Bachelorarbeit Nr. 2488

Lastquantifizierung in Molekülsimulationen mit langreichweitigen Kräften

Vincent Wagner

Prüfer: Prof. Dr. Dirk Pflüger

Zweitprüferin: Prof. Dr. Miriam Mehl

Betreuer: Steffen Hirschmann

Beginn am: 23.02.2016

Beendet am: 08.07.2016

CR-Nummer: D.2.8

12. Juli 2016

Kurzfassung

Durch die große Relevanz von Molekülsimulationen in der modernen Forschung ist es wichtig, auf diesem Gebiet Algorithmen nutzen zu können, die zum einen möglichst schnell und fehlerfrei funktionieren, und zum anderen sollten sie in der Lage sein, auf modernen Hochleistungscomputern deren volle Kapazität zu nutzen. Dies erfordert insbesondere effiziente Parallelisierung. Diese Arbeit gibt einen Überblick über die bereits bekannten, schnellen Algorithmen zur Simulation von Molekülen. Anschließend werden im Zuge der Parallelisierung dieser Algorithmen erst Metriken definiert, mit denen man Rechenlast quantifizierbar macht und im zweiten Schritt Partitionierungsmethoden vorgestellt, mit denen die Moleküle auf die zur Verfügung stehenden Prozesse verteilt werden. Messungen zu vielen dieser Verfahren werden präsentiert, um Aussagen über die Performance der verschiedenen Verfahren zu untermauern.

Inhaltsverzeichnis

1 Einführung und Motivation (Propädeutikum) 5

1 Aufbau der wissenschaftlichen Arbeit 6
2 Ähnliche wissenschaftliche Arbeiten 6
3 Anwendungen 7
4 Exakte Berechnung der Kräfte 7
5 Lang- und Kurzreichweitige Potenziale 8
5.a Kurzreichweitige Potenziale 8
5.b Anmerkungen 11

2 Methoden zur Simulation von langreichweitigen Potenzialen (Propädeutikum) 12

1 Baumbasierte Verfahren 12
1.a Der Algorithmus von Barnes und Hut 15
1.b Das Verfahren von Appel 22
1.c Das schnelle Multipolverfahren 24

3 Parallelisierung 27

1 Metriken zur Quantifizierung von Rechenlast 28
1.a Last-Metriken für Barnes und Hut 28
1.b Last-Metriken für das schnelle Multipolverfahren . . . 33
2 Speicher-Architekuren 34
2.a Shared-Memory-Architekturen 35
2.b Distributed-Memory-Architekturen 35
2.c Distributed-Shared-Memory-Architekturen 36
3 Partikelpartitionierung 36
3.a Statisches Partitionieren 37
3.b Dynamisches Partitionieren 42
4 Barnes-Hut auf Shared Memory Architekturen 44
4.a Baumaufbau 44
4.b Berechnung der Werte der Pseudopartikel 45
4.c Kraftberechnung 46
4.d Positionsupdate der Partikel 46
5 Barnes-Hut auf Distributed Memory Architekturen 46

5.a Initialisierung 46
5.b Kraftberechnung 47
5.c Update der Partikelpositionen 50
5.d Inkrementelles Baumupdate 50
6 Das schnelle Multipolverfahren auf Shared-Memory-Architekturen 50
6.a Aufwärts-Schritt 50
6.b Abwärts-Schritt 51
7 Das schnelle Multipolverfahren auf Distributed-Memory-Architekturen 51
7.a Baumaufbau 51
7.b Kraftberechnung 52

4 Messergebnisse **54**
1 Diskussion der Ergebnisse 54

5 Zusammenfassung und Ausblick **58**

Kapitel 1

Einführung und Motivation (Propädeutikum)

In vielen physikalischen Problemstellungen spielen Vielkörperprobleme eine entscheidende Rolle. Allgemein formuliert geht es dabei um die Berechnung des Ortes, der Geschwindigkeit und eventuell der Beschleunigung vieler Körper, die sich gegenseitig durch ein Potenzial beeinflussen. Dieses Potenzial kann verschiedener Natur sein. Oft spielen auch mehrere Potenziale eine Rolle. Betrachtet man zum Beispiel geladene Teilchen, so wirkt neben dem Gravitationspotenzial, was von jeder Masse ausgeht auch ein elektrisches Potenzial, welches durch die Ladungen der Teilchen erzeugt wird.
Für den Spezialfall von zwei Körpern lässt sich dieses Problem als System zweier gekoppelter Differenzialgleichungen lösen. Für drei oder mehr Körper ist eine analytische Berechnung jedoch nicht mehr möglich, weshalb man sich intensiv mit numerischen Methoden zur Lösung beschäftigt.
Auf die analytische Lösung des sogenannten Zweikörperproblems wird in dieser Arbeit nicht eingegangen. Informationen hierzu sind zum Beispiel in [1] zu finden. Der Fokus dieser Arbeit wird viel mehr auf der Analyse numerischer Methoden liegen. Dabei wird insbesondere die Adaption auf moderne Hochleistungsrechner und die damit einhergehende Parallelisierung beleuchtet. Hierbei steht stets die Effizienz der Algorithmen im Vordergrund, was für parallele Verfahren eng mit der sogenannten Lastbalance zusammenhängt. Last bezeichnet hierbei die Arbeit, die alle Prozesse für ein gegebenes Problem tätigen müssen. Als lastbalanciert bezeichnet man eine möglichst gleichmäßige Aufteilung der Gesamtlast auf die zur Verfügung stehenden Prozesse. Eine detaillierte Beschreibung und Diskussion dieses Begriffs findet sich in Kapitel 3.

1 Aufbau der wissenschaftlichen Arbeit

Im ersten Kapitel wird die direkte Methode zur Lösung des Vielkörperproblems vorgestellt. Da diese Methode sich nur eingeschränkt skalieren lässt, werden Potenziale im speziellen betrachtet, was erste Erkenntnisse für eine besser skalierbare Methode liefert, die allerdings nur für sogenannte kurzreichweitige Potenziale viele Rechenschritte einsparen kann.
Methoden zur Simulation langreichweitiger Potenziale werden im zweiten Kapitel beleuchtet. Dabei werden Baumstrukturen ausgenutzt, um geschickt Näherungen nutzbar zu machen. Die beiden verbreitetsten Vertreter dieser Klasse von Algorithmen sind der Barnes-Hut-Algorithmus und die schnelle Multipole-Methode. Das dritte Kapitel stellt gewissermaßen das Herzstück dieser Arbeit dar. Es behandelt die Parallelisierungsansätze, die nötig sind, um die aus Kapitel zwei bekannten Algorithmen auf modernen Hochleistungsrechnern zu implementieren. Nachdem die parallelen Algorithmen erarbeitet sind, werden in Kapitel 4 Messergebnisse zu verschiedenen, in Kapitel 3 dargelegten Methoden präsentiert. Kapitel 5 fasst die Arbeit zusammen und gibt einen Ausblick für weitere Forschung.

2 Ähnliche wissenschaftliche Arbeiten

Die meisten hier genannten Arbeiten bauen auf der Basis der Algorithmen von Barnes und Hut [2], Appel [3] und Greengard und Rokhlin [4] auf. Winkel et al. zeigen in [5] eine Version des Barnes-Hut-Algorithmus, der auf der Rechnerarchitektur parallelisiert ist, die in Kapitel 3 als *Distributed-Memory* vorgestellt wird und insbesondere spezielle MPI-Routinen nutzt, um optimale Performance zu erhalten. Auch in der hier genutzten Implementierung wird MPI verwendet. Board et al. [6], Liu [7], Grama et al. [8] [9] stellen in ihren Arbeiten weitere, verschiedene parallele Versionen des Algorithmus von Barnes und Hut vor, die unterschiedliche Partitionierungsmethoden und Lastmodelle nutzen, um gute Skalierbarkeit zu erreichen. Die erste parallele Version des Algorithmus von Greengard und Rokhlin wurde von Greengard selbst entwikelt [10]. Viele nachfolgende Arbeiten beschäftigen sich mit der Adaption dieser Methode auf inhomogene Anwendungsbeispiele. Beispiele hierfür sind Singh et al. [11] [12] und Board et al. [6]. Ltaief und Yakota beschäftigen sich in [13] mit der Frage, welche Arten von Kommunikationsroutinen auf modernen Hochleistungsrechner die beste Performance zeigen und gleichzeitig gute Aussichten auf noch höhere Skalierbarkeit besitzen. Teng [14], Warren und Salmon [15] und Lam et al. [16] zeigen Partitionierungsmethoden, die für viele Algorithmen anwendbar sind und speziell auch für die oben beschriebenen Algorithmen verwendet werden.
Die drei oben genannten Algorithmen sind natürlich nicht die einzigen, die für Molekülsimulationen genutzt werden. Anderson [17] nutzt zum Beispiel

Gittermethoden und die Poissongleichung, um dieses Problem zu berechnen. Um einen Überblick über die verschiedenen Methoden zu erlangen, ist [18] eine gute Wahl.

3 Anwendungen

Ursprünglich wurde der erste Algorithmus, der unter den baumbasierten Verfahren vorgestellt wird, dazu entwickelt, Galaxien zu simulieren. Dabei sind Sterne die Objekte, deren Positionen und Geschwindigkeiten ermittelt werden sollen. Die Wechselwirkungen der Sterne sind gravitativer Natur. Weitere Anwendungen sind molekulare Thermodynamik, Akustik oder Plasmaphysik. Um in diesen Fällen von der molekularen Skala auf Zusammenhänge in den Dimensionen des menschlichen Lebens schließen zu können, müssen ungeheuer viele Moleküle simuliert werden. Gerade deshalb sind effiziente Algorithmen wichtig. Für die Anwendung in der Astrophysik wurde sogar eine eigene Rechnerarchitektur entwickelt, die in [19] präsentiert wird.

4 Exakte Berechnung der Kräfte

Das erste Kapitel behandelt die direkte Methode zur Evaluierung der Potenziale. Potenziale äußern sich in Kraftfeldern, sobald ein Probeobjekt q vorhanden ist. In den obigen Anwendungen sind das zum Beispiel das gravitative oder das elektrische Feld. Dabei ist das Potenzial eines Objektes Q die Energie, die eine Probeobjekt q erfährt oder aufbringen muss, um aus der Unendlichkeit zu Q zu gelangen, normiert mit die Ladung von q. Ladung muss in diesem Fall nicht elektrisch sein, sie kann zum Beispiel auch die Masse eines Objektes sein, je nachdem welches Feld betrachtete wird.
In den hier betrachteten Beispielen erzeugt jedes Objekt ein eigenes Potenzial. Es werden ausschließlich Interaktionen zwischen jeweils zwei Objekten betrachtet, also sogenannte Paarpotenziale. Anschaulich bedeutet das, dass zum Beispiel jeder Planet, der simuliert wird, alle anderen Planeten anzieht. Die für uns wichtigste Eigenschaft eines Potenzials ist das Prinzip der Superposition für konservative Kraftfelder. Es besagt, dass sich das Gesamtpotenzial aus der Summe aller Einzelpotenziale der Objekte zusammensetzt. Daraus resultiert, dass sich die Kraft auf ein Objekt Q_0 auch aus der Summe aller Einzelkräfte zwischen Q_0 und einem anderen Objekt zusammensetzt. Man kann also alle Objekt-Objekt-Interaktionen berechnen und erhält so alle wirkenden Kräfte im zu simulierenden System.
Dies ist die grundlegende Idee für die direkte Berechnung der Kräfte. Je nachdem in welchem Kraftfeld man sich befindet, variiert zwar die Formel der Kraft, das grundsätzliche Vorgehen jedoch ist stets das gleiche.
Betrachtet man die Komplexität dieser Herangehensweise, so ist schnell ersichtlich, dass man bei der Simulation von n Objekten für jedes die-

ser Objekte mit $n-1$ (allen außer sich selbst) anderen Objekten Interaktionen berechnen muss. Insgesamt erhält man also eine Komplexität von $\mathcal{O}(n(n-1)) = \mathcal{O}(n^2)$. Durch Ausnutzung des dritten Newtonschen Gesetztes kann man die Hälfte der Operationen einsparen, was aber an der Komplexität nichts ändert.
Wie in Abschnitt 1.2 festgestellt, ist es in vielen Anwendungen wünschenswert, möglichst viele Objekte zu simulieren. Daher ist eine in der Zeit quadratische Komplexität nicht performant genug, um auf solchen Skalen zu rechnen.
Will man die Komplexität verbessern, liegt es also nahe, Interaktionen einzusparen. Potenziale (und damit auch die daraus resultierenden Kräfte) verschwinden zwar prinzipiell für endliche Distanzen zur Quelle nie, jedoch konvergieren sie gegen 0: Ein erster Ansatz ist daher, Interaktionen mit zu weit entfernten Objekten nicht zu berechnen, da die hier wirkende Kraft sehr klein ist. Man schneidet das Potenzial quasi ab (vgl. Abbildung 1.1 und 1.2). *Zu weit* ist natürlich ein schwammiger Begriff und variiert daher mit der speziellen Implementierung. In welchen Szenarien das Abschneiden des Potenzials in Ordnung ist, wird im folgenden Kapitel näher betrachtet.

5 Lang- und Kurzreichweitige Potenziale

Es stellt sich hierbei die Frage, wie groß der resultierende Fehler maximal sein kann, wenn man das Potenzial ab dem Radius r_0 abschneidet. Sei das Potenzial zweidimensional und mit $P(r, \eta)$ bezeichnet, so gilt für den Fehler $E(r_0)$:

$$E(r_0) \;=\; \int_0^{2\pi} \int_{r_0}^{\infty} P(r,\eta)\; dr d\eta \tag{1.1}$$

Hierbei ist r der Abstand zur Quelle und η der Winkel. Es wird in Polarkoordinaten integriert. Man kann für zweidimensionale Probleme zeigen, dass dieses Integral für Potenziale, die mit mindestens dritter Ordnung im Radius abfallen, konvergiert (im dreidimensionalen muss die Ordnung mindestens vier sein) [20]. Für alle anderen Potenziale divergiert es, was im Umkehrschluss bedeutet, dass der Fehler divergiert.
Mithilfe dieser Einsicht kann man Potenziale nun einteilen in kurzreichweitige und langreichweitige Potenziale. Für ein kurzreichweitiges Potenzial muss das Fehlerintegral konvergieren. Divergiert es hingegen, so liegt ein langreichweitiges Potenzial vor.

