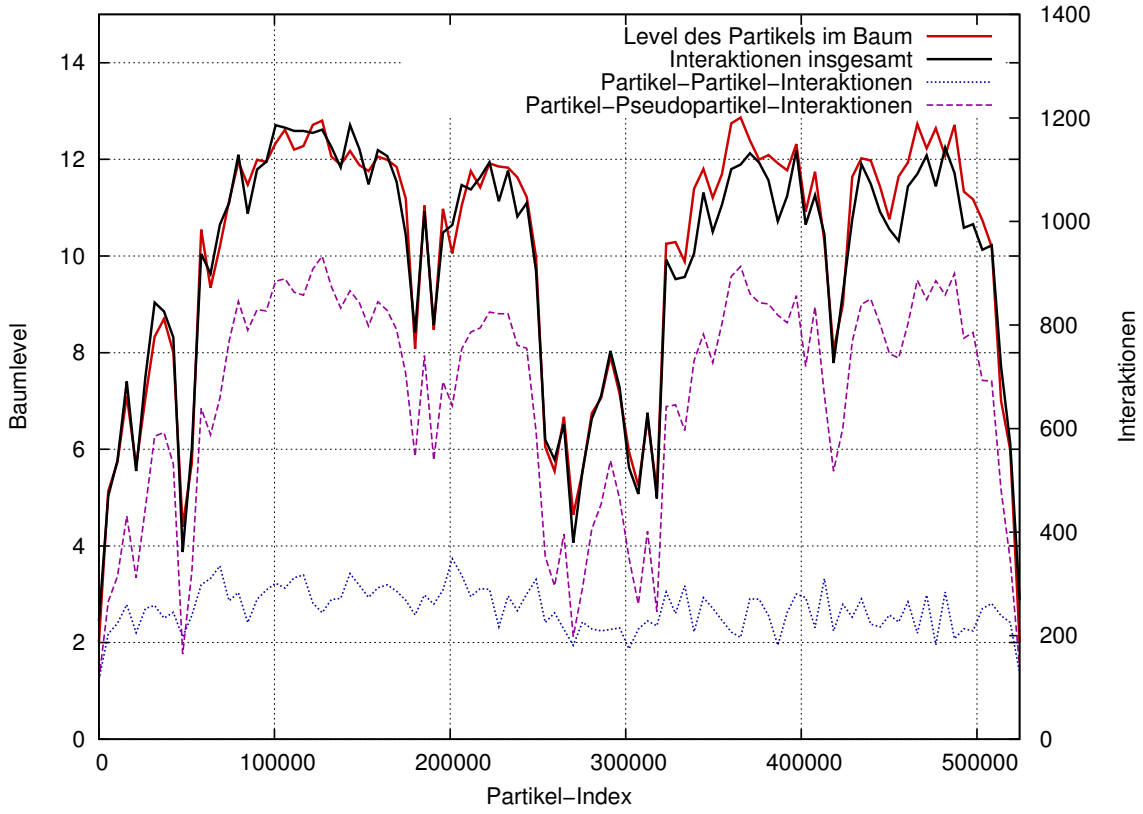
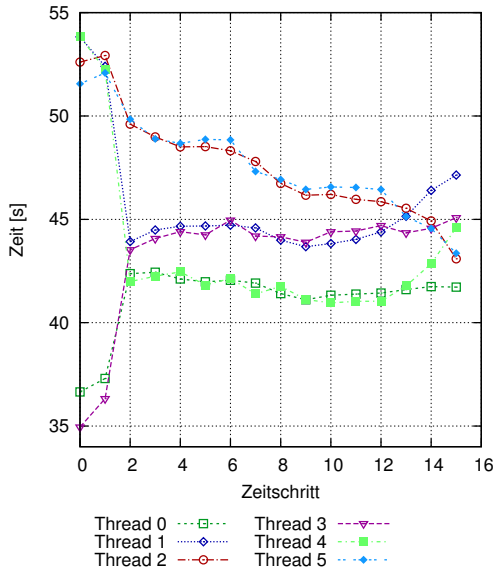


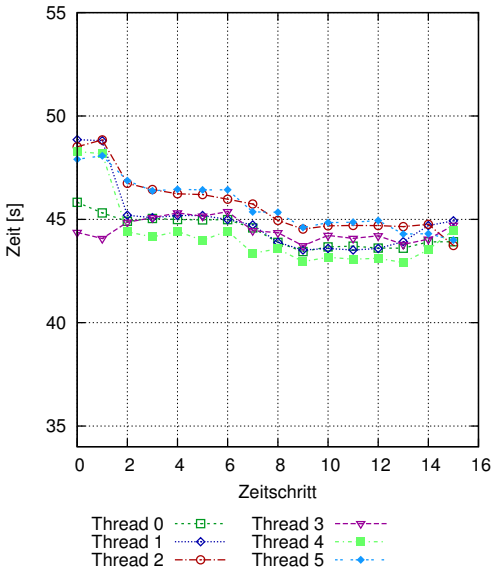
Abbildung 5.11: Vergleich der Anzahl von Interaktionen pro Partikel mit dem Level im Baum, in dem die Partikel jeweils einsortiert sind, für einen Zeitschritt. Alle Daten sind in Form von Bézier-Splines geglättet dargestellt. Die Daten entstammen dem ersten Zeitschritt der Simulation mit 524288 Partikeln aus Abbildung 2.2 auf Seite 18 mit $\theta = 0{,}75$.

in der Sortierreihenfolge der raumfüllenden Kurve des Baums identifiziert ist. Im Array wird dann die Präfix-Summe der Level der Partikel im Baum eingetragen, sodass die Werte im Array mit dem Index monoton steigen. Jeder Eintrag zu einem Index k im Array lässt sich als die Abschätzung der Rechenlast interpretieren, die durch Kraftauswertung aller Partikel $0 \ldots k$ entsteht. Der Array-Eintrag an Index $N - 1$ entspricht der zu erwartenden Gesamtlast, die im Folgenden als L bezeichnet wird.

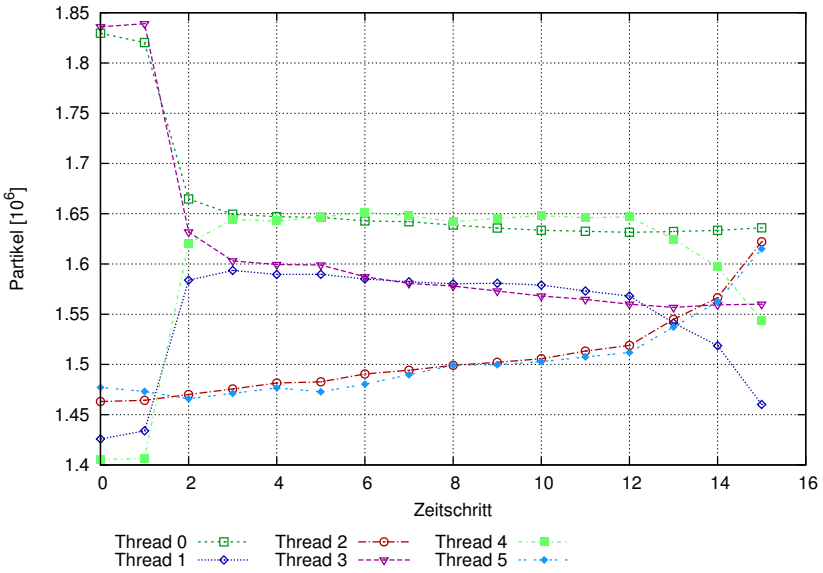
Um nun die Aufteilung der Partikel auf die einzelnen Threads zu erreichen, wird die eben beschriebene Abschätzung der Gesamtlast verwendet. Angenommen, p sei wieder die Anzahl eingesetzter Prozessoren bzw. Threads, dann wird die Partikelpartition dadurch bestimmt, dass die Last L in p gleichgroße Abschnitte zerlegt wird, sodass die Last der einzelnen Teilmengen von Partikeln jeweils mit $\frac{L}{p}$ angegeben werden kann.



(a) Verwendung statischer Lastverteilung



(b) Verwendung dynamischer Lastverteilung



(c) Partitionierung der Partikel bei dynamischer Lastverteilung

Abbildung 5.12: Vergleich der Laufzeit einzelner Threads bei Verwendung von statischer und dynamischer Lastverteilung der parallelen Kraftauswertung mit OpenMP auf TheCell über 16 Zeitschritte mit $\theta = 0{,}75$ für ein Problem mit $N = 9437184$ Partikeln.