5.a Kurzreichweitige Potenziale

Im Fall von kurzreichweitigen Potenzialen ist es nach der Betrachtung des entstehenden Fehlers akzeptabel, das Potenzial ab dem Radius r_0 abzuschneiden. Auf das tatsächliche Problem bezogen heißt das, dass man nur

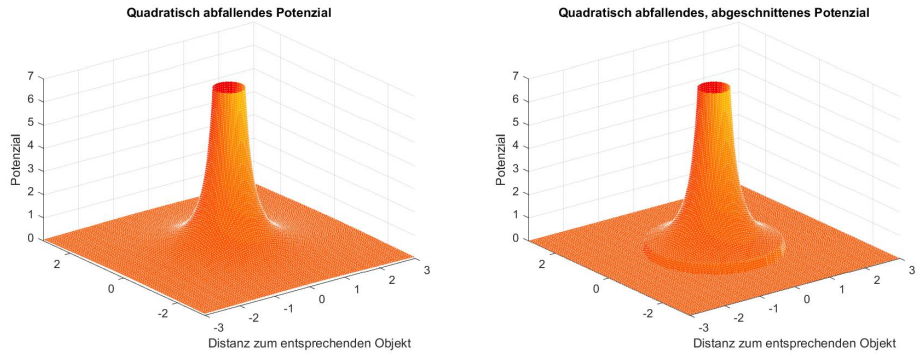

Abbildung 1.1: Zwei Darstellungen eines zweidimensionalen, quadratisch abfallenden Potenzials. Man erkennt im linken Bild, dass das Potenzial für große Distanzen schnell abnimmt. Rechts ist das Potenzial abgeschnitten zu sehen.

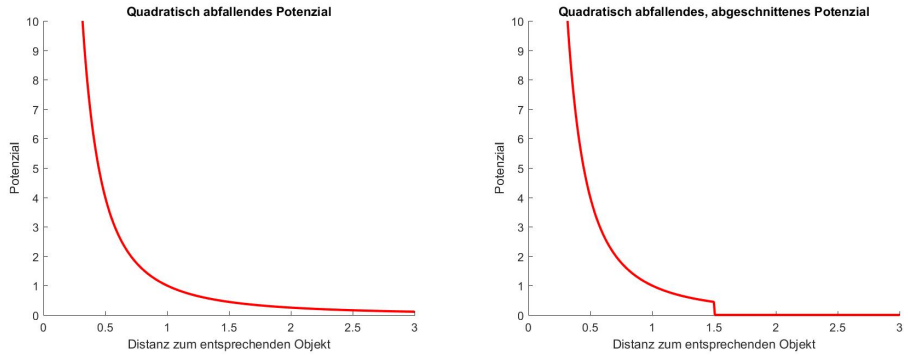

Abbildung 1.2: Wieder eine Abbildung von Potenzialfunktionen, dieses mal im Eindimensionalen. Im rechten Bild ist zu sehen, dass das Potenzial ab $r_0 = 1,5$ abgeschnitten wird.

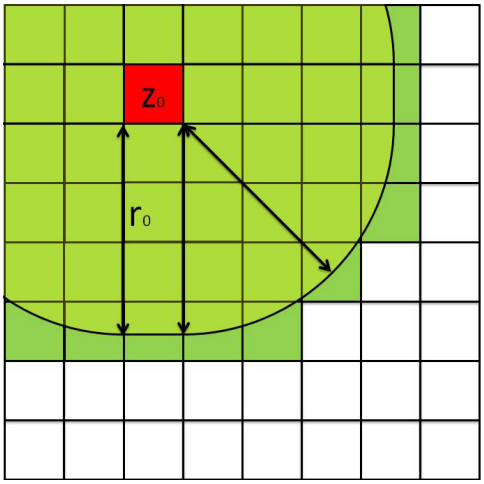

Abbildung 1.3: Die physische Domäne mit Gitterzellen. Die Zelle z_0 ist rot eingefärbt. Alle z_i, mit deren Partikeln ein Partikel aus z_0 potenziell interagiert, sind grün dargestellt. Die Vereinigung aller Kreise mit Zentrum in z_0 und Radius r_0 ist in schwarz eingezeichnet.

noch Interaktionen zwischen Objekten berechnet, deren euklidischer Abstand kleiner als r_0 ist. Eine effiziente Implementierung nennt sich *Linked-Cells* [20], [21]. Man legt sich dabei einen Abschneideradius r_0 fest und zieht ein Gitter mit Kantenlänge r_0 über das zu simulierende Gebiet mit den Objekten (das Gebiet wird im Folgenden physische Domäne genannt). Für jede Zelle z_0 des Gitters findet man nun alle Zellen z_i, die noch mindestens zu einem kleinen Teil innerhalb der Vereinigung aller Kreise mit Zentrum in z_0 und Radius r_0 liegen (vgl. Abbildung 1.3). Nun müssen pro Zelle für die Kraftberechnung nicht mehr alle anderen Zellen betrachtet werden, sondern nur noch eine konstante Anzahl, die natürlich von r_0, nicht mehr jedoch von der Anzahl der Partikel n abhängt. Die Zeitkomplexität ist nur noch linear in n.
Mit der Wahl von r_0 hat man dabei eine gute Stellschraube für die Güte des Ergebnisses: Wählt man ein sehr kleines r_0, so sinkt die Berechnungszeit drastisch, der Fehler hingegen wird natürlich deutlich größer, da deutlich mehr Anteile des Potenzials nicht mehr berücksichtigt werden. Wählt man r_0 hingegen in der Größenordnung der physischen Domäne, so berechnet man fast alle Interaktionen, was einen geringen Fehler, aber großen Rechenaufwand mit sich bringt. Für detaillierte Informationen zu diesem Verfahren, siehe [20].

5.b Anmerkungen

Bei allen hier vorgestellten Methoden und Algorithmen sollte man stets im Hinterkopf behalten, dass die Verbesserung der Laufzeit aus einer Verschlechterung der Genauigkeit resultiert. Es handelt sich nicht um eine schlicht bessere Implementierung der direkten Methode. Es gibt Korrekturterme, die eingeführt werden können, um den entstehenden Fehler zu verkleinern [21].
Es wird sich zeigen, dass es bei allen hier vorgestellten Methoden eine Stellschraube gibt, die es uns erlaubt aus jeder der Methoden eine exakte Methode zu konstruieren. Wie auch in der gerade präsentierten Linked-Cell-Variante bedeutet das bei fast allen Methoden, alle Interaktionen direkt zu berechnen.

Kapitel 2

Methoden zur Simulation von langreichweitigen Potenzialen (Propädeutikum)

Nachdem nun für kurzreichweitige Potenziale eine Lösung angerissen wurde, geht es im Rest der folgenden Arbeit um die Behandlung von langreichweitigen Potenzialen. Die Herausforderung hierbei ist, dass die Potenziale nicht einfach gestutzt werden können, da dies zu große Fehler mit sich bringen würde, wie im letzten Kapitel gezeigt wurde. Auf der anderen Seite ist die direkte Berechnung nicht performant genug, um im großen Maßstab umgesetzt zu werden.
Die hier vorgestellten Verfahren beruhen auf der Nutzung von Baumstrukturen, die im Folgenden eingeführt werden.

1 Baumbasierte Verfahren

Wenn in dieser Arbeit von einem Baum die Rede ist, so bezeichnet dies einen gerichteten, azyklischen Graphen (englisch häufig DAG), bei dem jeder Knoten genau einen Vater und eine feste Anzahl Söhne besitzt. Knoten mit Kindern werden innere Knoten genannt und kinderlose Knoten Blätter. Der Knoten, der keinen Vater hat und somit ganz oben im Baum steht, ist die Wurzel.

Die grundlegende Idee baumbasierter Verfahren besteht darin, Partikel mit ähnlichen Wechselwirkungen so zusammenzufassen, dass anschließend nur noch deren gemeinsame Wirkung betrachtet werden muss. In Abbildung 2.1 stellen die Punkte x_1 bis x_3 Partikel dar, die gravitativ miteinander interagieren. Hier ist die Kraft zwischen x_1 und x_2 der Kraft zwischen x_1 und

Abbildung 2.1: Partikel x_1 bis x_3.

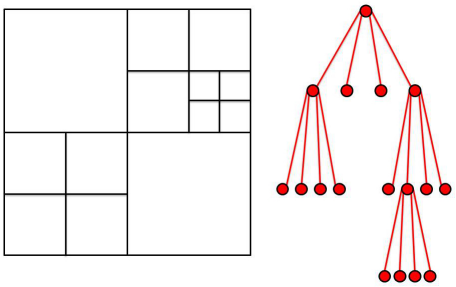

Abbildung 2.2: Ein Beispiel eines Quartärbaums, bei dem links die zweidimensionale Domäne und rechts die resultierende Baumstruktur zu sehen ist.

x_3 sehr ähnlich. Es liegt nahe, also nur eine Kraft von Partikel x_1 zu einem Pseudopartikel zu berechnen, wobei der Ort des Pseudopartikels der Schwerpunkt der beiden Partikel x_2 und x_3 und die Masse die Summe der beiden Einzelmassen von x_2 und x_3 ist.

Eine mögliche Art, diese Idee auch in größeren Maßstäben umzusetzen, beruht auf Baumstrukturen. Betrachtet wird eine quadratische oder kubische physische Domäne, in der n Partikel x_1 bis x_n beliebig verteilt sind. Die gesamte Domäne entspricht der Wurzel des Baums. Als Verfeinern des Baumes/der Zelle bezeichnet man den Vorgang der Aufteilung in vier oder acht gleich große Kinderknoten/Subdomänen. Das sind dann also entweder vier Quadrate oder acht Würfel, die eine Kantenlänge halb so lang wie die Kantenlänge der Wurzel besitzen (vgl. Abbildung 2.2). Im Folgenden werden diese Bäume Quartär- oder Oktalbäume genannt, je nach der Anzahl der Dimensionen.

Der Teil der physischen Domäne, der einem Baumknoten zugeordnet ist, bezeichnet man als (dessen) Zelle. Im Folgenden wird keine explizite Unterscheidung mehr zwischen den beiden gemacht. Für jeden Knoten eines Baumes kann man nun ein Pseudopartikel berechnen, das im Schwerpunkt

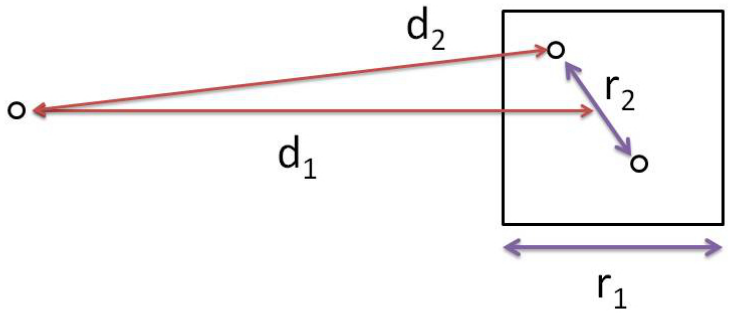

Abbildung 2.3: Verschiedene Möglichkeiten, d und r des θ-Kriteriums zu berechnen.

aller Partikel seiner Zelle liegt und deren Gesamtmasse besitzt.

Eine Möglichkeit zu entscheiden, ob man für eine Interaktion zwischen einem Partikel und einer Zelle die Näherung durch Zusammenfassen mehrerer Partikel nutzt oder nicht, besteht darin, das Verhältnis zwischen der physischen Ausdehnung der Partikel innerhalb der Zelle und der Distanz der beiden Objekte voneinander zu berechnen. Ist dieses Verhältnis kleiner als ein Parameter θ, so nutzt man die Näherung, ansonsten nicht. Sei also d die Distanz und r die Ausdehnung, so muss gelten:

$$\frac{r}{d} \leq \theta \tag{2.1}$$

Die genaue Implementierung ist natürlich variabel. So kann zum Beispiel für d die Distanz zwischen dem Partikel und dem zusammengefassten Pseudopartikel der Zelle genutzt werden oder aber auch das Minimum der Distanzen zwischen dem einzelnen, weit entfernten Partikel und allen Partikeln der Zelle. Für r hingegen nutzt man oft die Kantenlänge der Zelle, obwohl der maximale Abstand von Partikeln innerhalb der Zelle mehr Näherungen erlauben würde, da Ersteres eine Abschätzung von oben ist. In der später präsentierten Implementierung wird die Distanz zwischen Partikel und Pseudopartikel als d genutzt und r ist ihre Kantenlänge (vgl. Abbildung 2.3). Im Folgenden wird das Kriterium, das zur Entscheidung führt, ob Partikel in ihrer Betrachtung zusammengefasst werden sollen, als θ-Kriterium bezeichnet. Nach [18] ist $0,5$ ein guter Wert für θ.

Betrachtet man

$$\lim_{\theta \to 0} \frac{r}{d} \leq \theta \,, \tag{2.2}$$

so wird die Näherung einer Interaktion nur dann genutzt, wenn d bei konstanter Zellengröße beliebig groß wird. Da dies bei endlichen Domänen nie der Fall ist, werden am unteren Grenzwert von θ also alle Interaktionen direkt berechnet.

Nachdem nun dieses Kriterium zur Näherung von Wechselwirkungen formuliert wurde, wird nun der erste kompletten Algorithmus präsentiert, der von J. Barnes und P. Hut in [2] erstmals Erwähnung findet.

1.a Der Algorithmus von Barnes und Hut

Den Algorithmus von Barnes und Hut kann man grundsätzlich in vier Teile gliedern:

- Baumaufbau
- Berechnung der Werte der Pseudopartikel
- Kraftberechnung
- Positionsupdate der Partikel

Baumaufbau

Startend mit der Wurzel, die als zugehörige Zelle die gesamte Domäne besitzt, fügt man nacheinander alle Partikel in die Wurzel ein. Es kann zu drei verschiedenen Szenarien kommen:

- Bei der Zelle handelt es sich um eine bisher leere Zelle:
  In dies der Fall, so wird das Partikel der Zelle zugeordnet und mit dem nächsten Partikel fortgefahren.
- Die Zelle hat bereits genau ein Partikel gespeichert:
  Die Zelle wird nun verfeinert und für beide Partikel wird ermittelt, in welches Kind dieser Zelle sie aufgrund ihrer Position gehören. Nun werden die Partikel in die ermittelten Kinder eingefügt. Sollten beide Partikel dem gleichen Kind zuzuordnen sein, so muss das verfeinern rekursiv angewendet werden, bis die beiden Partikel in unterschiedlichen Zellen sind.
- In die Zelle wurden schon mindestens zwei Partikel eingefügt, sie ist also schon verfeinert worden:

Hier muss nur das Kind der Zelle ermittelt werden, dem das Partikel zuzuordnen ist und anschließend wird das Partikel in dieses Kind eingefügt. (Auch hier kann es dabei zu einem der drei Fälle kommen.)