Um die Partikelpartition zu bestimmen, die einem Prozessor $i \in 1 \ldots p$ zugewiesen werden soll, werden die Indizes der beiden Werte $(i-1) \cdot \frac{L}{p}$ und $i \cdot \frac{L}{p}$ durch binäre Suche im Last-Array ermittelt. Dies liefert zwei Array-Indizes, die gleichzeitig die Indizes der unteren und oberen Grenzen der zu berechnenden Partikelteilmenge für Prozessor i darstellen.

In Abbildung 5.12 ist die Laufzeit der einzelnen Threads einer Simulation auf sechs Prozessoren über 16 Zeitschritte hinweg mit statischer und dynamischer Lastverteilung im Vergleich zu sehen. Zudem ist die Partikelverteilung, die der dynamischen Lastverteilung zugrunde lag, abgebildet. Es ist deutlich zu erkennen, dass die Laufzeiten der einzelnen Threads bei dynamischer Lastverteilung deutlich homogener waren, als die Laufzeiten, die die einzelnen Threads bei statischer Lastverteilung aufwiesen. Da die Gesamtlaufzeit stets durch den langsamsten Thread dominiert wurde, konnte durch die dynamische Lastverteilung eine deutliche Beschleunigung der Kraftauswertung erreicht werden. In Abbildung 5.13 sind die

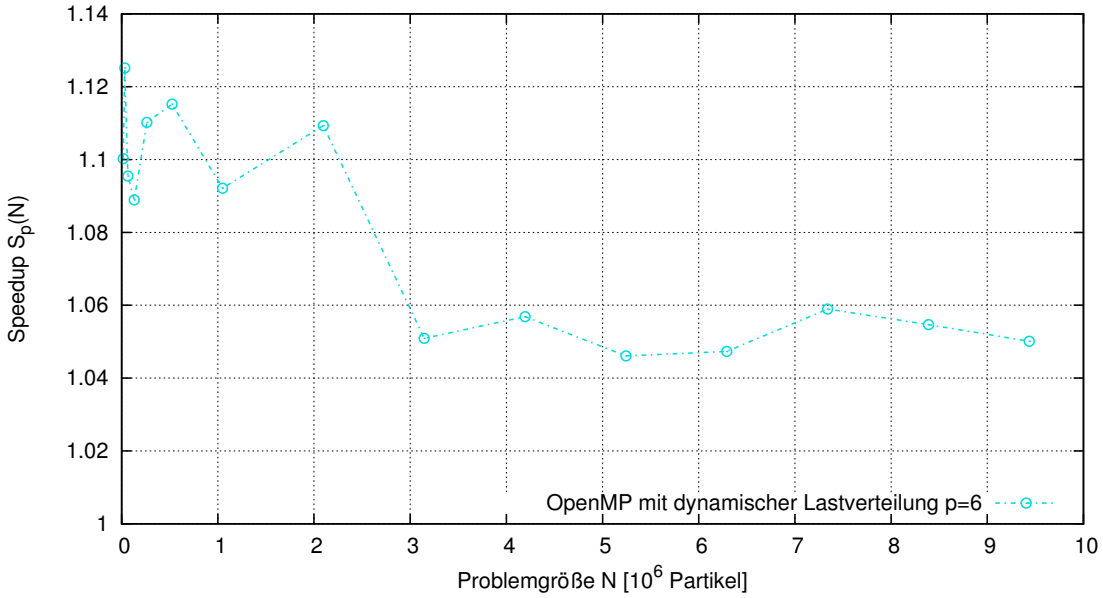


Abbildung 5.13: Speedup der parallelen CPU-basierten Kraftauswertung durch dynamische Lastverteilung auf TheCell im Vergleich zur parallelen Implementierung mit statischer Lastverteilung. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$.

zugehörigen Speedups für unterschiedliche Problemgrößen dargestellt, die durch die oben beschriebene dynamische Lastverteilung im Vergleich zur statischen Lastverteilung erreicht werden konnten.

5.8.4 Parallele Kraftauswertung auf der Graphikkarte mit CUDA

Neben den sequentiellen und parallelen Implementierungen für den Hauptprozessor wurde auch eine parallele Implementierung für die Ausführung auf der Graphikkarte entworfen. Es wird zur Bestimmung der Kräfte pro Partikel auf der Graphikkarte ein Thread gestartet und in jedem Thread eine Baumtraversierung durchgeführt. Die Partikel werden so auf die Threads verteilt, dass stets direkt benachbarte Partikel in Warps zur Ausführung zusammengefasst werden können. Da die Partikel nach raumfüllenden Kurven sortiert vorliegen, kann erwartet werden, dass die Baumtraversierungen der Partikel eines Warps ein großes Maß an Kohärenz aufweisen und somit alle Streaming Prozessoren der einzelnen Multiprozessoren die meiste Zeit beschäftigt sind.

5.8.5 Hybrid-Kraftauswertung auf Graphikkarte und CPU parallel

Da die hier beschriebenen Implementierungen der Kraftauswertung, egal ob auf CPU oder auf GPU ausgeführt, im Verhältnis zur Laufzeit des Gesamtalgorithmus' stets mit Abstand die meiste Zeit in Anspruch nehmen, lässt sich mit einer Beschleunigung der Kraftauswertung auch am ehesten eine Verbesserung der Gesamtleistung des Algorithmus' erzielen. Bei den Implementierungen, die entweder nur auf CPU oder nur auf GPU operieren, verharrt die jeweils andere Plattform komplett im Stillstand, und ein großer Teil der Rechenressourcen bleibt ungenutzt. Daher wurde in der vorliegenden Arbeit eine Variante der Kraftauswertung implementiert, die die Auswertung der Kräfte so auf GPU und CPU verteilt, dass beide möglichst ausgelastet sind und die Kraftauswertung insgesamt beschleunigt wird. Es werden dazu die Partikel in zwei Teilmengen partitioniert. Die erste Teilmenge wird auf der CPU von mehreren, mittels Pthreads erzeugten, CPU-Threads bearbeitet, während für die zweite Teilmenge auf der Graphikkarte Kräfte berechnet werden.

Die Lastverteilung zwischen den einzelnen CPU-Threads erfolgt entweder dynamisch, nach der oben beschriebenen Heuristik, oder statisch. Auf die Lastverteilung zwischen GPU und CPU wird im nächsten Abschnitt genauer eingegangen.

Dynamische Lastverteilung zwischen Hauptprozessor und Graphikkarte

Die Lastverteilung zwischen CPU und GPU gestaltet sich etwas aufwändiger als die Verteilung zwischen einzelnen CPU-Threads, da sich die Vorhersage der Laufzeit allein aus Baumeigenschaften nur für Threads gut eignet, die auf gleichartigen Plattformen ausgeführt werden. Stattdessen wird für die Hybrid-Kraftauswertung eine Lastbalancierung eingesetzt, die das Verhältnis zwischen Partikeln, die auf GPU und CPU berechnet werden sollen, dynamisch anpasst. Dazu werden die Partikelverteilung aus dem letzten Zeitschritt und die Dauer des langsamsten CPU-Threads und des GPU-Threads herangezogen. Die GPU wird für die Lastverteilung als Prozessor variabler Rechenleistung modelliert. Auf dem Hauptprozessor

werden Berechnungen in einer fest vorgegebenen Anzahl von Threads durchgeführt, auf CPU und GPU zusammen wird dagegen eine variable Anzahl *virtueller Threads* ausgeführt. Die Anzahl dieser virtuellen Threads wird dann in jedem Zeitschritt neu bestimmt, um die Lastverteilung zwischen Hauptprozessor und Graphikkarte zu realisieren.

Wenn CPU und GPU gleichviel Zeit für ihre Berechnung in Anspruch nehmen, bleibt die Anzahl virtueller Threads konstant. Wird dagegen mehr Zeit in den CPU-Threads als im GPU-Thread verbracht, wird das GPU:CPU-Verhältnis der zu berechnenden Partikel erhöht, indem die Anzahl der virtuellen Threads erhöht wird. Benötigt der GPU-Thread mehr Zeit als der langsamste CPU-Thread, kann die Anzahl virtueller Threads dagegen verringert werden. Im Falle einer Aktualisierung der Anzahl der virtuellen Threads wird die neue Anzahl jeweils durch eine Linearkombination aus der alten Anzahl und dem Verhältnis der Laufzeiten auf GPU und CPU bestimmt.