Es folgt Pseudo-Code zum Baumaufbau (Der gesamte Pseudo-Code ist stark an die in dieser Arbeit implementierte Version angelehnt):
Dabei ist *numPart* das Attribut eines Baumes, welches die eingefügten Partikel zählt und *connPart* das Partikel des Baumes, wenn der Baum ein Blatt ist. Die Funktion *corrChild* (**Particle** i, **Tree** t) finde für den Baum t und die Position des Partikels i das Kind von t, in das i eingefügt werden muss. Die Funktion $refine$ (**Tree** t) erweitert einen Baum um seine Kinder. Die Funktion *insertParticle* muss für alle Partikel mit der Wurzel aufgerufen werden.

```
function insertParticle (Particle i, Tree t)
    t.numPart ++;
    if t.numPart == 1 then
        t.connPart = i;
    else if t.numPart == 2 then
        refine (t);
        insertParticle (i, corrChild (i, t));
        insertParticle (t.connPart, corrChild (t.connPart, t));
        t.connPart == NULL;
    else
        insertParticle (i, corrChild (i, t));
    end if
end function
```

Bei homogener Verteilung der Partikel innerhalb der Domäne müssen insgesamt n Partikel in einen Baum der Tiefe $\log(n)$ (dies resultiert aus der Homogenität) einsortiert werden. Das Einsortieren aller Partikel entspricht also einem Aufwand von $\mathcal{O}(n \log n)$. Die Basis des Logarithmus ist vier im Zweidimensionalen und acht in drei Dimensionen.

Berechnung der Werte der Pseudopartikel

Nachdem alle Partikel in den Baum einsortiert wurden und jedes Blatt maximal ein Partikel enthält, müssen für alle inneren Knoten des Baumes (also die, die kein Partikel enthalten) die zugehörigen Pseudopartikel berechnet werden. Dies geschieht in einem Baumdurchlauf angefangen bei den Blättern. Jedes Blatt gibt die Koordinaten und die Masse seines Partikels an seinen Vaterknoten weiter, der dann die Werte seines Pseudopartikels berechnet. Dies geschieht rekursiv solange, bis auch die Wurzel die Koordinaten und die Masse ihres Pseudopartikels kennt. Danach endet dieser

Schritt.

Da die Tatsächliche Berechnung der Werte eines Pseudopartikels eines inneren Knotens nicht von n (also der Gesamtanzahl der Partikel im Baum) abhängt, kann für diese Berechnung ein konstanter Aufwand c_{pp} angenommen werden. Bei homogener Verteilung der Partikel innerhalb der Domäne ist die Anzahl n_{pp} der inneren Knoten (und somit auch Pseudopartikel):

$$n_{pp2D} = \sum_{i=1}^{\log n} \left(\frac{n}{4^i}\right) \leq \sum_{i=1}^{\infty} \left(\frac{n}{4^i}\right) = \frac{n}{\frac{3}{4}} - n = \frac{n}{3} \tag{2.3}$$

Beziehungsweise:

$$n_{pp3D} = \sum_{i=1}^{\log n} \left(\frac{n}{8^i}\right) \leq \sum_{i=1}^{\infty} \left(\frac{n}{8^i}\right) = \frac{n}{\frac{7}{8}} - n = \frac{n}{7} \tag{2.4}$$

In beiden Fällen ist die Anzahl insbesondere linear in n. Somit ist die Komplexität dieses Schrittes also $\mathcal{O}(n)$.
Eine andere Möglichkeit, die Werte der Pseudopartikel zu berechnen besteht darin, sie direkt beim Baumaufbau mitzuberechnen, während die Partikel rekursiv *durch den Baum gereicht werden.*

Kraftberechnung

Die eigentliche Kraftberechnung erfolgt partikelweise. Für jedes Partikel wird – bei der Wurzel beginnend – überprüft, ob das θ-Kriterium erfüllt ist, also ob die entsprechende Zelle des Baumes als Interaktionspartner gewählt werden kann, ohne zu große Fehler zu erzeugen.
Ist das θ-Kriterium erfüllt (was für *sinnvolle* Werte von θ bei der Wurzel nie der Fall ist), so kann die Kraft zwischen dem Partikel und der Zelle berechnet werden und alle Kinder der Zelle werden zur weiteren Kraftberechnung nicht mehr benötigt. Ist das Kriterium hingegen nicht erfüllt, so wird obige Prozedur für alle Kinder der Zelle rekursiv durchgeführt. Ist man in Zellen angelangt, die nur noch ein Partikel enthalten, ohne dass man diese spezielle Wechselwirkung bisher nähern konnte, so berechnet man die Interaktion der beiden zugehörigen Partikel direkt. Alle bisher berechneten Einzelkräfte ergeben nun in Summe die Gesamtkraft, die auf das jeweilige Partikel wirkt.
Es folgt der Pseudo-Code für die Phase der Kraftberechnung:
Die Funktion *thetaCheck* (**Particle** i, **Tree** t) überprüft, ob man die Teildomäne von t nutzten darf, um die auf i wirkenden Kräfte zu berechnen, oder ob man diese weiter aufteilen muss. Ist eine Näherung in Ordnung, gibt sie 1 zurück, anderenfalls 0. Um die eigentlich Kraft zu berechnen und auf einen gegebenen Kraftvector aufzuaddieren, wird *addForce* (**Particle** i, **Tree** t, **Vector** $\&totalForce$) verwendet. Der unten stehende Code muss für jedes Partikel aufgerufen werden.

```
Vector totalForce = (0, 0);
function CALCFORCE (Particle i, Tree t, Vector &totalForce)
    if thetaCheck (i, t) == 0 then
        for c = 1 : 4 do
            calcForce (i, t.child_c, totalForce);
        end for
    else if thetaCheck (i, t) == 1) then
        addForce (i, t, totalForce);
    end if
end function
```

Der Rechenaufwand, der durch die Kraftberechnung anfällt, hängt natürlich stark von der Wahl von θ ab, da dieser Wert maßgeblich beeinflusst, wie viele Interaktionen tatsächlich genähert werden dürfen, beziehungsweise wie viele berechnet werden müssen. Wie oben schon erwähnt, erreicht man durch die Wahl von $\theta = 0$, dass alle Interaktionen zwischen Partikeln direkt berechnet werden. Dies entspricht n_{ppI} Interaktionen, wobei gilt:

$$n_{ppI} = \sum_{i=1}^{n} i = \frac{n \cdot (n-1)}{2} \tag{2.5}$$

Somit wäre der Aufwand dieses Schrittes $\mathcal{O}(n^2)$. Auf der anderen Seite ist es möglich, einen Wert für θ zu wählen, so dass für jedes Partikel nur die Interaktion mit der Wurzelzelle berechnet wird. Die Komplexität wird somit auf $\mathcal{O}(n)$ gedrückt, der Fehler ist natürlich relativ groß. In der Literatur wird die Komplexität für *sinnvolle* θ und homogene Partikelverteilung mit $\mathcal{O}(n \log n)$ angegeben. Grundlage dafür ist die Annahme, dass pro Baumlevel nur eine konstante Anzahl von Interaktionen berechnet werden müssen. Somit müssen für jedes der n Partikel $\log n$ Baumlevel mit jeweils konstantem Aufwand berechnet werden. Aussagen zur Komplexität treffen auch J. Barnes und P. Hut in [2].

Positionsupdate der Partikel

Nachdem alle Kräfte berechnet wurden, stellt sich die Frage, wie sich die Partikel tatsächlich bewegen. Die resultierende Bewegungsgleichung ist eine gewöhnliche Differenzialgleichung zweiter Ordnung, da die Kräfte sich nach Newtons Gesetzen der Bewegung auf die Beschleunigung auswirken. Es gilt hierbei

$$F = m \cdot \ddot{x} \,, \tag{2.6}$$

wobei F die wirkende Kraft, m die Masse des Partikels, T die betrachtete Zeitspanne und x den Ort (also $\ddot{x}$ die Beschleunigung) darstellt.

Weiterhin gilt für die Geschwindigkeit $\dot{x}$

$$\dot{x}(t) \ = \int_T \ddot{x}(t) \ dt \tag{2.7}$$

und für den Ort x

$$x(t) \ = \int_T \dot{x}(t) \ dt. \tag{2.8}$$

Das Ergebnis dieser Überlegungen sind also zwei Differenzialgleichungen erster Ordnung. Ein häufig verwendeter Ansatz zur numerischen Lösung dieser Differenzialgleichungen ist die Näherung der Ableitung durch ein Polynom. Im einfachsten Fall also eine konstante Funktion. Dieses Vorgehen wirkt auf den ersten Blick stark fehlerbehaftet, man kann den Fehler jedoch durch eine sehr kleine Wahl des Zeitschritts, über den man die Näherung betrachtet, gering halten. Die Wahl der Polynome oder eines komplett anderen Ansatzes ist natürlich offen, da alle Methoden Vor- und Nachteile mit sich bringen. In dieser Arbeit wird daher in erster Linie auf die Eigenschaften eingegangen, die für das gestellte Problem wichtig sind. Im Vorlesungsskript [22] wird genauer auf die verschiedenen Facetten der Methoden eingegangen.

Grundsätzlich lassen sich die meisten Löser für Differenzialgleichungen in explizite und implizite Verfahren einteilen (vgl. [22]). Beide haben jedoch ein und dasselbe Problem: Sie sind nicht energieerhaltend. Im Falle der expliziten Verfahren steigt die Gesamtenergie des Systems mit jedem Zeitschritt, bei impliziten Verfahren hingegen sinkt sie Schritt für Schritt. Das ist natürlich beides nicht wünschenswert. Eine Lösung dieses Problems bieten symplektische Verfahren [20]. Die grundlegende Idee beruht dabei darauf, die Geschwindigkeiten explizit und die Orte implizit zu berechnen, oder andersherum. Durch dieses Vorgehen kann man, solange die Partikel weit genug voneinander entfernt beziehungsweise die Zeitschrittweiten klein genug sind, energieerhaltend simulieren. Bekannte Verfahren sind das symplektische Euler-Verfahren oder der Leapfrog-Algorithmus (vgl. [20]). Da für jedes Partikel nur eine konstante Anzahl an Rechenoperationen vonnöten ist, ist dieser Schritt in $\mathcal{O}(n)$ berechenbar.

Inkrementelles Baumupdate

Um den Algorithmus noch zu verbessern kann man den Baum in jedem Zeitschritt inkrementell updaten, ohne ihn komplett neu aufzubauen. Die Idee resultiert aus der Beobachtung, dass der Baum sich von Zeitschritt zu Zeitschritt nur minimal ändert, da die Positionen der Partikel ja auch nur geringfügig variieren (ansonsten wäre der Zeitschritt zu groß und die Näherung somit nicht mehr vertretbar). Eine Möglichkeit, dies zu implementieren, besteht darin, nach dem Positionsupdate einmal durch die Blätter zu iterieren

und zu überprüfen, ob sie sich noch in der gleichen Zelle wie vor dem Update befinden. Ist dies der Fall, kann mit dem nächsten Partikel fortgefahren werden, anderenfalls speichert man sowohl die neue, als auch die alte Zelle des Partikels in einer Liste. Ist man mit allen Blättern fertig, stellen die Elemente der Liste die Blätter dar, die gegebenenfalls verfeinert oder zusammengefügt werden müssen, je nachdem wie viele Partikel jetzt in ihr sind.
Dass jede Zelle des Baums maximal ein Partikel enthalten darf, ist nicht fundamental für den Algorithmus. Es können je nach Anwendung durchaus auch mehr sein. Bei Implementierungen, die mehr als ein Partikel pro Zelle erlauben, müssen natürlich im letzten Schritt tendenziell weniger Blätter aktualisiert werden, was dieses Vorgehen noch effizienter macht.

Komplexität des Algorithmus von Barnes und Hut

Fasst man die Komplexitäten der Einzelschritte zusammen, erhält man einen Gesamtalgorithmus der Komplexität $\mathcal{O}(n \log n)$ für jeden Zeitschritt. Tatsächlich ist der größte Teil der Arbeit in der Kraftberechnung versteckt. Messdaten, die diese Aussage untermauern, findet man im folgenden Abschnitt.

Messungen mit dem Algorithmus von Barnes und Hut

Die folgenden Messergebnisse sind aus einer Implementierung des Barnes-Hut-Algorithmus entstanden, die der oben vorgestellten entspricht. Der einzige Unterschied ist, dass das hier Abgebildete θ der Kehrwert des oben vorgestellten ist. So bedeutet im hiesigen Algorithmus ein θ von 8, dass der Zelldurchmesser 8 mal kleiner sein muss als der Abstand Partikel-Zellschwerpunkt, um eine Näherung zu erlauben.
Die gemessene Größe ist dabei die Zeit, die für verschiedene Phasen des Algorithmus benötigt wird.
In den ersten beiden Diagrammen (Abbildung 2.4) ist die benötigte Zeit der Phasen des Algorithmus für verschiedene Werte von θ dargestellt; Die Anzahl der Partikel ist dabei stets 10.000.
Im linken Diagramm der Abbildung 2.4 erkennt man, dass die beiden Kurven unabhängig von θ fast identisch sind. Das bedeutet, dass in die Kraftberechnung der größte Teil der Zeit investiert wird. Weiterhin zeigt dieses Diagramm, dass es für die benötigte Zeit in Abhängigkeit von θ zwei Grenzwerte gibt. Der eine liegt knapp über der 0 und stellt den Extremfall dar, dass jede Interaktion direkt durch eine Interaktion mit der Wurzel genähert wird. Der andere liegt bei ca. 140 s und tritt auf, wenn keine einzige Interaktion genähert wird, also alle Partikel-Partikel-Interaktionen direkt berechnet werden. In dieser Hinsicht entspricht die Kurve also genau den bisherigen Erwartungen.
Ob die Wahl von $\theta = 2$ gut ist, lässt sich aus diesem Plot alleine noch nicht sagen. Dazu muss man auch den Fehler betrachten. Schnell ersichtlich ist

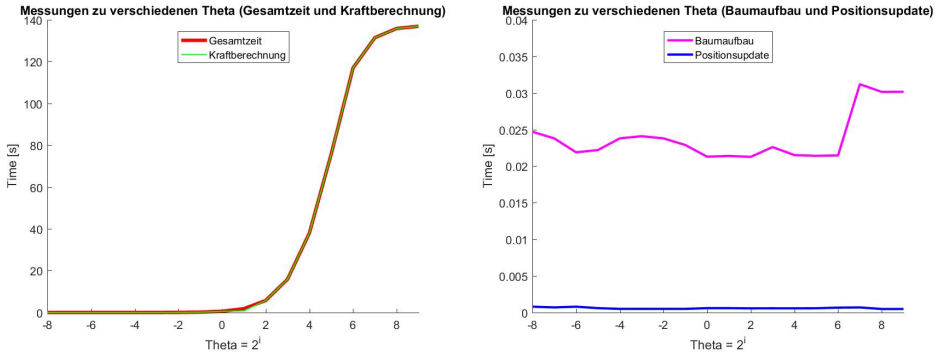

Abbildung 2.4: In der linken Abbildung ist die insgesamt benötigte Zeit des Algorithmus und die Zeit der Kraftberechnungsphase für verschiedene Werte von θ dargestellt. Auf der x-Achse sind logarithmisch (zur Basis 2) die Werte von Theta abgebildet. Sie reichen von $2^{-8} \approx 0,0039$ bis $2^9 = 512$. Die y-Achse zeigt die benötigte Zeit in Sekunden. In der rechten Abbildung ist die benötigte Zeit der Phase des Baumaufbaus sowie des Positionsupdates der Partikel für verschiedene Werte von θ dargestellt. Auf der x-Achse sind wieder logarithmisch (zur Basis 2) die Werte von Theta abgebildet, genauso wie die y-Achse die benötigte Zeit in Sekunden zeigt. Zu beachten ist hierbei der geänderte Maßstab: War der Maßstab in Abbildung 2.4 noch 0 s bis 140 s, so liegen die gemessenen Zeiten und im Intervall von 0 s bis 0.04 s.

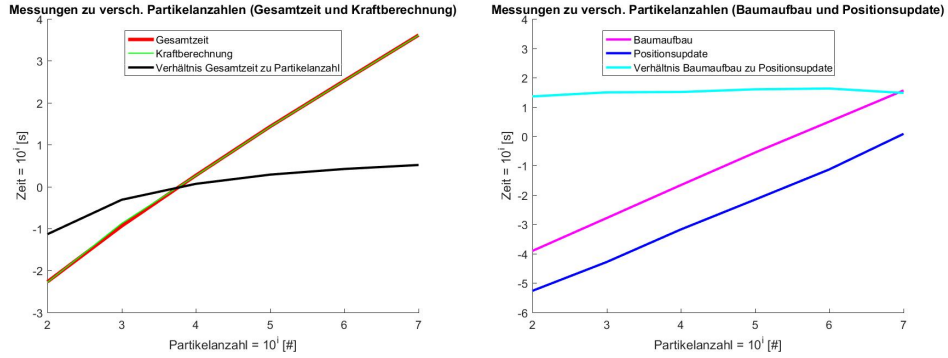

Abbildung 2.5: Links ist die insgesamt benötigte Zeit des Algorithmus und die Zeit der Kraftberechnungsphase für verschiedene Partikelanzahlen dargestellt. Beide Achsen sind logarithmisch zur Basis 10 skaliert. Die x-Achse bildet die Anzahl der Partikel von $10^2 = 100$ bis $10^7 = 10.000.000$ ab. Auf der y-Achse ist wieder die Zeit abgebildet. Die schwarze Linie gibt dabei keine Zeit an, sondern das Verhältnis der Gesamtzeit zur Anzahl an Partikeln. Rechts hingegen ist die benötigte Zeit der Phase des Baumaufbaus sowie des Positionsupdates der Partikel für verschiedene Partikelanzahlen dargestellt. Auch dieser Plot hat zwei logarithmisch (zur Basis 10) skalierte Achsen, die die benötigte Zeit und die Anzahl der Partikel abbilden.

jedoch, dass der Zeitbedarf nichtlinear mit θ steigt und die Kurve bei $\theta = 2$ noch lange nicht so stark ansteigt, wie für $\theta = 16$ zum Beispiel.
Die sehr verschiedenen Dimensionen der Zeitachsen in den Diagrammen der Abbildung 2.4 alleine untermauern nochmals die Aussage des letzten Bildes, dass die Phase der Kraftberechnung den eigentlichen Rechenaufwand verursacht. Des weiteren sind die beiden Kurven rechten Teil von 2.4 von Messungenauigkeiten abgesehen konstant. Das ergibt unter dem Gesichtspunkt Sinn, dass die Wahl von θ nur Auswirkungen auf die Anzahl der Operationen während der Kraftberechnung hat.
Das zweite Paar Plots (Abbildung 2.5) zeigt die investierte Zeit für verschiedene Partikelanzahlen. θ ist dabei immer 2. Wie auch im linken Bild aus 2.4 liegen im linken Bild von 2.5 die beiden farbigen Kurven quasi aufeinander. Das bedeutet, dass auch für variierende Partikelanzahlen die Kraftberechnung stets den größten Teil der Arbeit mit sich bringt. Eine zweite Einsicht ist jedoch noch viel wichtiger: Die schwarze Linie ist fast konstant. Wäre Sie konstant, wäre der Algorithmus von Barnes und Hut linear in der Anzahl der Partikel. So hingegen stützt dieser Verlauf die Aussage, dass die Abhängigkeit leicht superlinear ist, zum Beispiel $\mathcal{O}(n \log n)$.
Die überraschendste Kurve der Abbildung 2.5 ist die hellblaue des rechten Diagramms. Bisher war die Annahme, dass der Baumaufbau, genau wie die Kraftberechnung, eine Komplexität von $\mathcal{O}(n \log n)$ besitzt. Gleichzeitig hat das Positionsupdate der Partikel nur eine Komplexität von $\mathcal{O}(n)$, da jedes Partikel nur ein mal bewegt werden muss. Jedoch gilt für Geraden, die in einem doppelt-logarithmischen Plot parallel sind, dass sie in einem normalen Plot Polynome des selben Grades mit unterschiedlichem Vorfaktor sind. Daraus lässt sich also folgern, dass die Phase des Baumaufbaus in der Praxis deutlich weniger Aufwand mit sich bringt, als bisher angenommen, nämlich quasi linear zur Anzahl der Partikel.

1.b Das Verfahren von Appel

Ein zweiter Algorithmus wurde 1983 von A. Appel in [3] präsentiert. Der nach ihm benannte Algorithmus ist konzeptionell dem von Barnes und Hut sehr ähnlich, weist aber für homogen verteilte Partikel eine Komplexität von $\mathcal{O}(n)$ auf. Diese Tatsache war Appel bei der Veröffentlichung seiner Idee nicht bewusst; Er führt in seinem Paper Argumente für eine Komplexität von $\mathcal{O}(n \log n)$ an. 1992 wurde allerdings in [23] gezeigt, dass die tatsächliche Komplexität $\mathcal{O}(n)$ ist.

Wie auch beim Algorithmus von Barnes und Hut wird die Idee der Partikel-Cluster genutzt, um Interaktionen zu nähern. Appel geht allerdings noch einen Schritt weiter: Er berechnet nicht nur Partikel-Zell-Wechselwirkungen sondern nutzt auch Zell-Zell-Wechselwirkungen, um den Einfluss einer Menge von Partikeln auf eine weit entfernte, zweite Menge von Partikeln zu

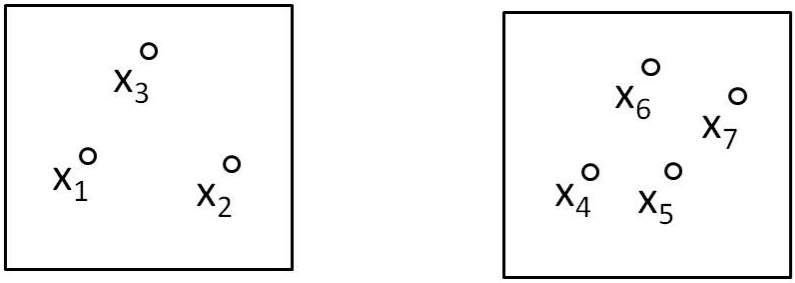

Abbildung 2.6: Zwei Zellen, die eventuell weit genug voneinander entfernt sind, damit die Wirkung der einen auf alle Partikel der anderen gleichermaßen genutzt werden kann (und andersherum).

berechnen. Wann es erlaubt ist, die Näherung zu nutzen, wird von einem Kriterium überprüft, das dem θ-Kriterium sehr ähnlich ist. Der unterschied ist, dass dieses neue Kriterium zwei Richtungen prüfen muss, da es sein kann, dass die Näherung für die Wirkung von Zelle 1 auf Zelle 2 genutzt werden kann, anders herum jedoch nicht. Wegen der konzeptionellen Ähnlichkeit wird es im Weiteren trotzdem als θ-Kriterium bezeichnet. (vgl. Abbildung 2.6)

Sowohl der Baumaufbau, als auch die Berechnungen für die Pseudopartikel und das Positionsupdate der Partikel können ohne weiteres aus dem Barnes-Hut-Algorithmus übernommen werden. Der Unterschied liegt in der Phase der Kraftberechnung. Appel geht den Baum nur ein einziges Mal von der Wurzel bis zu den Kindern durch, wobei auf jeder Ebene überprüft wird, ob Zell-Zell-Interaktionen berechnet werden können. Die Implementierung als Pseudocode findet man in [3] und eine vollständige Implementierung in [18], wobei der Algorithmus in letzterer Quelle als "Fast Multipole"bezeichnet wird.

Der Fehler, der bei diesem Algorithmus auftritt, hängt natürlich wieder davon ab, wie streng das θ-Kriterium ist. Prinzipiell ist der Fehler ähnlich wie beim Algorithmus von Barnes und Hut.

Bisher unterscheidet sich der beschriebene Algorithmus nur in der Phase der Kraftberechnung von dem zuerst genannten. Allerdings benötigt die aktuelle Methode des Baumaufbaus $\mathcal{O}(n \log n)$ Operationen. Somit wäre die Gesamtkomplexität des Verfahrens von Appel immer noch $\mathcal{O}(n \log n)$ und nicht besser. Eine Möglichkeit, diese Gesamtkomplexität zu reduzieren, be-

steht darin, den Baum in jedem Zeitschritt nur inkrementell zu verändern und nicht jedes Mal neu aufzubauen, wie es im Kapitel 2.1.a beschrieben wird. Die Anzahl der Zellen, die in der genutzten Liste stehen, ist dabei in der Ordnung $\mathcal{O}(n)$, da pro Partikel maximal zwei Zellen der Liste hinzugefügt werden können. Mit dem Argument der homogen verteilten Partikel kann man nun noch begründen, dass die Anzahl dieser Verfeinerungs- oder Vereinigungsschritte klein genug ist, um die Gesamtkomplexität des Algorithmus nicht zu erhöhen. Somit ist nun jede Phase des Algorithmus linear in der Anzahl der Partikel.

1.c Das schnelle Multipolverfahren

Das schnelle Multipolverfahren (*FMM* für Fast Multipole Method) ist ein weiteres Verfahren der Komplexität $\mathcal{O}(n)$ zur Berechnung von langreichweitigen Potenzialen (und somit auch Kräften). Es wurde in [4] von Greengard und Rokhlin erstmals präsentiert. Auch dieses Verfahren nutzt Baumstrukturen und die Idee der Näherung von Zell-Zell-Wechselwirkungen, wobei die Implementierung sich jedoch von beiden bisher vorgestellten Algorithmen unterscheidet. Zwei Arten der Entwicklung sind für diesen Algorithmus existentiell. Die Multipolentwicklung wird genutzt, um das das Potenzial einer Vaterzelle aus den Einzelpotenzialen ihrer Kindzellen zu berechnen. Ein Äquivalent in den bisherigen Algorithmen wäre die Formel, die zur Berechnung der Werte der Pseudopartikel genutzt wird. Greengard und Rokhlin nutzen jedoch eine Entwicklung, die einen beliebig hohen Grad an Genauigkeit durch mehr Näherungsterme der Entwicklung zulässt. Die zweite wichtige Entwickung ist die Taylorentwicklung. Sie wird genutzt, um in einer späteren Phase des Algorithmus das Potenzial einer Vaterzelle auf ihre Kindzellen zu übertragen. Durch die Nutzung höherer Entwicklungsgrade ist es in diesem Algorithmus möglich, den Fehler bis auf Maschinengenauigkeit zu reduzieren, ohne die Komplexität von $\mathcal{O}(n)$ zu verlieren, wie es bei Barnes und Hut der Fall war. Der mathematische Hintergrund hierzu wird in [4, 2] gegeben.

Die Implementierung von Greengard und Rokhlin nutzt, wie oben schon angedeutet, nicht direkt das θ-Kriterium, wie es aus den bisherigen Algorithmen bekannt ist. Stattdessen wird der Begriff *well separated* definiert. Der grundlegende Unterschied zum bisherigen θ-Kriterium ist, dass nach [4] nur Zellen gleicher Größe *well separated* sein können. Für die Veranschaulichung siehe Abbildung 2.7. (Eine ähnliche Abbildung ist in [4] zu finden.)

Durch die Definition dieses Begriffes kann man nun wieder entscheiden, ob man die jeweilige Multipolentwicklung einer Zelle nutzen kann, um ihre Wirkung auf andere Zellen zu berechnen. Sind die Zellen *well separated*, so ist diese Näherung ausreichend, ansonsten nicht.

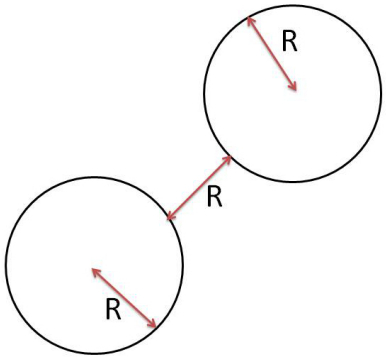

Abbildung 2.7: Die Abbildung zeigt zwei runde Zellen, bei denen der Zellrand gleichzeitig der Radius R der Multipolentwicklung ist. Sind die Mittelpunkte der Zellen mindestens 3 R voneinander entfernt, so gelten sie als *well separated*. Somit ist es gewissermaßen ein Spezialfall des θ-Kriteriums für $\theta = 3$.

Die Phasen des schnellen Multipole Algorithmus

- Baumaufbau
- Aufwärts-Schritt: Berechnung der Multipolentwicklung für alle Zellen im Baum
- Abwärts-Schritt: Berechnung der Kräfte
- Update der Partikelpositionen

Baumaufbau

Der Baumaufbau wird von Greengard und Rokhlin gar nicht angesprochen. Allerdings wurde in dieser Arbeit für das Verfahren von Appel in Kapitel 2.1.b eine Möglichkeit des Baumaufbaus beschrieben, die (bis auf den einmalig ausgeführten Initialisierungsschritt) linear in der Anzahl der Partikel ist und auf die hier zurückgegriffen werden kann.

Aufwärts-Schritt

Greengard und Rokhlin gehen im Allgemeinen von mehr als einem Partikel pro Blattzelle aus. Singh et al. erlauben in [12] zum Beispiel bis zu 40 Partikel pro Zelle. Die Massen beziehungsweise Ladungen aller Partikel werden im ersten Schritt in den Zellschwer- oder mittelpunkt entwickelt. Nachdem jede Blattzelle ihre Multipolentwicklung kennt, wird eben diese nun auch für alle gröberen Level des Baumes berechnet, solange bis selbst die Wurzel ihre Entwicklung kennt.

Abwärts-Schritt

Aus Performancegründen sollte eine Interaktion immer auf dem höchstmöglichen Level berechnet werden. Im Abwärts-Schritt wird nun für jede Zelle z jede Interaktion mit anderen Zellen berechnet, die zum einen *well separated* von z sind und deren Einfluss zum anderen nicht schon für die Vaterzelle (oder Vorherige) berechnet werden konnte.
Zudem gibt jede Vaterzelle ihr Potenzial mithilfe der lokalen Taylorentwicklung an ihre Kinder weiter. Im letzten Schritt geben die Blattzellen ihr Potenzial an die tatsächlichen Partikel weiter, die somit die gesamte Kraft kennen, die auf sie wirkt. Man akkumuliert quasi die auf die Partikel wirkenden Kräfte in einem Top-Down-Durchlauf des Baumes auf.