Gleichung 5.1 zeigt, wie die drei unterschiedlichen Fälle für die Anpassung der Lastverteilung zwischen GPU und CPU aus den Laufzeiten und der Anzahl virtueller Threads des letzten Zeitschritts berechnet werden können.

$$\text{Threads}_{t+\delta t} = \begin{cases} \text{Threads}_t \cdot \left(c_{inc} \cdot \frac{t_{CPU}}{t_{GPU}} + (1 - c_{inc}) \right) & \text{wenn } \frac{t_{GPU}}{t_{CPU}} < 1 \\ \text{Threads}_t & \text{sonst} \\ \text{Threads}_t \cdot \left(c_{dec} \cdot \frac{t_{CPU}}{t_{GPU}} + (1 - c_{dec}) \right) & \text{wenn } \frac{t_{GPU}}{t_{CPU}} > 1 \end{cases} \tag{5.1}$$

Die Konstanten c_{inc} und c_{dec} gewichten jeweils, wie stark die Partikelverteilung des letzten Schritts verändert werden soll und sind zwischen 0 und 1 zu wählen. Für c_{inc} wurde für die weiter unten beschriebenen Testläufe 0,5 gewählt, damit die GPU möglichst schnell mehr Last erhält, wenn sie unterausgelastet ist, da sie in den eingesetzten Testsystemen bei der Kraftauswertung deutlich schneller arbeitet als die CPU. Dementsprechend wurde der Wert für c_{dec} mit 0,1 deutlich geringer festgelegt, sodass die CPU-Last nur langsam erhöht und eine Verteilung, in der die CPU-Last größer wird als die GPU-Last, vermieden wird.

Da die Kräfte nach der Auswertung auf GPU und CPU in verteilten Speichern liegen, entstehen notwendigerweise Speicherlatenzen, um die berechneten Kräfte zusammenzuführen. Diese Latenzen können aber dadurch versteckt werden, dass die erwartete Latenz entweder zu t_{GPU} oder t_{CPU} addiert wird, je nachdem auf welcher Plattform die Zeitintegration erfolgt. Dann wird die Last so verteilt, dass Datenübertragung und Kraftauswertung auf der Zielplattform zeitgleich abgeschlossen werden.

5.8.6 Ergebnisse und Diskussion der verschiedenen Implementierungen zur Kraftauswertung

Im Folgenden sollen zunächst die Ergebnisse der rein CPU-basierten Implementierungen vorgestellt und diskutiert werden. Später folgen dann die Ergebnisse der durch Graphikkarten beschleunigten Implementierungen.

Ergebnisse und Diskussion der CPU-basierten Implementierungen der Kraftauswertung

Ein Vergleich der parallelisierten Kraftauswertung mit OpenMP auf nur einem Prozessorkern mit der sequentiellen Kraftauswertung zeigte zunächst, dass, wie zu erwarten war, keine Unterschiede bei der Ausführungsdauer auftraten, da keinerlei Sperrvariablen bei der Parallelisierung eingesetzt wurden.

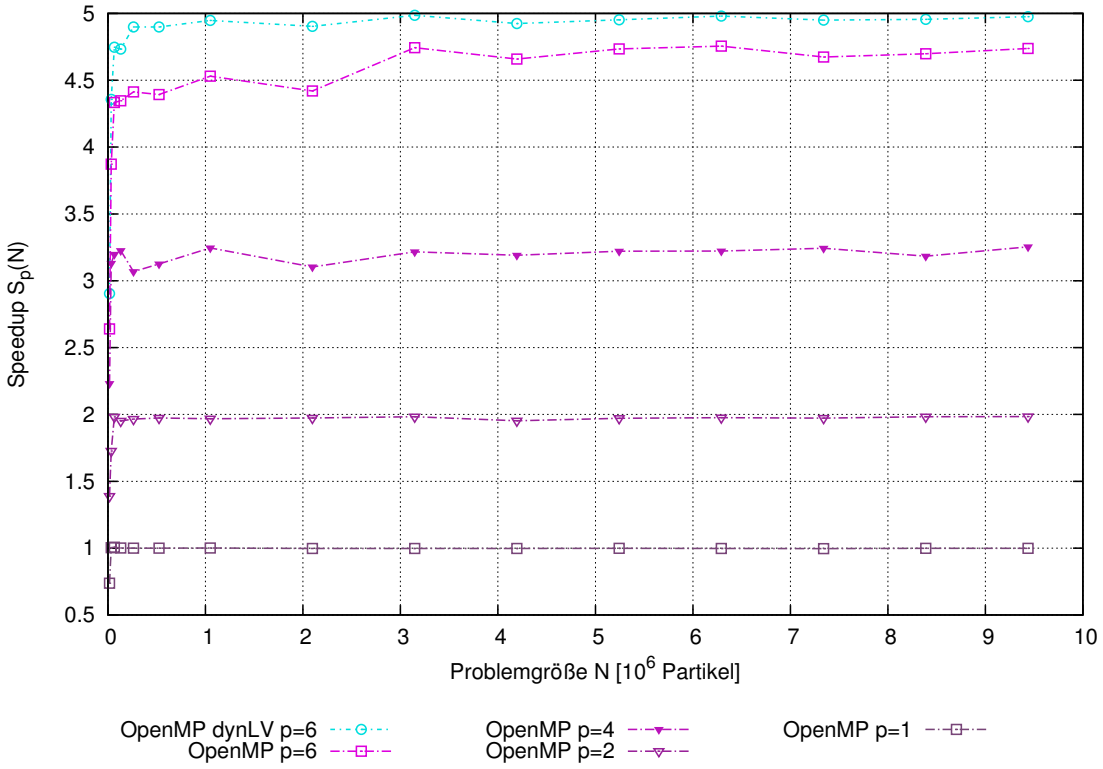


Abbildung 5.14: Speedup unterschiedlicher paralleler Implementierungen zur Kraftauswertung auf TheCell im Vergleich zur rein sequentiellen Implementierung. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$.
dynLV: Implementierung mit dynamischer Lastverteilung

Bei Verwendung mehrerer Prozessorkerne wurde bereits für sehr kleine Problemgrößen eine Sättigung des Speedups erreicht. Auf TheCell waren mit sechs Prozessorkernen bei statischer Lastverteilung Speedups von über 4,5 zu erreichen, wie in Abbildung 5.14 nachvollzogen werden kann. Durch Einsatz der oben beschriebenen dynamischen Lastverteilung konnten die Speedups für alle Problemgrößen nochmals um mindestens 4–5 % gesteigert werden, wie bereits in Abschnitt 5.8.3 gezeigt wurde.

Die Speedups, die durch Parallelisierung auf sechs Prozessorkernen mit dynamischer Lastverteilung erreicht wurden, nahmen für die unterschiedlichen Problemgrößen sehr ähnliche Werte an. Bei statischer Lastverteilung waren dagegen deutlich stärkere Schwankungen bei den Speedups zu beobachten. Dies lässt sich mit einer deutlich größeren Abhängigkeit von den Eingabedaten bei statischer Lastverteilung erklären. Diese Abhängigkeit konnte durch die dynamische Lastverteilung weitgehend ausgeglichen werden.

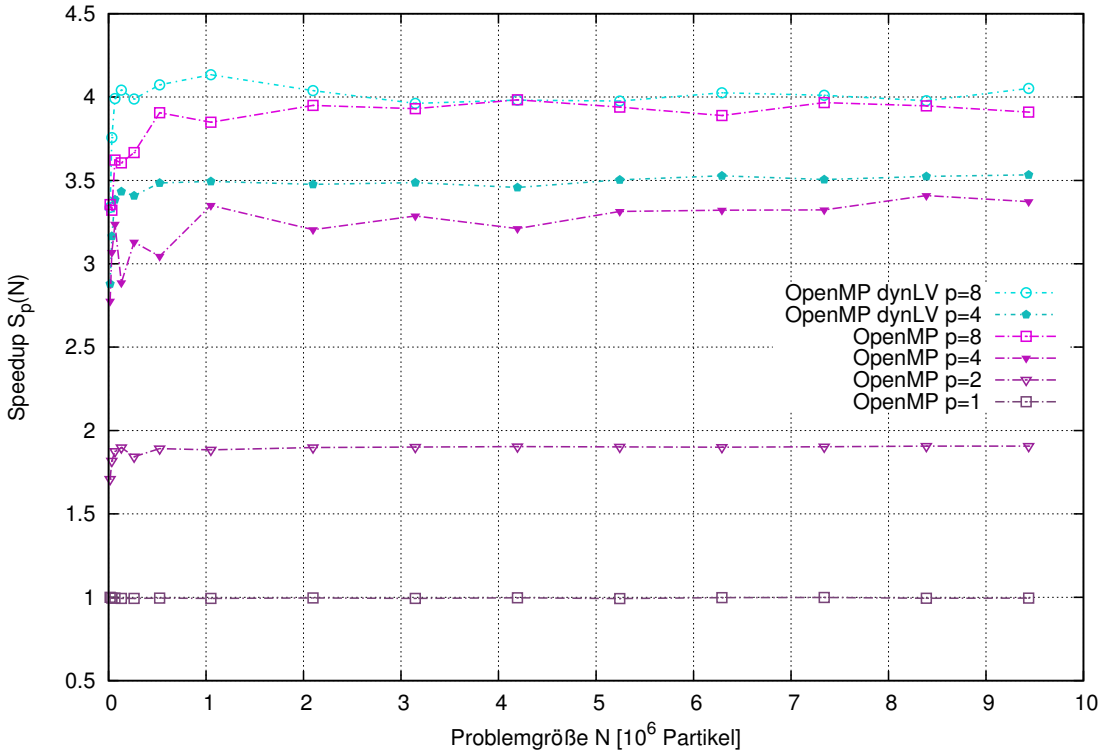


Abbildung 5.15: Speedup unterschiedlicher paralleler Implementierungen zur Kraftauswertung auf StarCluster im Vergleich zur rein sequentiellen Implementierung. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$.
dynLV: Implementierung mit dynamischer Lastverteilung

Auf StarCluster ließen sich für die Parallelisierung mit einem, zwei und vier Prozessorkernen zunächst sehr ähnliche Ergebnisse wie auf TheCell erzielen. Die Ergebnisse für die Ausführung auf acht logischen Prozessorkernen wichen allerdings von den übrigen Resultaten ab. Die Speedups, die durch dynamische gegenüber statischer Lastverteilung erreicht wurden, waren bei acht Prozessorkernen deutlich geringer als bei vieren. Abbildung 5.15 zeigt die zugehörigen Daten.

Ergebnisse und Diskussion der Implementierungen der Kraftauswertung mit GPU-Beschleunigung

Durch den Einsatz von Graphikkarten konnten im Vergleich zur rein sequentiellen Berechnung der Kräfte sehr starke Beschleunigungen erreicht werden, wie den Abbildungen 5.16 und 5.17 entnommen werden kann. Die Kraftauswertung durch die Graphikkarte

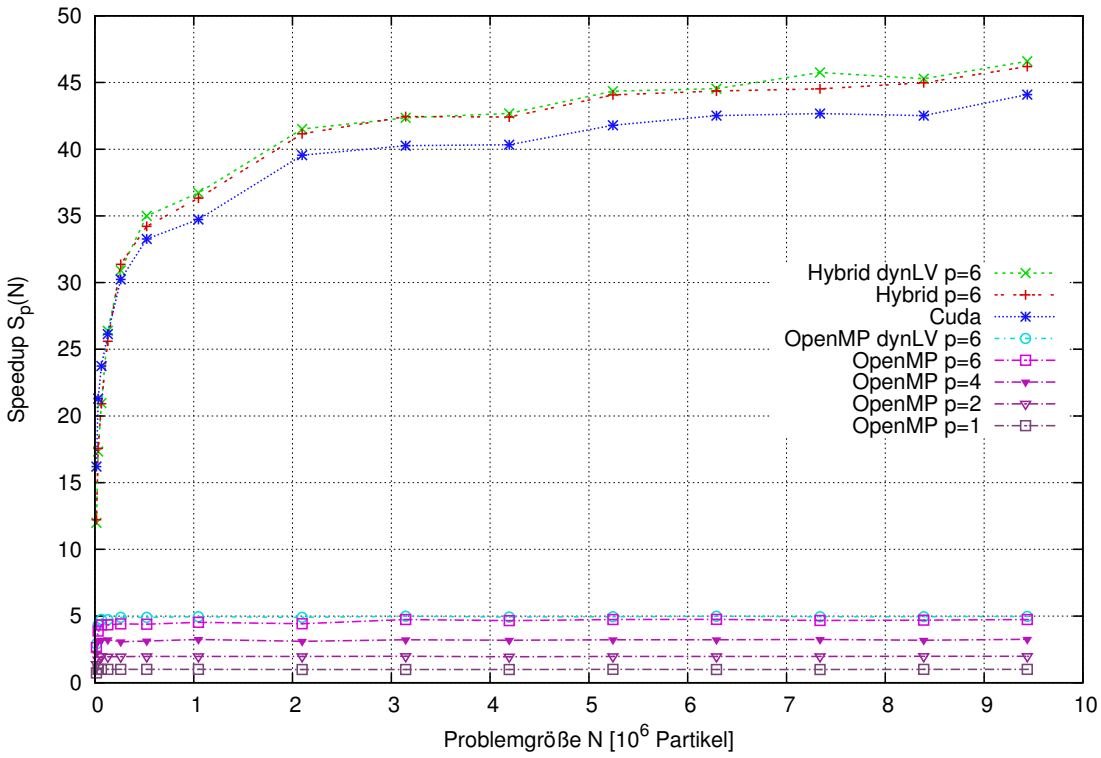


Abbildung 5.16: Speedup der unterschiedlichen parallelen Implementierungen der Kraftauswertung im Vergleich zur rein sequentiellen Implementierung auf TheCell. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$.
dynLV: Implementierung mit dynamischer Lastverteilung

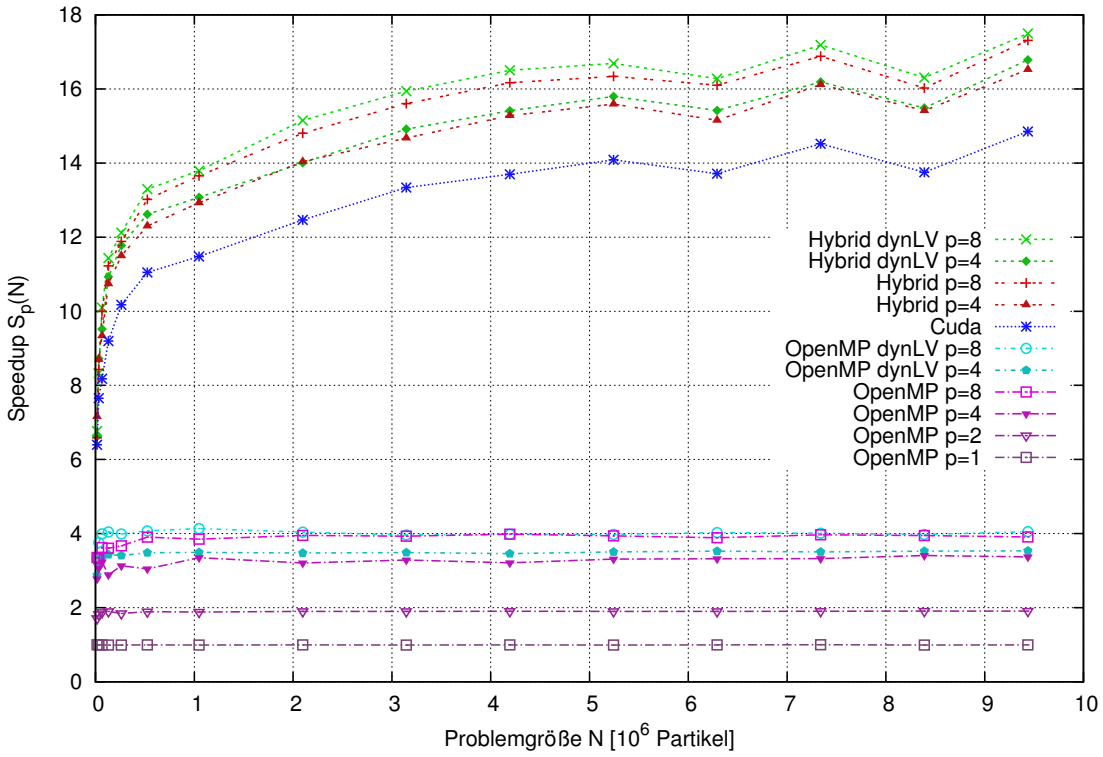


Abbildung 5.17: Speedup der unterschiedlichen parallelen Implementierungen der Kraftauswertung im Vergleich zur rein sequentiellen Implementierung auf StarCluster. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$. dynLV: Implementierung mit dynamischer Lastverteilung

auf TheCell erreichte für die größten getesteten Probleme Speedups von über 40. Durch Hybrid-Kraftauswertung auf Graphikkarte und allen sechs Prozessorkernen mit dynamischer Lastverteilung konnten sogar Speedups von über 45 erreicht werden. Der Speedup der GPU-basierten Kraftauswertung stieg dabei recht langsam mit der Problemgröße an. Eine Sättigung schien selbst bei den größten Problemen noch nicht erreicht.

Auf StarCluster war der Speedup durch die Verlagerung der Kraftauswertung auf die Graphikkarte deutlich geringer, da die Unterschiede in der Rechenleistung zwischen GPU und CPU auf StarCluster weniger groß sind als auf TheCell. Aufgrund dieser Tatsache ließen sich durch die Hybrid-Kraftauswertung mit acht logischen Prozessoren und dynamischer Lastverteilung Speedups von deutlich über 15 % gegenüber der reinen GPU-basierten Kraftauswertung erreichen. In Abbildung 5.18 ist dieser Sachverhalt nachzuvollziehen. Auf TheCell wurden durch den Hybrid-Modus immerhin über 5 % Speedup gegenüber der reinen GPU-Implementierung ermittelt.