Update der Partikelpositionen

Auch dieser Schritt unterscheidet sich nicht von Implementierungen der anderen Algorithmen.

Weitere Anmerkungen

Da das schnelle Multipolverfahren gerade in letzter Zeit oft genutzt wird, existieren auch hierzu viele verschiedene Implementierungen. In [10], [17] und [11] sind einige Beispiele zu finden. Die tatsächliche Implementierung dieses Algorithmus erfolgt über verschiedene Listen für die Zelle i, in denen diejenigen Zellen stehen, mit denen i in einer gewissen Weise interagieren muss. Man kann Näherungen höherer Ordnung prinzipiell auch in den beiden obigen Algorithmen verwenden, siehe hierzu [18].

Kapitel 3

Parallelisierung

Das Thema der Parallelisierung spielt bei modernen Algorithmen eine immer größere Rolle. Der Grund hierfür sind die physischen Beschränkungen von Prozessoren. Dadurch, dass man nicht beliebig rechenstarke Prozessoren konstruieren kann, muss man also mehrere Prozessoren zusammen arbeiten lassen, um größere Probleme zu berechnen. Je nachdem, um welches Problem es sich dabei handelt, ist der Schritt der Parallelisierung einfacher oder komplizierter umzusetzen.

Bevor die Verteilung der Daten und die damit einhergehenden Herausforderungen beleuchtet werden, wird zuerst den Begriff der Rechenlast eingeführt und gewissermaßen quantifizierbar gemacht. Wenn dies erfolgreich geschehen ist, kann man sich mit der Aufgabe beschäftigen, diese Last möglichst gut auf alle Prozesse zu verteilen, so dass möglichst wenig Wartezeit für alle Prozesse entsteht, die Last also balanciert ist. Grunsätzlich fallen die meisten hier angeführten Verfahren in die Kategorie *Domain-Decomposition*. Man teilt dabei die physische Domäne auf die verschiedenen Prozesse auf. Die Kriterien, nach denen aufgeteilt wird, variieren natürlich.

Eigentlich ist es das erklärte Ziel, möglichst schnell ein gegebenes Problem zu lösen. Schnell impliziert, dass nach der Zeit optimiert wird. Problematisch hierbei ist, dass die benötigte Zeit sehr plattformabhängig ist: Ein Hochleistungsrechner wird ein Problem in kürzerer Zeit lösen können, als ein Desktop-PC, quasi unabhängig davon, wie gut oder schlecht der Algorithmus ist, den er nutzt. Eine weitere Facette dieser Plattformabhängigkeit ist die eigentliche Architektur der Prozessoren. So kann es sein, dass eine Implementierung der Vektoraddition auf dem Prozessor einer gewissen Firma deutlich performanter ist als auf dem Rechner einer zweiten Firma. Um das Problem so plattformunabhängig wie möglich zu gestalten, werden also in irgendeiner Weise elementare Rechenoperationen gezählt und nicht die tatsächlich verstrichene Zeit.

Das folgende Kapitel beschäftigt mit der Frage, welche Rechenoperationen man zur Quantifizierung der Last zählen könnte und welche Vor- und Nachteile sich aus der jeweiligen Metrik ergeben. Mit Metrik ist dabei keine formale, mathematische Metrik im eigentlichen Sinn gemeint, sondern ein generelles Maß. Da der größte Aufwand bei der Berechnung der Kräfte entsteht (was in Kapitel 2.1.a anhand von Messdaten gezeigt wurde), werden die präsentierten Metriken sich hauptsächlich an diesem Schritt orientieren. Trotzdem wird, speziell bei Metrik 3, auch der Einfluss des Baumaufbaus erwähnt.
Der Algorithmus von Appel wird von nun an nicht mehr explizit behandelt, da sich viele Modelle und Ideen, die sich im Zusammenhang mit den anderen beiden Algorithmen ergeben, ohne größeren Aufwand auf den Algorithmus von Appel übertragen lassen.

1 Metriken zur Quantifizierung von Rechenlast

Die folgenden Ansätze zur Lastquantifizierung steigern sich in ihrer Komplexität. Dies resultiert daraus, dass gerade bei inhomogenen Partikel-Verteilungen immer wieder Szenarien auftreten, die eine Metrik an die Grenzen ihrer Zuverlässigkeit bringt. Die danach vorgestellte Metrik wird dann so konstruiert, dass sie auch das Problem-Szenario ihrer Vorgängermetrik zuverlässig misst. Jedoch ist immer im Hinterkopf zu behalten, dass gerade bei homogenen Partikel-Verteilungen auch simple Metriken gute Performance zeigen und einfacher zu evaluieren sind als die komplexeren Varianten.

1.a Last-Metriken für Barnes und Hut

Metrik 1

Eine Möglichkeit, für die gegebene Aufgabenstellung die Rechenlast zu quantifizieren, besteht darin, die Fläche der physischen Domäne, die einem Prozess zugewiesen ist, als Indikator der Last zu betrachten. Diese Möglichkeit wird auch in [24] erwähnt. Der Vorteil dieser Methode ist ihre Einfachheit. Denn dadurch, dass die zugrundeliegende Datenstruktur eine rekursive Baumstruktur ist, ist es ein Leichtes, jedem Prozess einen gleichen Anteil der physischen Domäne zur Berechnung zuzuweisen, was ja nach dieser Metrik einer homogenen Lastverteilung entspricht.
Bei homogen verteilten Partikeln sollte diese Metrik auch gute Ergebnisse liefern, ohne nennenswerten Mehraufwand zu verursachen. Problematisch hingegen könnte es sein, wenn die Partikel inhomogen verteilt sind. In Abbildung 3.1 ist eine beispielhafte Partikelverteilung gegeben, wie sie bei der Simulation zweier Galaxien aussehen könnte. Verteilt man an jeden Prozess einen gleichen Anteil der Fläche und nutzt dazu die vorhandene Baumstruk-

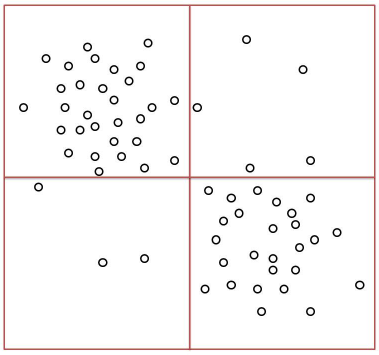

Abbildung 3.1: Eine inhohmogene Partikel-Verteilung, wie sie zum Beispiel bei der Simulation von Galaxien auftritt. In rot ist die erste Verfeinerung der Wurzel des Baumes dargestellt. Sind vier Prozesse gegeben, würde man nach Metrik 1 jeweils ein Kind der Wurzel einem Prozess zuordnen.

tur, so würde bei vier Prozessen jeder einzelne eines der vier Kinder der Wurzel zugeordnet bekommen, wie es in der Abbildung dargestellt ist. Es ist schnell einzusehen, dass dieses Vorgehen nicht zu einer tatsächlich guten Lastverteilung führt.

Anmerkung zu den folgenden Metriken

Die gerade beschriebene Metrik hat Last mit der Fläche der physischen Domäne des Simulationsgebiets gleichgesetzt. In den Metriken, deren Beschreibung nun folgt, wird Last jeweils einem Partikel zugeordnet. Das verkompliziert die Aufteilung der Last auf die Prozesse beträchtlich, weshalb diese Diskussion in Kapitel 3.3 stattfindet.

Metrik 2

Die zweite Metrik ordnet jedem Partikel eine gleiche, konstante Last zu. Bei idealer Lastbalancierung führt dies also dazu, dass jeder Prozess die gleiche Anzahl an Partikeln zur Berechnung zugewiesen bekommt. Andere Arbeiten, in denen diese Metrik angesprochen wird, sind [12], [5] und [24].
Betrachtet man nochmals die Abbildung 3.1, so erreicht man durch diese Metrik eine deutlich bessere Verteilung der Last auf die Prozesse als mit Metrik 1. Auf der anderen Seite ist die Zuweisung der Partikel auf die Prozesse, wie schon erwähnt, lange nicht mehr so einfach wie mit der letzten Metrik. Um die Tauglichkeit der Metrik zu untersuchen, wird Abbildung 3.2 zu Rate gezogen und überlegt, wie oft man bei der Berechnung der Kräfte auf das orange beziehungsweise violett umrandete Partikel die Näherung nutzen kann. Da das violette Partikel deutlich weiter von den anderen Partikeln entfernt ist als das orange, kann man davon ausgehen, dass für violett

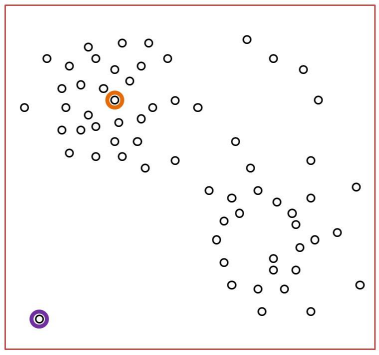

Abbildung 3.2: Eine weitere inhohmogene Patikelverteilung. Die rote Umrandung ist die gesamte, physische Domäne und die farblich umkreisten Punkte sind die im Text relevanten.

deutlich häufiger genähert werden darf als für orange. Somit verursachen die Partikel nicht die gleiche Last.
Zusammenfassend kann man also sagen, dass Metrik 2 besser ist als die zuerst vorgestellte, jedoch immer noch nicht wirklich gut ist und dafür aber einen großen Mehraufwand bei der Lastverteilung mit sich bringt.
Abbildung 3.3 zeigt für ein Beispiel-Szenario von 1.000 Partikeln, welche Last welchem Blatt (das jeweils ein Partikel enthält) durch Metrik 2 zugeordnet wird.

Metrik 3

Will man die Idee aus Metrik 2 fortführen und dabei jedem Partikel eine individuelle Last zuordnen, stellt sich die Frage, welche Partikel überhaupt eine hohe, beziehungsweise niedrige Last erzeugen. Aus Abbildung 3.2 wurde gefolgert, dass Partikel innerhalb einer Wolke deutlich weniger Näherungen nutzen können als weiter Abseits liegende. Ein heuristischer Indikator dafür, wie sehr ein Partikel Teil einer Wolke ist, ist zum Beispiel seine Tiefe im Baum (auch in [24] zu finden).
Der Mehraufwand verglichen mit Metrik 2, der durch die Berechnung dieser Metrik entsteht, ist vernachlässigbar, da sich jedes Partikel nur seine Tiefe im Baum merken muss. Man kann für Metriken 2, 3 (und auch 4) die gleichen Verfahren verwenden, um den Prozessen die jeweiligen Partikel zuzuweisen. Diese Erkenntnis folgt aus Überlegungen in Kapitel 3.3. Jedoch gibt es es auch für diese Metrik Szenarien, in denen sie die entstehende Last nicht adäquat schätzt. Ein mögliches ist in Abbildung 3.4 zu sehen. Dadurch, dass die orange umkreisten Punkte nah beieinander liegen, müssen sie sehr tief in den Baum einsortiert werden. Trotzdem sind sie weit entfernt von einem Großteil der Partikel, weshalb eine Näherung in vielen Fällen möglich ist. Diese Metrik überschätzt also den tatsächlichen Aufwand (zumindest

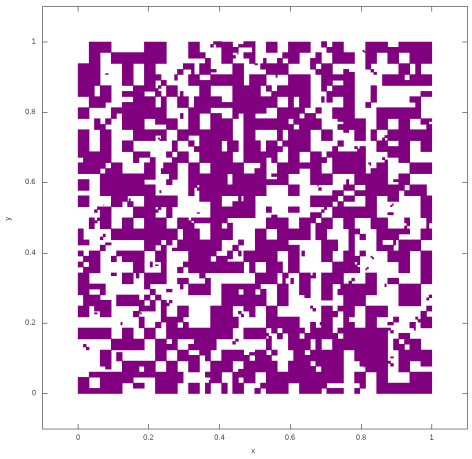

Abbildung 3.3: Diese Abbildung zeigt die Schätzung der Last durch Metrik 2. Dabei sind alle Blätter dargestellt. Da jedes Blatt maximal ein Partikel enthält, kann man die Last eines Blattes mit der Last des enthaltenen Partikels identifizieren. Es sind insgesamt 1.000 Partikel in der physischen Domäne verteilt, die sich von 0 bis 1 in beiden Richtungen erstreckt. Die Partikel sind innerhalb der Domäne gleichmäßig verteilt. Die Last wird durch die Farbgebung der Zellen verdeutlicht. Weiße Zellen enthalten kein Partikel. Da die Last einer jeden gefüllten Zelle gleich geschätzt wird, sind alle gefüllten Zellen gleichfarbig.

für die Phase der Kraftberechnung).
Anders verhält es sich jedoch für die Phase des Baumaufbaus. Hier lässt sich leicht überprüfen, ob die erzeugte Last proportional zu der Tiefe des Partikels im Baum ist, da es genau einmal weitergereicht beziehungsweise neu eingefügt werden muss, um eine Ebene tiefer zu gelangen. Für die Schätzung der Last in der Phase des Baumaufbaus ist diese Metrik also ein optimales Maß. Wie oben schon erwähnt, stellt jedoch die tatsächliche Kraftberechnung den größten Aufwand während der Simulation dar. Natürlich sind Baumdurchläufe auch für diese Phase des Algorithmus von elementarer Bedeutung, trotzdem wäre es gut, eine Metrik zu haben, die die dort entstehende Last *optimal* abbildet. Abbildung 3.5 zeigt, welche Last welchem Blatt durch Metrik 3 zugeordnet wird.

Metrik 4

Die Metrik, die ich zu diesem Zweck vorstellen will, wird in den meisten modernen Implementierungen [12] [8] [9, 2.1] [25] verwendet. Die Idee ist, dass man die Anzahl der zu berechnenden Interaktionen für jedes Partikel vor der Kraftberechnung separat berechnet. Somit hat jeder Prozess an-

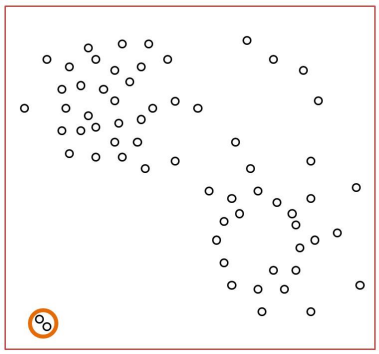

Abbildung 3.4: Ähnlich wie Abbildung 3.2 mit dem Unterschied, dass nun 2 Partikel nah beieinander, jedoch weit entfernt von den anderen in der unteren, linken Ecke der Domäne liegen.