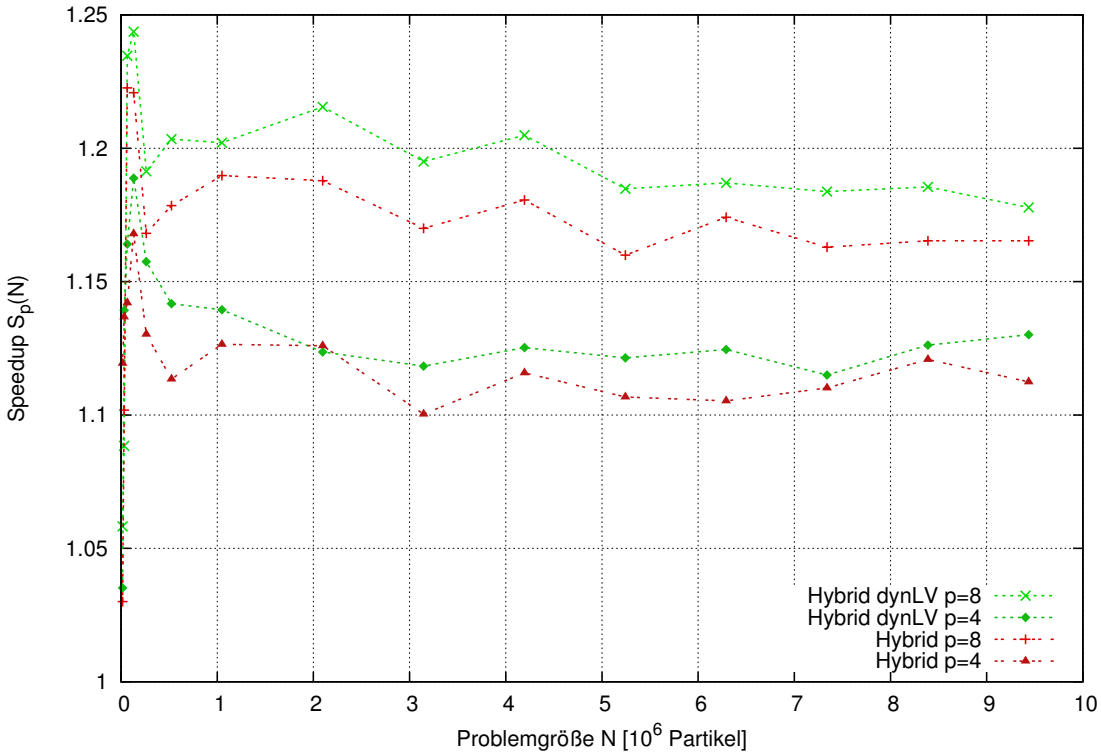


Abbildung 5.18: Speedup der Hybrid-Kraftauswertung durch parallelen Einsatz von GPU und CPU und dynamischer Lastverteilung im Vergleich zur reinen GPU-Implementierung auf StarCluster. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$.
dynLV: Implementierung mit dynamischer Lastverteilung

5.9 Aktualisieren von Partikelpositionen und -geschwindigkeiten

Nachdem alle Kräfte, die auf die einzelnen Partikel wirken, berechnet sind, können daraus nach unterschiedlichen Zeitintegrationsverfahren neue Geschwindigkeiten und Positionen für die Partikel errechnet werden. Es wurde in der vorliegenden Arbeit auf das Leapfrog-Verfahren zurückgegriffen.

Zur Aktualisierung der Partikelpositionen und -geschwindigkeiten wurden je eine sequentielle und parallele CPU-Version und eine parallele Version, die auf der Graphikkarte ausgeführt wird, implementiert. In der sequentiellen Implementierung wird die Aktualisierung der Partikel durch eine einfache Schleife über alle Partikel realisiert. Für die CUDA-Implementierung wird pro Partikel ein Thread ausgeführt, wobei stets direkt zusammenhängende Gruppen

von Partikeln in Blöcken zusammengefasst werden, sodass optimal reguläre Speicherzugriffsmuster entstehen.

Ergebnisse und Diskussion der Partikelaktualisierung

Es konnten vergleichbare Speedups durch den Einsatz der Graphikkarte gegenüber der sequentiellen CPU-Variante erreicht werden, wie es oben für die GPU-basierte Kraftauswertung zu beobachten war. Dies wird durch Abbildung 5.19 veranschaulicht. Insgesamt war die Aktualisierung der Partikel stets das am wenigsten zeitintensive Teilproblem.

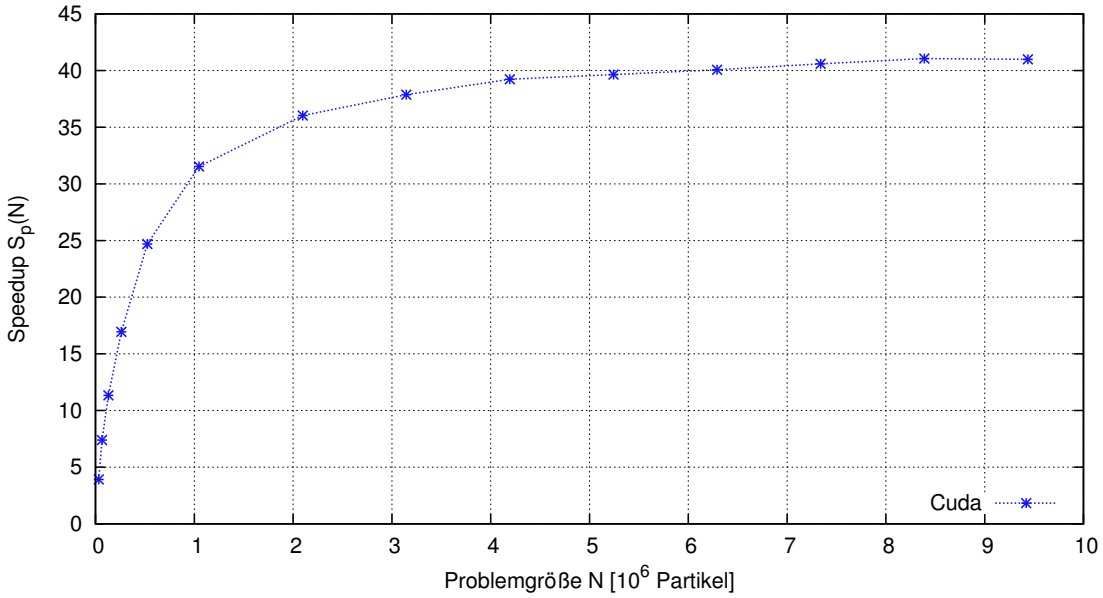


Abbildung 5.19: Speedup der Partikelaktualisierung durch die Graphikkarte im Vergleich zur sequentiellen CPU-Implementierung auf TheCell. Gemessen jeweils am Mittelwert aus 16 Zeitschritten.

5.10 Erzeugung von Anfangswerten und Validierung von Ergebnissen

Die Anfangswerte für alle durchgeführten Testläufe wurden mit Hilfe des Anfangswertegenerators StarScream [Bil] erzeugt. Zur Validierung der Simulationsergebnisse wurden visuelle Vergleiche mit den Ergebnissen von Gadget-2 [Spro5] durchgeführt.

6 Zusammenfassung und Diskussion der Ergebnisse

Wurden im letzten Kapitel bereits die Ergebnisse der einzelnen Modul-Implementierungen für jedes Teilproblem getrennt vorgestellt und diskutiert, soll in diesem Kapitel in Abschnitt 6.1 nun zunächst eine kurze Zusammenfassung der Teilergebnisse folgen. Davon ausgehend wird anschließend darauf eingegangen, wie der Gesamtalgorithmus aus den einzelnen Modul-Varianten zusammengesetzt werden kann und welche Ergebnisse in Abhängigkeit von unterschiedlichen Modul-Kombinationen erreicht werden konnten. In Abschnitt 6.2 wird gezeigt, wie sich die Auswahl der Modul-Implementierungen auf das Datentransfervolumen zwischen CPU und GPU und die Möglichkeiten des Versteckens von Speicherlatenzen auswirkten.

6.1 Zusammenfassung der Ergebnisse und Auswahl der optimalen Kombination von Modul-Implementierungen

Im letzten Kapitel konnte in Abschnitt 5.3 zunächst nachgewiesen werden, dass die Wahl der Datenstrukturen maßgeblichen Anteil an einer effizienten Berechnung von N-Body-Problemen trägt. Die Verwendung von Next- und More-Arrays stellte sich als die beste Datenstruktur für die Kraftauswertung heraus. Auch die Berechnung der Pseudopartikel wurde durch Einsatz einer Array-Struktur gegenüber der Verwendung zeigerbasierter Baumdarstellungen beschleunigt, weshalb alle Zeigerstrukturen stets linearisiert wurden.

Der Baumaufbau ließ sich wegen des hohen Maßes an Datenabhängigkeiten nur eingeschränkt auf Graphikkarten parallelisieren. Auch die Parallelisierung des Baumaufbaus auf dem Hauptprozessor erzielte geringere Speedups als die Parallelisierung der Kraftauswertung, wie in Abschnitt 5.4 beschrieben. Mit Hilfe dynamischer Lastverteilung konnten sowohl der Baumaufbau als auch die Kraftauswertung auf dem Hauptprozessor deutlich beschleunigt werden. Für alle anderen Teilprobleme konnten auf den eingesetzten Testsystemen große bis sehr große Geschwindigkeitssteigerungen durch den Einsatz von Graphikkarten nachgewiesen werden. Es konnte auch gezeigt werden, dass sich der Einsatz von Graphikkarte und Hauptprozessor zur Lösung von Teilproblemen gegenseitig keinesfalls ausschließen. Im Gegenteil konnte in Abschnitt 5.8 für die Kraftauswertung, die die Laufzeit des Barnes-Hut-Verfahrens insgesamt dominiert, durch Zusammenarbeit von GPU

und CPU eine deutliche Leistungssteigerung im Vergleich zu allen anderen vorgestellten Implementierungen nachgewiesen werden.

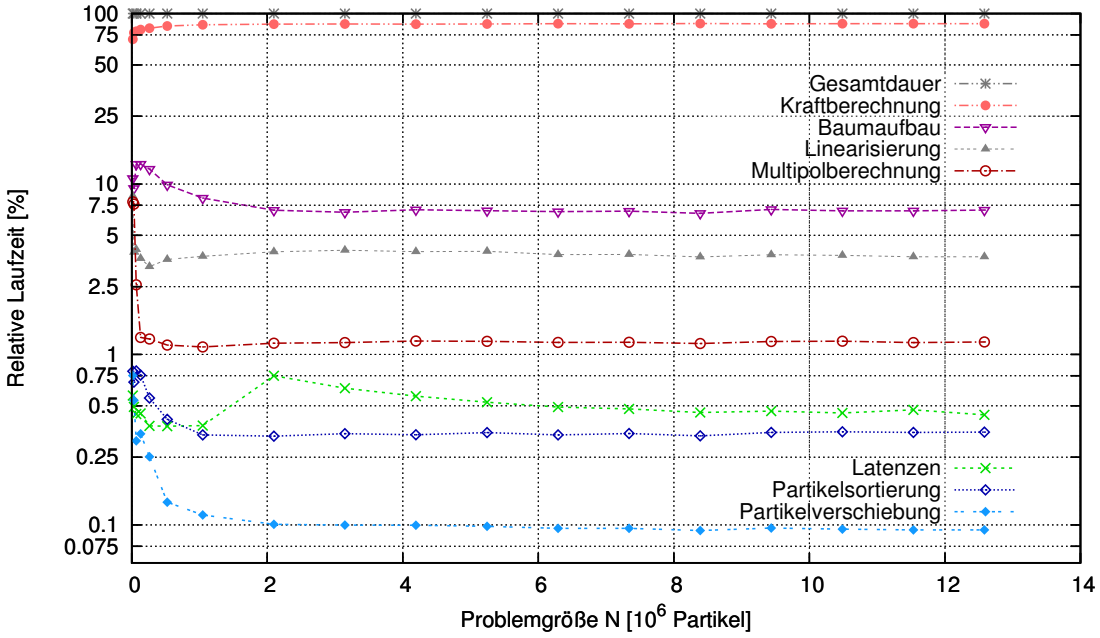


Abbildung 6.1: Anteil der einzelnen Module und Speicherlatenzen am Gesamtzeitaufwand der schnellsten Modulkombination auf TheCell. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$. Die Y-Achse ist logarithmisch skaliert.

Die optimale Modul-Kombination für alle relevanten Problemgrößen setzte sich aus den jeweils schnellsten Implementierungen der einzelnen Module zusammen. Wie im nächsten Abschnitt gezeigt wird, spielten Speicherlatenzen dabei nur eine untergeordnete Rolle. Im Einzelnen waren die schnellsten Implementierungen der Baumaufbau mit OpenMP-Parallelisierung auf Ebene einzelner Partikel mit dynamischer Lastverteilung und Hybrid-Kraftauswertung auf Graphikkarte und Hauptprozessor mit dynamischer Lastverteilung. Alle anderen Teilprobleme konnten durch Ausführung allein auf der Graphikkarte am besten beschleunigt werden. In Abbildung 6.1 ist dargestellt, welche Zeitanteile an der Gesamtausführungsdauer die einzelnen Module und die Speicherlatenz für unterschiedliche Problemgrößen auf TheCell aufwiesen.

In Abbildung 6.2 sind für die schnellste oben beschriebene Konfiguration und einige andere Konfigurationen zum Vergleich die Gesamtlaufzeiten pro Zeitschritt für unterschiedliche Problemgrößen aufgeführt. Für die Anfangswerte mit $12 \cdot 2^{20}$ Partikeln wurden pro Zeitschritt für die schnellste Konfiguration mit Hybrid-Kraftauswertung weniger als 8 s benötigt. Es wurde

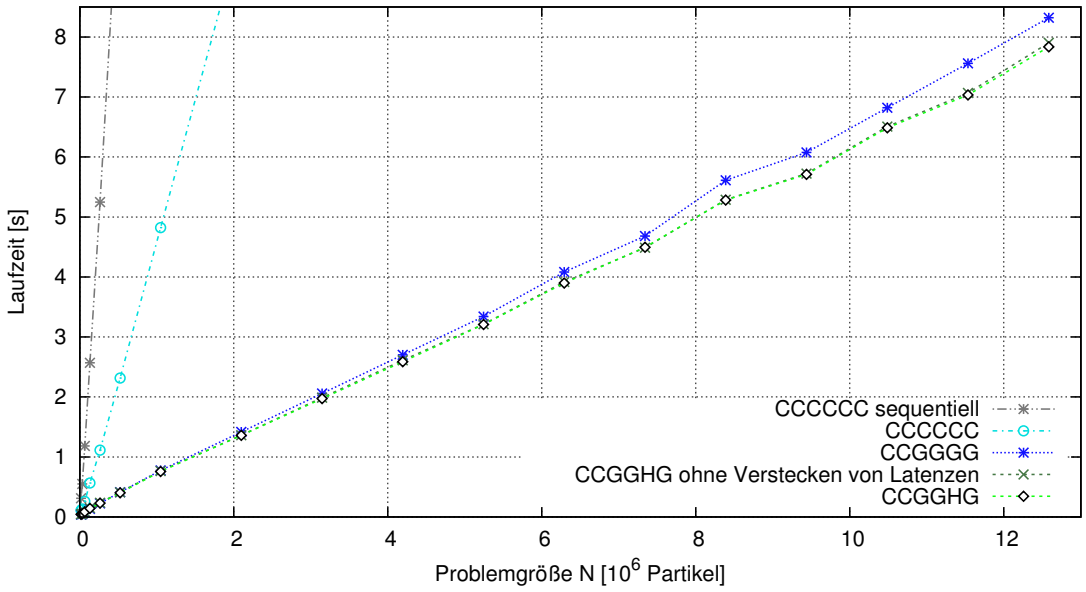


Abbildung 6.2: Gesamtlaufzeit pro Zeitschritt für unterschiedliche Kombinationen des Barnes-Hut-Algorithmus' auf TheCell. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$. Der Buchstabencode beschreibt jeweils die Plattformen, auf denen die einzelnen Module ausgeführt wurden. C = CPU, G = GPU, H = Hybrid. Für alle Kombinationen außer CCCCCC sequentiell wurde dynamische Lastverteilung auf sechs Prozessorkernen eingesetzt.

damit ein Speedup von über 40 gegenüber der rein sequentiellen CPU-Implementierung des Barnes-Hut-Verfahrens erreicht.

6.2 Speicheraufwand und Ergebnisse des Versteckens von Speicherlatenzen

In Abschnitt 5.1.2 konnte gezeigt werden, dass der Speicheraufwand des Barnes-Hut-Algorithmus' insgesamt nur linear von der Anzahl der Partikel abhängt. Dementsprechend ist auch nur mit einem linearen Transfervolumen pro Zeitschritt zu rechnen. Messungen auf TheCell in Abschnitt 5.2.1 hatten für die Übertragung von 1 GiB Daten auf TheCell eine Speicherlatenz von 200 ms ergeben. Zudem konnten in Abschnitt 5.2.2 Ansätze zum Verstecken von Speicherlatenzen aufgezeigt werden, die in der vorliegenden Implementierung des Barnes-Hut-Verfahrens umgesetzt wurden.