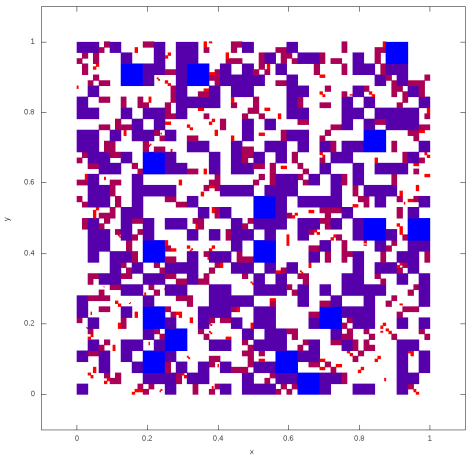

Abbildung 3.5: Hier ist die durch Metrik 3 geschätzte Last zu sehen. Je roter eine Blattzelle dargestellt wird, desto mehr Last erzeugt das enthaltene Partikel. Es sind wieder 1.000 Partikel in der Domäne verteilt. Man erkennt schnell, dass große Zellen blau und kleine Zellen rot dargestellt werden.

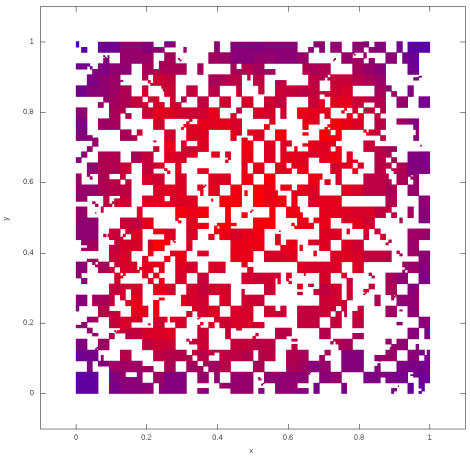

Abbildung 3.6: Abbildung der Lastschätzung durch Metrik 4. Die Farbgebung ist dabei wie in Abbildung 3.5. Wie auch vermutet, sind Zellen im Zentrum die rechenintensiveren.

schließend die gleiche Anzahl an Interaktionen zu berechnen. Diese Metrik ist allerdings auch am aufwändigsten zu berechnen. Einen großen Teil des Aufwands kann man sich jedoch mit folgender Überlegung sparen: Die Bewegung der Partikel erfolgt sehr langsam, in jedem Zeitschritt wird also von jedem Partikel nur eine geringe Distanz zurückgelegt. Ist dies nicht der Fall, so ist die Zeitschrittweite zu groß gewählt und eine Näherung von zeitschrittweise konstanten Kräften ist falsch.
Dadurch, dass sich die Partikelpositionen also nur minimal von Zeitschritt zu Zeitschritt ändern, kann man die Anzahl der berechneten Interaktionen des letzten Zeitschritts als gute Schätzung für die Interaktionen im aktuellen Zeitschritt verwenden. So kann man zum Beispiel in jedem Zeitschritt und für jedes Partikel alle auftretenden Interaktionen zählen und erhält somit ein gute Lastschätzung für den nächsten Zeitschritt. Abbildung 3.6 stellt die durch Metrik 4 zugewiesene Last exemplarisch dar.

Alle vier Metriken vereint, dass ihre Berechnung die Gesamtkomplexität des Algorithmus nicht erhöht! Dies ist leicht ersichtlich, da selbst die aufwändig zu berechnenden Metriken 3 und 4 durch elementare Operationen im Standartverlauf des Algorithmus erfasst werden können.

1.b Last-Metriken für das schnelle Multipolverfahren

Die Idee, jedem Prozess ein gleich großen Stück der physischen Domäne zur Berechnung zu überlassen, ist natürlich direkt auf das schnelle Multipolverfahren übertragbar. Allerdings war eine grundlegende Erkenntnis des letzten Kapitels, dass inhomogene Verteilungen eine schlechte Lastbalancierung mit

sich bringen würden, weshalb diese Metrik nicht genutzt wird. Will man die anderen Metriken auf das schnelle Multipolverfahren übertragen, stößt man jedoch auf ein grundsätzliches Problem: Last wurde bisher immer mit Partikeln assoziiert. Das schnelle Multipolverfahren arbeitet allerdings erstens auch auf den inneren Zellen des Baums und zweitens umfassen in den meisten Implementierungen selbst die Blattzellen mehrere Partikel (z.B. in [12]). Die bisher erarbeiteten Metriken müssen also auf Zellen verallgemeinert werden.
Dieses Problem wird ebenfalls in [11] und [12, 8] erwähnt. Im Folgenden wird eine Metrik aus [11] umrissen, aus der viele generelle Konzepte übernommen werden können.
Grundsätzlich berechnet man die durch Blätter beziehungsweise innere Knoten erzeugte Last auf zwei verschiedene Arten. Um die Last einer inneren Zelle im Zeitschritt t zu schätzen, zählt man die Interaktionen, die diese Zelle auszuführen hat. Dies kann man mithilfe von Interaktionslisten implementieren, die auch später im Algorithmus wieder verwendet werden können. Komplizierter ist die Lastabschätzung für Blätter. Hierzu wird für jede Art der Interaktion (von denen es je nach Implementierung mehr oder weniger gibt) durch eine einfache Funktion die entstehende Last berechnet. Die genannte Funktion hängt dabei von der Partikelanzahl der Zelle und der Ordnung der genutzten Näherung ab. Nun werden alle Einzellasten (die gerade berechnet wurden) für alle Partikel aufsummiert, was der Gesamtlast dieses Blattes entspricht.
Ideal wäre es wieder, die Last aus Zeitschritt $t - 1$ als Schätzung im Zeitschritt t zu verwenden. Dies ist jedoch nicht ohne weiteres möglich, da Zellen (im Allgemeinen), im Gegensatz zu Partikeln nicht über die Zeitschritte hinaus erhalten bleiben. Um diese Herausforderung zu meistern projiziert jede Blattzelle ihre Last am Ende eines Zeitschritts gleichmäßig auf alle Partikel, die sie enthält. Die Partikel bleiben logischerweise erhalten. Somit kann im nächsten Zeitschritt jede Zelle ihre geschätzte Last aus der Summe aller Partikel-Einzellasten berechnen.
Auch wenn diese Metrik sehr aufwändig wirkt, so ist der entstehende Mehraufwand nach [11] vernachlässigbar.

2 Speicher-Architekuren

Bevor diese Arbeit auf die Aufteilung der Last und eigentlichen Parallelisierung eingeht, folgt in diesem Kapitel eine kleine Einführung in das Thema der Speicher-Architekturen. Bisher wurde angenommen, dass Daten per se zugänglich sind. Der folgende Abschnitt wird zeigen, dass es auf der einen Seite Architekturen gibt, in denen man sich über Speicherlokalität tatsächlich keine Gedanken machen muss, da alle Daten in einem großen Speicher gehalten werden, auf den alle Prozessoren gleichermaßen zugreifen

können. Es gibt jedoch auf der anderen Seite auch Architekturen, bei denen es reiflicher Überlegung bedarf, wie ein paralleler Algorithmus effizient implementiert werden kann.

2.a Shared-Memory-Architekturen

Wie der Name schon sagt gibt es in einer Shared-Memory-Architektur nur einen geteilten Arbeitsspeicher für alle Prozessoren. Da man sich also keine Gedanken machen muss, wo die verschiedenen Daten liegen, sind viele Programme ohne großen Mehraufwand auf diesen Architekturen parallelisierbar. Man muss sich allerdings Gedanken zur Synchronisation machen. Wenn mehrere Prozesse auf die gleiche Speicherzelle zugreifen wollen, kann es hier zu Komplikationen kommen. Mehr Informationen dazu findet man auf Seite 63 in [26].
Mit immer größeren Rechenclustern ist es jedoch nicht möglich, konkurrenzfähige Supercomputer mit einem großen Speicher zu bauen.

2.b Distributed-Memory-Architekturen

Aus diesem Grund haben modernen Großrechner fast ausschließlich Distributed-Memory-Architekturen. In diesen hat jeder Prozessor (der wiederum mehrere Rechenkerne haben kann) seinen eigenen Arbeitsspeicher mit eigenem Adressraum. Das führt dazu, dass Skalierung die Effizienz der Architektur nicht direkt einschränkt, da man einfach mehr Prozessoren in ein größeres Cluster zusammenführen kann, die natürlich auf ihren privaten Speicher eine geringe Zugriffszeit haben. Um große Probleme parallel zu lösen, muss es auch Kommunikation unter den verschiedenen Prozessoren geben. Dazu sind clusterweite Netzwerke vorhanden, über die Daten transferiert werden können. Diese Kommunikation stellt allerdings eine weitere Lastquelle dar, da sie nicht vernachlässigbaren Aufwand mit sich bringt. Durch geschicktes Anpassen der oben vorgestellten Lastmetriken kann man diese Last auch bei der Partikelpartitionierung berücksichtigen. Auch zu dieser Architektur findet man auf Seite 63 in [26] weitere Informationen.
Dadurch, dass jeder Prozessor nur noch über einen kleinen Teil der Daten verfügt, muss man sich intensiv damit beschäftigen, wie man große Probleme löst, ohne übermäßig viel Kommunikationsaufwand zu verursachen. Eine Möglichkeit, an dieses Problem heranzugehen ist eine Master-Slave-Einteilung der Prozesse, bei denen die Kommunikation nur zwischen Master und Slave stattfindet und nicht zwischen den Slaves. In der Praxis wird jedoch oft eine andere Lösung bevorzugt, da der Kommunikationsoverhead sehr groß ist.

2.c Distributed-Shared-Memory-Architekturen

Auf [26, S. 64] wird eine weitere Architektur vorgestellt, die die Vorteile der beiden gerade genannten vereinen soll. Eine sogenannte Distributed-Shared-Memory-Architektur. Auf einem physisch getrennten Speicher arbeiten latente Routinen, die es dem Programmierer ermöglichen sollen, Programme wie für Shared-Memory-Architekturen zu schreiben. Singh et al. nutzen in [12] zum Beispiel eine "DASH-Architektur". Ihre Besonderheiten sind Cache-Kohärenz und ein gemeinsamer Adressraum, der sie zu einer Distributed-Shared-Memory-Architektur macht [27]. Auf der anderen Seite gilt nach dem No-Free-Lunch-Theorem, dass eine allgemeine Kommunikationsroutine nicht so gut sein kann wie eine Routine, die speziell für ein bestimmtes Problem geschrieben wurde. Aus diesem Grund haben die modernen Großrechner fast ausschließlich Distributed-Memory-Architekturen.

3 Partikelpartitionierung

Nachdem jedes Partikel beziehungsweise jede Zelle oder jedes Gebiet der physischen Domäne durch eine der oben genannten Metriken eine Last zugewiesen bekommen hat, stellt sich nun die Frage, wie genau die Partikel oder Zellen auf die Prozesse verteilt werden können, um eine möglichst gute Lastbalance zu erhalten. Ideal wäre, wenn kein Prozess jemals warten müsste. Des weiteren sollte es auf Distributed-Memory-Architekturen möglichst wenig Kommunikationsoverhead geben (vgl. [14, S. 636]). Eine Möglichkeit, die Kommunikation zu minimieren, besteht für baumbasierten Methoden darin, den Prozessen möglichst lokal zusammenhängende Teildomänen zuzuweisen, da durch das θ-Kriterium die meisten Interaktionen für nah beieinander liegende Partikeln beziehungsweise Zellen berechnet werden müssen [12, 7f.]. Die oben genannte Eigenschaft der lokal zusammenhängenden Teildomänen wird im weiteren als physische Lokalität bezeichnet. Eine gute physische Lokalität äußert sich auch im Verhältnis zwischen Volumen und Oberfläche der einzelnen Teildomänen. Je größer das Volumen im Verhältnis zur Oberfläche ist, desto besser ist die physische Lokalität.
Grundsätzlich kann man die verschiedenen Partitionierungsansätze in zwei Kategorien unterteilen: Bei der Variante, die im Folgenden als statisches Partitionieren bezeichnet wird, werden alle Partikel neu auf die Prozesse verteilt. Beim dynamischen Partitionieren (oft auch dynamisches Repartitionieren genannt) hingegen baut man auf eine bereits bestehende Partitionierung auf und führt nur inkrementelle Updates durch, bei denen nur ein kleiner Teil der Partikel neu zugeordnet wird. In der Praxis kann es natürlich auch hybride Varianten geben, die zum Beispiel prinzipiell dynamisch arbeiten, aber nach einer gewissen Anzahl von Zeitschritten alle Partikel neu verteilen. Im Folgenden werde ich gängige Verfahren für beide Partitionierungsansätze mit ihren jeweiligen Vor- und Nachteilen präsentieren. Im All-

gemeinen kann statisches Partitionieren bessere Ergebnisse erzielen, da es größere Einflussmöglichkeiten hat, auf der anderen Seite ist dynamisches Partitionieren deutlich performanter, da im Allgemeinen deutlich weniger Partikel überhaupt betrachtet und gegebenenfalls neu zugewiesen werden müssen.

3.a Statisches Partitionieren

Randomisierung

Eine Möglichkeit, die Last möglichst gleichmäßig auf die Prozesse zu verteilen, baut auf die Nutzung des Zufalls: Hat man p Prozesse zur Verfügung, unterteilt man die physische Domäne in $d \geq p$ Teildomänen und weist diese randomisiert den Prozessen zu [8]. Der große Vorteil dieses Verfahrens ist, dass man überhaupt keine Lastmetrik benötigt, um es anzuwenden. Außerdem kann man dieses Verfahren auf die Partikel-Verteilung anpassen. Hat man zum Beispiel eine sehr homogene Partikelverteilung gegeben, so reicht $d = p$, da somit jedem Prozess in etwa die gleiche Last zugewiesen wird. Weiß man hingegen, dass die Partikel sehr inhomogen verteilt sind, kann man $d \gg p$ wählen und hoffen, dass jeder Prozess ein paar der dicht besetzten Teildomänen und ein paar der weniger dicht besetzten abbekommt. Nach dem zentralen Grenzwertsatz sollte dieses Verfahren für $d \to \infty$ zu idealer Lastverteilung führen (vgl. Abbildung 3.7). Bei der homogenen Partikel-Verteilung links erkennt man, dass $d = p$ ausreicht, um jedem Prozess die gleiche Anzahl an Partikeln zukommen zu lassen. Bei der inhomogeneren Verteilung rechts wird durch $d = 16$ auch eine gute Aufteilung der Partikel auf die Prozesse erzeugt. Trotzdem ist nicht garantiert, dass man eine wirklich gute Lastbalance erreicht.

Bei verteiltem Speicher ist die oben beschriebene physiche Lokalität jedoch ein ernst zu nehmendes Problem dieses Verfahrens. Es lässt sich auch nur schwer beheben, da es ausdrücklich erwünscht ist, dass die verschiedenen Teildomänen, die ein bestimmter Prozess zugewiesen bekommt, aus ganz verschiedenen Teilen der Gesamtdomäne stammen, um eine möglichst gute Lastbalance zu erreichen. Für Shared-Memory-Architekturen ist es jedoch ein gutes Verfahren, da es ohne Frage sehr einfach umzusetzen ist und dafür gute Ergebnisse liefert. Weiterführende Informationen sind in [8] zu finden. Um dieses Verfahren für das schnelle Multipolverfahren nutzbar zu machen, sollte man bei der Erstellung der Teildomänen darauf achten, dass diese sich mit den Grenzen der Zellen des Quartär- beziehungsweise Oktalbaums decken. So kann man das Problem, dass Last auch in Zellen und nicht nur Partikeln auftritt, geschickt umgehen.

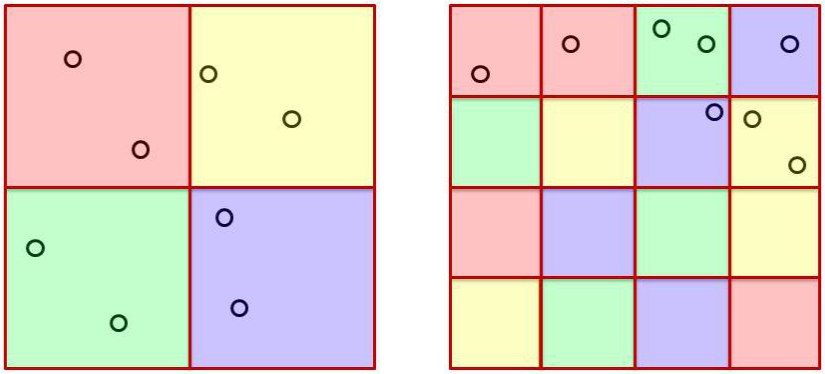

Abbildung 3.7: In dieser Abbildung ist die Zuweisung von Teildomänen auf vier Prozesse dargestellt, bei der d im linken Fall 4 und im rechten 16 ist. Die schwarzen Kreise stellen die Partikel dar und die rotem Umrandungen schließen Teildomänen ein. Die bunten Schattierungen der Teildomänen sollen veranschaulichen, welchem der vier Prozesse sie zugewiesen sind.

Orthogonal Recursive Bisection (ORB)

Der größte Malus der Randomisierung ist, dass die pysische Lokalität in keinster Weise gegeben ist. Um Kommunikation weitestgehend zu vermeiden ist dies jedoch eine wünschenswerte Eigenschaft. Das ORB-Verfahren versucht zum einen, eine der oben eingeführten Metriken für Last zu nutzen, um die Arbeit möglichst gut auf die Prozesse zu verteilen, und zum anderen physische Lokalität zur Kommunikationsminimierung zu erreichen.
Dazu teilt es die physische Domäne rekursiv in jeweils zwei Teildomänen, die möglichst gleiche Last erzeugen. Dies geschieht durch Optimierung entlang einer Dimension, ist also erschwinglich. Um die physische Lokalität zu optimieren wird die Dimension, in der die Domäne unterteilt wird in jedem Rekursionsschritt gewechselt (vgl. Abbildung 3.8).

Die Rekursion wird in dem Moment abgebrochen, in dem es für jeden Prozess genau eine Teildomäne gibt. Ideal ist deshalb für dieses Verfahren, wenn die Anzahl der Prozesse $p = 2^i$ mit beliebigem i ist.
Das Verfahren hat mehrere Vorteile gegenüber der randomisierten Methode. Es können zum Beispiel Lastmetriken verwenden werden, um den Prozessen nicht nur mit einer gewissen Wahrscheinlichkeit die gleiche Last zuzuteilen. Außerdem ist die physische Lokalität deutlich eher gegeben. Um sie noch weiter zu verbessern kann man in Fällen, in denen der Wechsel der Dimension zu einer Entartung der Domänen führt, auch erlauben, den Wechsel für diesen Schritt der Rekursion auszusetzen.
Auf der anderen Seite ist dieses Verfahren generell mit deutlich höherem Implementierungs- und Rechenaufwand verbunden. Man benötigt auch eine

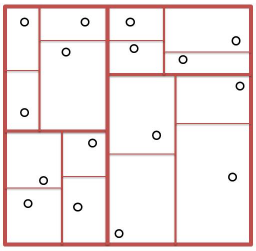

Abbildung 3.8: Die Abbildung zeigt eine Aufteilung von 16 Partikeln mittels Orthogonal Recursive Bisection. Die Stärke der Unterteilungslinien nimmt mit der Tiefe der Rekursion ab. Zu beachten ist, dass normalerweise nur so lange verfeinert wird, bis jeder Prozess (nicht jedes Partikel!) eine Teildomäne hat. In diesem Fall sind also die schwarzen Punkte zum Beispiel als Partikel-Wolken zu verstehen, die alle die gleiche Rechenlast verursachen.

zweite Baumstruktur, die sich von der bisherigen stark unterscheidet. Diese Bäume werden oft als *k-d-Bäume* bezeichnet. Ein *k-d-Baum* ist ein Suchbaum, der eine Menge k-dimensionaler Punkte speichert. Der Bereich, in dem die zu speichernden Punkte liegen, wird dabei rekursiv immer in Hyperebenen unterteilt. Weitere Informationen zur ORB sind in [12] und [13, 2.2] zu finden.
Bei der Implementierung des ORB-Verfahrens wird zuerst der Algorithmus von Barnes und Hut betrachtet. Da hier Last mit Partikeln identifiziert werden kann, die punktuelle Objekte ohne räumliche Ausdehnung sind, werden die Partikel entlang aller Dimensionen sortieren und dann nacheinander in dieser Reihenfolge abgezählt, bis die Hälfte der Gesamtlast erreicht ist. Bei dem schnellen Multipolverfahren ist diese Unterteilung aufgrund der Last von inneren Zellen nicht intuitiv – bisher hatten nur Partikel, die als punktförmig angenommen werden, eine Last. Nun muss man sich Gedanken machen, inwieweit man räumliche Zellen zum Beispiel nach den Dimensionen sortieren kann, ect. Die im Folgenden gezeigte Lösung stützt sich hauptsächlich auf [11, 5.2.1].
Die erste Idee ist, eine Zelle mit ihrem Schwerpunkt, respektive Mittelpunkt (je nach Implementierung) zu identifizieren. Nutzt man die Schwerpunkte, so ist dieses Verfahren anwendbar; Probleme treten vor allem auf, wenn man die Mittelpunkte nutzt. Dadurch dass die Struktur des Baumes hierarchisch ist, sind die Koordinaten der Mittelpunkte in einer Dimension oft exakt gleich. Wenn man diese nun in dieser Dimension sortiert, ergeben sich exakt die gleichen Positionswerte und die entsprechenden Zellen müssen zwangsläufig demselben Prozess zugewiesen werden, was nach [11, 5.3] zu großen Last-

Imbalancen führt. Dieses Problem lässt sich zum Beispiel lösen, in dem man solche problematischen Punkte identifiziert und die Zellen nach der eigentlichen Zuweisung nochmals umverteilt, bis eine gute Balance der Last erreicht ist. Testergebnisse der verschiedenen Implementierungen finden sich in [11, 5.3].

Costzones

Im ORB-Verfahren wurde mit Hilfe einer weiteren Baumstruktur eine gute Aufteilung der Last auf die verschiedenen Prozesse erreicht. Das Prinzip der Costzones versucht nun durch Nutzung der schon vorhandenen, rekursiven Quartär- beziehungsweise Oktalbaumstruktur ähnlich gute Ergebnisse in der Lastbalancierung zu erreichen. Costzones finden in vielen Implementierungen Anwendung. Einige Beispiele sind die Arbeiten von [12], [5] und [13, 2.2].
Es folgt zuerst die Betrachtung für den Algorithmus von Barnes und Hut. Betrachtet man die Menge aller Blätter, deren Last bekannt ist, stellt sich in erster Linie die Frage, wie physische Lokalität erreicht werden kann. Eine gängige Herangehensweise baut auf raumfüllenden Kurven auf, wie sie in Abbildung 3.9 zu sehen sind. Raumfüllende Kurven sind rekursiv aufgebaute Kurven, die nach einem bestimmten, einfachen Muster aufgebaut werden, wie es im unteren Teil der Abbildung zu sehen ist. Sie wurden erstmals von Warren [15] in diesem Kontext erwähnt. Eine Arbeit, die sich mit der effizienten Implementierung einer dieser Kurven (der Hilbertkurve) beschäftigt, ist [16].

Als erstes berechnet man nun die Gesamtlast und teilt sie durch die Anzahl der zur Verfügung stehenden Prozesse, um die Last zu erhalten, die jedem Prozess bei perfekter Lastbalance zugeteilt werden sollte. Um die Blätter dann auf die Prozesse zu verteilen, folgt man der Kurve, wobei man die Lasten der durchlaufenen Zellen aufsummiert. Hat man so viel Last aufsummiert, wie einem Prozess obiger Rechnung nach zusteht, weist man die bisher durchlaufenen Partikel diesem Prozess zu und wiederholt die Prozedur beginnend in der nächsten Zelle, die noch keinem Prozess zugewiesen wurde.
Man kann beweisen, dass mit Hilbert-Kurven optimal kompakte und somit auch physisch lokale Teildomänen für jeden Prozess entstehen. Aber auch Morton-Kurven zeigen gute Ergebnisse [12].
Somit wird also sowohl optimale Verteilung der Last (zumindest unter der Annahme, dass die Metrik optimale Ergebnisse liefert), als auch gute physische Lokalität, ohne eine weitere Baumstruktur nutzen zu müssen, erreicht.
Es gibt eine weitere Möglichkeit, Costzones zu implementieren. Dabei geben in einem ersten Baumdurchlauf alle Kinder ihre Last an ihre Eltern weiter, welche die Einzellasten aufsummieren. Schlussendlich hat also jede Zelle die Gesamtlast ihrer Teildomäne gespeichert. Da nun auch für die inneren Zellen

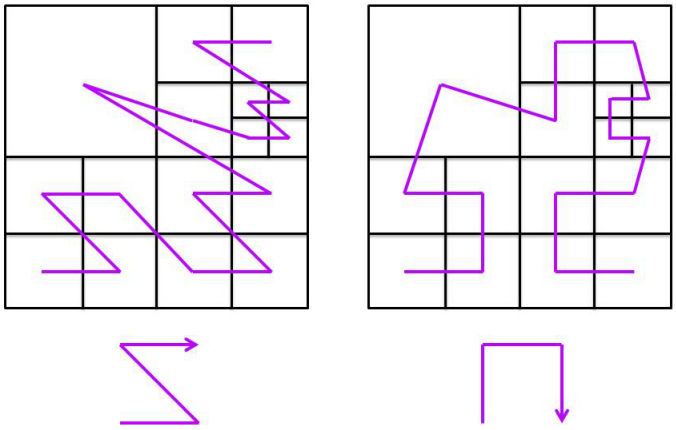

Abbildung 3.9: Gezeigt sind zwei veschiedene raumfüllende Kurven, die man als Morton- (links im Bild)/ oder Hilbertkurven (rechts) bezeichnet. Oben sieht man einen Quartärbaum, dessen Blätter entsprechend der Kurve unten links beginnend durchlaufen werden. Unter der Darstellung des Baumes findet man das zugrunde liegende Basismuster der jeweiligen Kurve.

des Baums, das heißt für die Pseudopartikel, in denen mehrere Einzelpartikel zusammengefasst sind, Werte für die Last bekannt sind, kann man die Costzones auch top down berechnen, was dazu führt, dass jeder Prozess schon von Haus aus eine kleinstmögliche Menge an Zellen kennt, die seine komplette Teildomäne abdecken. Die Zellen, die in dieser kleinstmöglichen Menge enthalten sind, werden im Folgenden Indexzellen genannt. In Architekturen mit verteiltem Speicher benötigt man sie später, um mit ihnen die komplette Teildomäne eines jeden Prozesses zu identifizieren. Details zu den zwei Implementierungen finden sich in [9].
Um diese Methode der Lastverteilung auf die schnelle Multipole Methode übertragen zu können ([11, 5.2.2]), muss man nicht allzu viel ändern. Man muss nur auch die inneren Zellen des Baumes auch auf die Prozesse verteilen. Die einzige Frage, die noch offen bleibt ist, wie man die raumfüllenden Kurven auch für die inneren Zellen des Baumes nutzen kann. Dazu gibt es verschiedene Ansätze, die aber alle intuitiv sind. Die Grundidee besteht hierbei oft darauf, in jeder Zelle zwei Werte für Last zu speichern: Einen Wert für die Last der Zelle und einen Wert für die Gesamtlast aller darunter liegenden Zellen. Auf dieser Basis kann man das bisher verwendete Verfahren quasi kopieren. Eine Veröffentlichung, die genauer hierauf eingeht, ist [11, 5.2.2].

Graphbasiertes Partitionieren

Teng formuliert in [14] eine Idee, die sich von den bisherigen Methoden grundlegend unterscheidet. Das Problem der Partitionierung wird mit Hilfe eines Graphen gelöst, bei dem die Knoten des Graphen Arbeitsaufgaben entsprechen, die getätigt werden müssen und denen man eine Last zuweisen kann. Je nach Implementierung können diese Aufgaben zum Beispiel mit gefüllten Blättern des bisherigen Baums identifiziert werde. Die Kanten stellen die Kommunikation zwischen den Zellen dar. Sie sind gewichtet, um die Menge der nötigen Kommunikation darzustellen, die hier getätigt werden muss und sie sind gerichtet, um darzustellen, in welcher Richtung die Kommunikation stattfindet. Wie der Graph im Detail aufgebaut wird, findet man in [14, 3].
Nachdem man den Graphen aufgebaut hat, unterteilt man ihn unter zwei Bedingungen: Erstens soll die Gesamtlast jedes Teilgraphen gleich groß sein und zweitens sollten die Schnitte minimal sein, also Kanten möglichst geringen Gesamtgewichts schneiden. Dadurch, dass die Kantengewichte die Menge an nötiger Kommunikation darstellen, stellt eine Unterteilung des Graphen nach obigen Kriterien eine Partitionierung der Zellen des Baumes in Teile gleich großer Last und mit möglichst geringer Kommunikation sicher. Teng zeigt in [14] eine effiziente Methode, diese Schnitte zu finden.

3.b Dynamisches Partitionieren

Bei dynamischen Verfahren ist zu beachten, dass zu Beginn die Partikel schon einmal zugewiesen sein müssen, da dynamische Verfahren nur inkrementell arbeiten. Prinzipiell ist dabei vollkommen egal, welches Verfahren genutzt wird, um zu dieser ersten Zuweisung zu kommen.