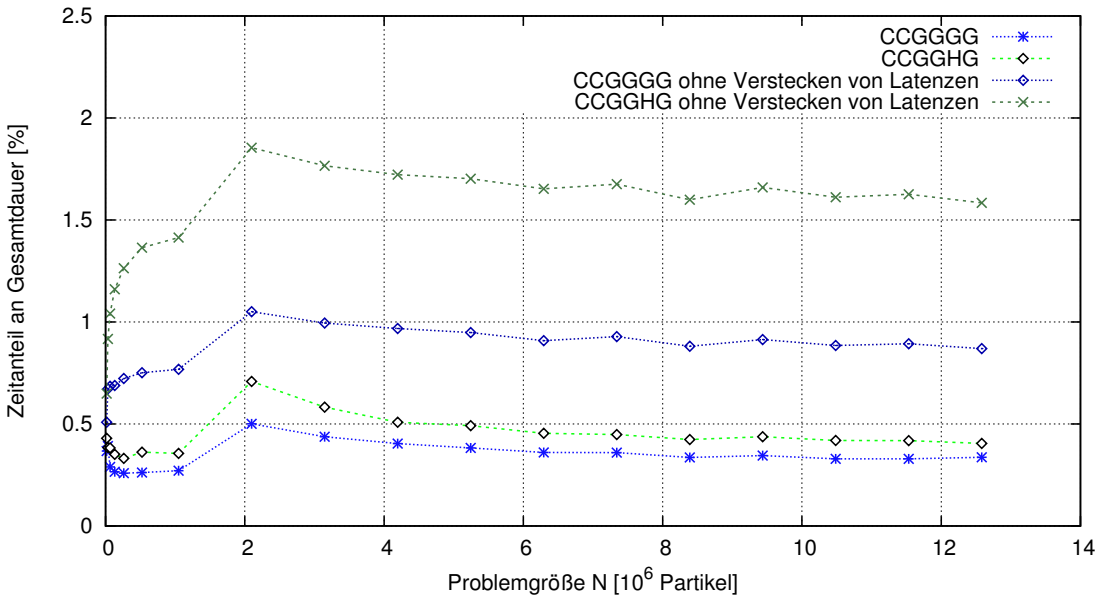


Abbildung 6.3: Anteil von Speicherlatenzen am Gesamtzeitaufwand unterschiedlicher Implementierungen mit und ohne Verstecken von Speicherlatenzen auf TheCell. Gemessen jeweils am Mittelwert aus 16 Zeitschritten mit $\theta = 0{,}75$. Der Buchstabencode beschreibt jeweils die Plattformen, auf denen die einzelnen Module ausgeführt wurden. C = CPU, G = GPU, H = Hybrid.

Abbildung 6.3 zeigt für zwei unterschiedliche Kombinationen des Barnes-Hut-Verfahrens die Auswirkungen des Versteckens von Speicherlatenzen. Zum einen ist zu erkennen, dass Speicherlatenzen ab $2 \cdot 2^{20}$ Partikeln mit wachsender Problemgröße stetig kleiner wurden, zum anderen ist zu sehen, dass durch das Verstecken von Latenzen der Anteil derselben an der Gesamtlaufzeit deutlich reduziert werden konnte. Für das in Abbildung 6.3 dargestellte größte Problem mit $12 \cdot 2^{20}$ Partikeln wurden auf der Graphikkarte 880 MiB Speicher allokiert. In jedem Zeitschritt entstand in der Kombination CCGGHG im Mittel ein Transfervolumen von 750 MiB. Durch Verstecken von Latenzen konnte die Speicherlatenz von ca. 125 ms auf unter 35 ms verringert werden. Für alle getesteten Problemgrößen machten Speicherlatenzen somit deutlich weniger als 1 % der Gesamtlaufzeit aus. Ab $5 \cdot 2^{20}$ Partikeln lag der Anteil der Speicherlatenzen an der Gesamtlaufzeit sogar unter 0,5 %. Obwohl für die Implementierung mit Hybrid-Kraftauswertung ein größerer Anteil von Speicherlatenzen als für andere Implementierungen auftrat, konnten mit diesem Ansatz die besten Ergebnisse erzielt werden. Dementsprechend spielten Speicherlatenzen eine untergeordnete Rolle bei der Auswahl der einzelnen Modul-Implementierungen.

7 Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurde eine effiziente parallele Implementierung des Barnes-Hut-Verfahrens zur näherungsweisen Berechnung von N-Body-Problemen vorgestellt. Dazu wurde das Barnes-Hut-Verfahren in mehrere Teilprobleme zerlegt, die getrennt voneinander gelöst werden können. Um festzustellen, welche Teilprobleme sich in welchem Maße für die Parallelisierung auf der Graphikkarte oder dem Hauptprozessor eignen, wurden unterschiedliche Lösungsansätze für die einzelnen Teilprobleme implementiert, die sich dann flexibel als Module zu einem Gesamtprogramm kombinieren ließen.

Als Einführung in die Thematik wurden zunächst die physikalischen Grundlagen der N-Body-Probleme vorgestellt. Es folgte ein Überblick über verschiedene algorithmische Lösungsansätze für N-Body-Probleme, in dem auch der Barnes-Hut-Algorithmus vorgestellt wurde. Um den Übergang von der sequentiellen Beschreibung zur parallelen Implementierung zu ermöglichen, wurden anschließend unterschiedliche Klassifikationen von Parallelrechnerarchitekturen eingeführt und Aufbau sowie Programmierung von Multicore-Systemen und Graphikkarten beschrieben.

Zur Implementierung des Barnes-Hut-Verfahrens wurde dieses zunächst in mehrere Teilprobleme zerlegt, deren Lösungen sich getrennt voneinander als Module umsetzen ließen. Für jedes Modul wurden anschließend unterschiedliche sequentielle und parallele CPU-Implementierungen und GPU-Implementierungen beschrieben und jeweils miteinander verglichen. Es zeigte sich dabei, dass sich insbesondere das zeitaufwändigste Teilproblem, die Kraftauswertung, sehr gut für eine Parallelisierung sowohl auf CPU als auch auf GPU eignet, wobei die Wahl der eingesetzten Datenstrukturen einen maßgeblichen Anteil an der effizienten Umsetzung hatte. Zudem konnte demonstriert werden, dass sich die Berechnungsgeschwindigkeit durch eine gemeinsame Berechnung der Kräfte auf dem Hauptprozessor und der Graphikkarte noch weiter steigern ließ.

Nach der Vorstellung der Implementierungen der verschiedenen Module wurden schließlich durch Kombination einzelner Implementierungen unterschiedliche Gesamtprogramme zusammengestellt und miteinander verglichen. Es stellte sich dabei heraus, dass das optimale Gesamtprogramm aus den jeweils besten Implementierungen der Teilproblemlösungen bestand. Anhand verschiedener Gesamtprogramme konnte zuletzt auch nachgewiesen werden, dass sich Speicherlatenzen sehr gut verstecken ließen, wobei diese auch ohne Verstecken nur einen kleinen Bruchteil des Gesamtaufwands ausmachten und somit zu vernachlässigen waren.

7.1 Ausblick

Der in [GBZ10] vorgestellte Ansatz, die Kraftauswertung auf der Graphikkarte durch stackbasierte Baumtraversierungen und den Einsatz von Interaktionslisten zu realisieren, führte zu sehr guten Ergebnissen. Es wäre spannend zu untersuchen, wie sich im Vergleich dazu eine Implementierung der Kraftauswertung auf der Graphikkarte mit Baumtraversierung durch Next- und More-Arrays und Interaktionslisten verhält und ob sich damit eine Beschleunigung gegenüber der Kraftauswertung auf Basis einzelner Partikel durch Next- und More-Arrays erreichen lässt.

Durch die Verlagerung der meisten Teilprobleme auf die Graphikkarte und durch die kooperative Berechnung der Kräfte auf Hauptprozessor und Graphikkarte konnten sehr deutliche Steigerungen der Ausführungsgeschwindigkeit gegenüber reinen CPU- und GPU-Implementierungen des Barnes-Hut-Verfahrens erreicht werden. Allerdings ließ sich die Kraftauswertung auf der GPU im Gegensatz zum Baumaufbau wesentlich besser parallelisieren, was zur Folge hatte, dass der Zeitanteil des Baumaufbaus am Gesamtzeitaufwand mit wachsendem Parallelisierungsgrad deutlich anstieg. Während des Baumaufbaus, unabhängig davon, ob dieser auf dem Hauptprozessor oder der Graphikkarte erfolgt, blieb die jeweils andere Plattform bisher gänzlich beschäftigungslos.

Mit einer pipelineartigen Berechnung können für Partikel, deren Kräfte bereits berechnet wurden, direkt neue Positionen und Geschwindigkeiten bestimmt werden. Für die aktualisierten Partikel kann anschließend, parallel zur Kraftauswertung und Aktualisierung der übrigen Partikel, bereits die Baumstruktur für den folgenden Zeitschritt aufgebaut werden, wobei sich die Abmessungen des Simulationsgebiets sehr gut aus den alten Partikelpositionen und -geschwindigkeiten vorhersagen lassen. Durch diesen Pipeline-Ansatz und die gemeinsame Kraftauswertung auf CPU und GPU könnte so eine sehr effiziente Umsetzung des Barnes-Hut-Verfahrens realisiert werden, bei der alle Rechenressourcen stets nahezu voll ausgelastet wären.

Die in dieser Arbeit vorgestellte Implementierung der Kraftauswertung des Barnes-Hut-Verfahrens weist ein äußerst hohes Maß an Flexibilität auf, sodass auf Systemen mit beliebigen Kombinationen von Graphikkarten und Hauptprozessoren eine sehr gute Ausnutzung aller zur Verfügung stehenden Ressourcen zu erwarten ist. Der vorgestellte Ansatz zur Kraftauswertung eignet sich deshalb auch besonders gut für den Einsatz in Implementierungen des Barnes-Hut-Verfahrens auf verteilten Systemen in Peer-to-Peer-Grids, deren Rechnersysteme sich durch ihren hohen Grad an Heterogenität auszeichnen. Eine solche Implementierung lässt sich beispielsweise mit Hilfe der Desktop-Grid Plattform COHESION [SBHD08] realisieren.

Literaturverzeichnis

[Bar90] Barnes, Joshua E.: A Modified Tree Code: Don't Laugh; It Runs. In: *Journal of Computational Physics* 87 (1990), Nr. 1, S. 161–170. – ISSN 0021–9991 (Zitiert auf den Seiten 49 und 50)

[BGZ12] Bédorf, Jeroen ; Gaburov, Evghenii ; Zwart, Simon P.: A Sparse Octree Gravitational N-Body Code That Runs Entirely on the GPU Processor. In: *Journal of Computational Physics* 231 (2012), April, Nr. 7, S. 2825–2839. – ISSN 0021–9991 (Zitiert auf den Seiten 50, 66 und 69)

[BH86] Barnes, Josh ; Hut, Piet: A Hierarchical O(N log N) Force-Calculation Algorithm. In: *Nature* 324 (1986), Dezember, Nr. 6096, S. 446–449. – ISSN 0028–0836 (Zitiert auf Seite 49)

[Bil] Billings, Jay J.: *http://code.google.com/p/starscream/* (abgerufen am 1.7.2012) (Zitiert auf Seite 88)

[BKWb98] Blochinger, Wolfgang ; Küchlin, Wolfgang ; Weber, Andreas: The Distributed Object-Oriented Threads System DOTS. In: Ferreira, A. (Hrsg.) ; Rolim, J. (Hrsg.) ; Simon, H. (Hrsg.) ; Teng, S.-H. (Hrsg.): *Fifth Intl. Symp. on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*. Berkeley, CA, USA : Springer-Verlag, August 1998 (LNCS 1457), S. 206–217 (Zitiert auf den Seiten 38 und 49)

[BP11] Burtscher, Martin ; Pingali, Keshav: An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. In: Hwu, Wen-mei W. (Hrsg.): *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., 2011, Kapitel 6, S. 75–92 (Zitiert auf den Seiten 50 und 66)

[BZBP09] Bungartz, Hans-Joachim ; Zimmer, Stefan ; Buchholz, Martin ; Pflüger, Dirk: *Modellbildung und Simulation: Eine anwendungsorientierte Einführung*. Berlin : Springer-Verlag, 2009. – ISBN 978–3540798095 (Zitiert auf den Seiten 16 und 20)

[GBZ10] Gaburov, Evghenii ; Bédorf, Jeroen ; Zwart, Simon P.: Gravitational Tree-Code on Graphics Processing Units: Implementation in CUDA. In: *Procedia CS* 1 (2010), Nr. 1, S. 1119–1127 (Zitiert auf den Seiten 50 und 94)

[GKZC03] Griebel, Michael ; Knapek, Stephan ; Zumbusch, Gerhard ; Caglar, Attila: *Numerische Simulation in der Moleküldynamik: Numerik, Algorithmen, Parallelisierung, Anwendungen*. Berlin, Heidelberg : Springer-Verlag, 2003. – ISBN 978–3540418566 (Zitiert auf den Seiten 7, 18, 20, 22, 23, 30, 31 und 32)

[GR87] Greengard, Leslie ; Rokhlin, Vladimir: A Fast Algorithm for Particle Simulations. In: *Journal of Computational Physics* 73 (1987), Dezember, S. 325–348. – ISSN 0021–9991 (Zitiert auf Seite 32)

[JD11] Jurkiewicz, T. ; Danilewski, P.: Efficient Quicksort and 2D Convex Hull for CUDA, and MSIMD as a Realistic Model of Massively Parallel Computations. (2011) (Zitiert auf Seite 34)

[KH10] Kirk, David B. ; Hwu, Wen-mei W.: *Programming Massively Parallel Processors: A Hands-on Approach*. 1st. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2010. – ISBN 978–0123814722 (Zitiert auf den Seiten 7, 42, 43, 44, 45 und 46)

[LHN05] Lefebvre, Sylvain ; Hornus, Samuel ; Neyret, Fabrice: Octree Textures on the GPU. In: Pharr, Matt (Hrsg.): *GPU Gems 2*. Amsterdam : Addison-Wesley Longman, 2005. – ISBN 978–0321335593, Kapitel 37, S. 595–613 (Zitiert auf Seite 50)

[Nak12] Nakasato, Naohito: Implementation of a Parallel Tree Method on a GPU. In: *Journal of Computational Science* 3 (2012), Nr. 3, S. 132 – 141. – ISSN 1877–7503 (Zitiert auf Seite 50)

[NHP07] Nyland, Lars ; Harris, Mark ; Prins, Jan: Fast N-Body Simulation with CUDA. In: Nguyen, Hubert (Hrsg.): *GPU Gems 3*. Addison-Wesley Professional, August 2007, Kapitel 31 (Zitiert auf Seite 49)

[NVI09] NVIDIA: *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. 2009 (Zitiert auf den Seiten 7 und 43)

[NVI12] NVIDIA: *NVIDIA CUDA Programming Guide 4.1*. 2012 (Zitiert auf Seite 43)

[RR07] Rauber, Thomas ; Rünger, Gudula: *Parallele Programmierung*. 2., neu bearb. u. erw. Aufl. Berlin, Heidelberg : Springer-Verlag, 2007. – ISBN 978–3540731139 (Zitiert auf den Seiten 34, 36, 38, 39, 41 und 47)

[SBHD08] Schulz, Sven ; Blochinger, Wolfgang ; Held, Markus ; Dangelmayr, Clemens: COHESION - A Microkernel Based Desktop Grid Platform for Irregular Task-Parallel Applications. In: *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Applications* 24 (2008), Mai, Nr. 5, S. 354–370. – ISSN 0167–739X (Zitiert auf Seite 94)

[Sch06] Schneider, Peter: *Einführung in die Extragalaktische Astronomie*. Berlin, Heidelberg : Springer-Verlag, 2006. – ISBN 978–3540258322 (Zitiert auf Seite 18)

[Spr05] Springel, Volker: The Cosmological Simulation Code GADGET-2. In: *Monthly Notices of the Royal Astronomical Society* 364 (2005) (Zitiert auf den Seiten 49 und 88)

[Tan09] Tanenbaum, Andrew S.: *Modern Operating Systems*. 3. Aufl. Upper Saddle River, NJ 07458 : Pearson Education, 2009. – ISBN 978–0136006633 (Zitiert auf Seite 39)

[TB07] Thomaszewski, Bernhard ; Blochinger, Wolfgang: Physically Based Simulation of Cloth on Distributed Memory Architectures. In: *Parallel Computing* 33 (2007), Nr. 6, S. 377–390 (Zitiert auf Seite 49)

[WEA10] Watkins, Laura L. ; Evans, N. W. ; An, Jin H.: The Masses of the Milky Way and Andromeda Galaxies. In: *Monthly Notices of The Royal Astronomical Society* 406 (2010), Juli, S. 264–278. – ISSN 0035–8711 (Zitiert auf Seite 19)

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Hendrik Hochstetter)