Dynamische Costzones

Dynamische Costzones stellen eine Erweiterung der bisher bekannten Costzones dar, die zum Beispiel von [8], [13, 2.2] und [9] verwendet wird. Die erste Zuweisung verläuft dabei wie bei den oben beschriebenen Costzones. Einmal initialisiert ist das Vorgehen wie folgt:
Nach jedem Zeitschritt wird die Gesamtlast des Systems ermittelt und daraus berechnet, wie viel Last jeder Prozess im nächsten Zeitschritt tragen muss. Nun geht man die Prozesse der Reihe nach durch und überprüft, ob sie nach der aktuellen Partitionierung die gerade berechnete Last abbekommen haben. Die Reihenfolge, in der die Prozesse überprüft werden, ist dabei durch die raumfüllende Kurve gegeben, nach der die Zellen den Prozessen der Reihe nach zugeteilt werden. Hat ein Prozess zu wenig Last, nimmt er dem nächsten Prozess solange Zellen ab, bis er genug Last hat. Auch hierbei folgt er der raumfüllenden Kurve, um die physische Lokalität beizubehalten. Hat ein Prozess hingegen zu viel Last, so gibt er dem nächsten Prozess seine

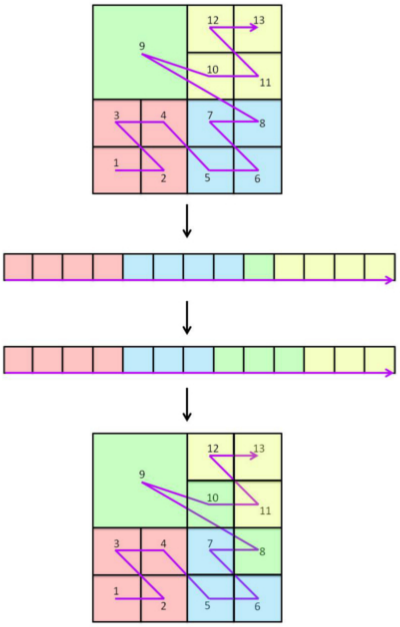

Abbildung 3.10: Die Abbildung verdeutlicht den Ablauf der dynamischen Costzones. Dazu ist eine hierarchisch aufgeteilte, physische Domäne in schwarz, die raumfüllende Kurve durch alle Blätter in violett und die Zugehörigkeit zu einem der vier Prozesse in rot, blau, grün oder gelb dargestellt. Es wird angenommen, dass die Zellen alle die gleiche Last erzeugen (nämlich 1), unabhängig davon wie groß sie sind. Die waagrechten Strukturen stellen die Blätter des Baumes dar, wie sie der raumfüllenden Kurve nach sortiert sind.
Der oberste Teil stellt die physische Domäne am Ende eines Zeitschritts dar.

letzten (in der Reihenfolge der raumfüllenden Kurve) Zellen ab. Schlussendlich kommuniziert jeder Prozess jedem anderen seinen Startpunkt innerhalb der Kurve, damit immer klar ist, welcher Prozess für welche Zellen zuständig ist. Das Verfahren ist in Abbildung 3.10 zu finden. Da jede Zelle eine Last von 1 erzeugt, ist die Gesamtlast des Systems offensichtlich 13. Es werden also bei einem der Prozesse 4, bei allen anderen 3 zugeordnete Zellen akzeptiert.
Es folgt eine genauere Betrachtung der einzelnen Prozesse: Der "rote"Prozess hat 4 zugewiesene Zellen, also behält er diese, da 4 Partikel für einen der Prozessen erlaubt sind. Der nächste, nämlich der "blaue"Prozess hat auch 4, was aber nun nicht mehr erlaubt ist, also gibt er seinem nächsten, dem "grünen"Prozess seine letzte Zelle (Zelle 8) ab. Prozess "grün"hat somit 2,

was eine zu wenig ist. Also bekommt er die erste des "gelben"Prozess (Zelle 10). Der "gelbe"hat nun also noch 3, wie ursprünglich angestrebt. Damit ist der Umverteilungsprozess beendet. Die neue Zuweisung ist unten im Bild zu sehen.

Dynamic Task Scheduling

Ein weiteres dynamisches Partitionierungsverfahren wird in [13, 2.3] beschrieben. Alle bisherigen Verfahren haben ausschließlich zwischen den einzelnen Zeitschritten die Last balanciert. Allerdings ist es möglich, auch während der Kraftberechnungsphase Arbeit neu zu verteilen, um maximale Effizienz zu erreichen. In diesem Fall wird ein Tool namens QUARK [28] genutzt, um die Arbeit dynamisch den Prozessen zuzuteilen. Durch sogenanntes *task stealing* werden Prozesse mit zu viel Last entlastet, wohingegen andere Prozesse nicht warten müssen. Die zugrundeliegende Technik von QUARK ist ein Datenfluss-Modell, bei dem die Aufteilung der Aufgaben wie auch bei graphbasiertem Partitionieren durch einen Graphen bestimmt wird. Die Knoten des Graphen sind dabei Aufgaben, die Kanten stellen die Abhängigkeiten von Daten dar. Da QUARK von Haus aus auf Kernels arbeitet, bietet sich diese Art der Aufteilung vor allem für die schnelle Multipol-Methode an, die Kernels nutzt, um die verschiedenen Entwicklungen zu berechnen. In dieser Arbeit wurde zwar der Barnes-Hut-Algorithmus nicht in seiner Kernelformulierung dargestellt, jedoch handelt es sich bei den Näherungen im Prinzip um Taylorentwicklungen, weshalb es kein Problem ist, eine solche Formulierung zu finden.

4 Barnes-Hut auf Shared Memory Architekturen

Es wurden nun alle Grundlagen erörtert, um die erste parallele Implementierung des Barnes-Hut-Algorithmus zu beschreiben. Im ersten Fall wird eine Shared-Memory-Architektur vorausgesetzt.

4.a Baumaufbau

Beim parallelen Baumaufbau fügen die verschiedenen Prozesse parallel die Partikel in den zu Beginn leeren Wurzelknoten ein, wie auch im sequentiellen Fall. Da der entstehende Baum von der Reihenfolge, in der die Partikel eingefügt werden, unabhängig ist, kann dabei ein freier Prozess also einfach ein beliebiges Partikel als nächstes einfügen. Eine Möglichkeit, den Baumaufbau parallel zu implementieren, ist in [12, 3.1.1.1] vorgestellt.
Um Zugriffsfehler zu vermeiden, die zum Beispiel auftreten könnten, wenn zwei Prozesse eine Zelle gleichzeitig verfeinern beziehungsweise überhaupt mit einem Partikel belegen wollen, werden Sperrvariablen eingeführt. Ändert

ein Prozess eine Zelle des Baumes, so sperrt er die Zelle, was es anderen Prozessen nicht mehr erlaubt, auf diese Zelle zuzugreifen. Sperren muss man allerdings nur die Blattzellen, da die inneren Zellen beim Einfügen zwar genutzt, aber nicht geändert werden. Deshalb ist es in diesem Fall ratsam, die Werte der Pseudopartikel gerade nicht beim Baumaufbau zu berechnen, sondern in einer eigenen Phase. Mehr Details zu diesem Vorgehen sind in [24] zu finden.
Die Notwendigkeit von Sperrvariablen kann man vermeiden, indem ein Prozess die Partikel anfangs in einen Baum geringer Tiefe einsortiert, wobei die Anzahl der Partikel pro Blatt nicht limitiert ist. Danach lässt man jeden Prozess eines dieser Blätter weiter verfeinern. Bei homogener Verteilung der Partikel ist dieses Verfahren natürlich deutlich effizienter als bei einer inhomogenen.
Auch Grafikkarten fallen in die Kategorie der Shared-Memory-Prozessoren. Hochstätter beschäftigt sich in seiner Arbeit [24] unter anderem mit dem Baumaufbau auf Grafikkarten und stellt Messungen zur Performance an. Einer der Kernpunkte seiner Implementierung ist dabei die Speicherung des Baumes als Struktur von Arrays ([24, 5.3]). Seiner Aussage nach lässt sich so eine Halbierung des Zeitbedarfs erreichen.

4.b Berechnung der Werte der Pseudopartikel

Die parallele Berechnung der Werte für die Pseudopartikel läuft prinzipiell genau so wie im sequentiellen Fall. Der einzige Unterschied ist hier, dass (so lange genug Operationen vorhanden sind) ein frei werdender Prozess die Werte des nächsten Pseudopartikels berechnet, dessen Zelle alle Pseudopartikel für seine Kindzellen schon kennt, da sie bereits berechnet wurden [12, 3.1.1.2]. Aufgrund der sich verjüngenden Struktur des Baumes ist es dabei jedoch nicht möglich, jeden Prozess komplett auszulasten. Dies ist einfach zu erkennen, wenn man die Baumwurzel betrachtet. Um die Masse und den Schwerpunkt der Wurzel zu berechnen, müssen die Werte aller anderen Zellen schon bekannt sein. Somit müssen in diesem Schritt alle bis auf einen Prozess warten. Wie viel Latenz entsteht, lässt sich anhand der folgenden Rechnung nachvollziehen. Es wird von einer homogenen Partikelverteilung ausgegangen. Dabei ist $l = \log_4 n$ die Anzahl der Baumlevel und die Dimension der physischen Domäne ist 2. Die Komplexität dieser Phase für p Prozesse ist:

$$\sum_{i=0}^{l-1} \left(\frac{4^i}{p} \right) = \sum_{i=\log_4 p}^{l-1} \left(\frac{4^i}{p} \right) + \sum_{i=1}^{\log_4 p} 1 \tag{3.1}$$

Dabei steht der erste Term der aufgeteilten Summe für den Teil der Berechnung, der komplett parallel ausgeführt werden kann, da für jede Prozess noch eine zu berechnende Zelle vorhanden ist. Der zweite Term hingegen

steht für den Teil, bei dem Prozesse warten müssen [10, 4]. Eine effiziente Implementierung dieser Phase auf Grafikkarten liefert [24, 5.7.2].

4.c Kraftberechnung

Hier zeigt sich der große Vorteil der Shared-Memory-Architektur. Da jeder Prozess den gesamten Baum kennt, kann hier, wie in der Phase des Baumaufbaus, ein freier Prozess die Kraft für das nächste, unbearbeitete Partikel wie im sequentiellen Fall berechnen. Sperrvariablen sind hier nicht nötig. Eine mögliche Implementierung ist, eine Liste mit allen Partikeln zu erzeugen, aus der die Prozesse immer das erste Element herausnehmen, aus der Liste löschen und dann die Kräfte für dieses Partikel berechnen. Sind die Berechnungen für dieses Partikel fertig, nimmt der Prozess das neue, momentan erste Element der Liste und fährt wie gerade beschrieben fort.

4.d Positionsupdate der Partikel

Auch hier gilt, wie in der Phase der Kraftberechnung, dass die Parallelisierung kein Problem darstellt.

5 Barnes-Hut auf Distributed Memory Architekturen

Auf Architekturen mit verteiltem Speicher ist die Implementierung des Barnes-Hut-Algorithmus deutlich schwieriger [18], [5], [8]. Gerade die Phase der Kraftberechnung (von der in Kapitel 2 gezeigt wurde, dass sie am aufwändigsten ist), wird hier im Fokus stehen. Die vorgestellte Implementierung ist der sehr ähnlich, die im Zuge dieser Arbeit erstellt wurde.

5.a Initialisierung

Vor der Schleife über die Zeitschritte wird in dieser Implementierung ein Initialisierungsschritt gemacht. Ein einzelner Prozess (im Folgenden Prozess 0) baut dabei eine globalen Baum der Partikel auf, den er dann geschickt auf alle Prozesse aufteilt. Dies geschieht zum Beispiel, indem man anfangs von konstanter und gleicher Last aller Partikel ausgeht und dann, einer raumfüllenden Kurve folgend, für jeden Prozess die gleiche Anzahl an Partikeln abzählt. Die Nutzung der raumfüllenden Kurven sorgt schon hier für gute physische Lokalität.
Als nächstes berechnet Prozess 0 für jeden Prozess die minimale Menge an Baumknoten, deren Nachkommen alle seine Partikel enthalten. Dies ist gleichzeitig die Menge an Knoten, die global bekannt sein wird. Im Folgenden werden diese Knoten Indexknoten genannt. Indexknoten können Blätter oder innere Knoten sein.

Nach ihrer Berechnung werden die Indexknoten in einem Broadcast an alle Prozesse geschickt, wobei jeder Indexknoten auch den ihm zugewiesenen Prozess kennt. Zusätzlich bekommt jeder Prozess natürlich noch die Partikeldaten der Partikel zugesendet, für deren Berechnung er zuständig ist. Das sind pro Partikel mindestens ein Wert für die Masse und zwei Koordinaten. Jeder Prozess berechnet nun separat die Werte der Pseudopartikel oberhalb des Levels der Indexknoten. Alternativ kann dies auch Prozess 0 tun und die oberen Baumknoten dann mit den Indexknoten versenden. Dieser Ansatz wird jedoch in der vorliegenden Arbeit nicht verfolgt.
Nachdem jeder Prozess seine Partikel in seinen Baum eingefügt hat, hat also nun jeder Prozess einen Baum bestehend aus einem global bekannten, oberen Teil und einem lokalen, unteren Teil, den nur er kennt und der bis auf die Granularität von Einzelpartikeln hinunterreicht. Nun kann man mit dem eigentlichen Algorithmus beginnen. Weitere Informationen und Anmerkungen zum Aufbau der Bäume sind in [8] und [18] zu finden.

Anmerkung

Prinzipiell muss man für die Menge der Indexknoten nicht fordern, dass sie minimal ist. Im Gegenteil müssen natürlich deutlich weniger Knoten kommuniziert werden, wenn der global bekannte Teil der Knoten größer ist. Auf der anderen Seite belegt dieses Vorgehen natürlich auch mehr Speicher. Hier gilt es, einen guten Kompromiss zu finden.

5.b Kraftberechnung

In der Phase der Kraftberechnung gibt es eine fundamentale Herausforderung: Kein Baum kennt alle Partikel, benötigt aber viele von ihnen, um die Kräfte für seine Partikel zu berechnen. Um die benötigten Daten bereitzustellen, gibt es verschiedene Ansätze, die im Folgenden diskutiert werden. Die Methode, die in der beiliegenden Arbeit verwendet wird, wird Data-Shipping genannt und als zweites vorgestellt.

Latency Hiding Scheme

Bei diesem Vorgehen berechnet jeder Prozess so weit er mit dem ihm bekannten Baum kann. Dabei speichert er sich die Knoten, für die er eine Verfeinerung benötigt und überspringt die Kraftberechnung mit eben diesen (da er nicht die nötigen Daten kennt). Dann folgt ein Kommunikationsschritt, bei dem jeder Prozess die Knoten empfängt, die er zur weiteren Kraftberechnung benötigt, und die Knoten versendet, die andere Prozesse von ihm benötigen. Eine mögliche Implementierung könnte wie folgt aussehen: Jeder Prozess führt während der Kraftberechnungsphase Buch, welche Knoten er verfeinert benötigt. Im anschließenden Kommunikationsschritt schickt jeder Prozess 0 jedem anderen Prozess i erst die Anzahl der der Knoten, die 0 von

i verfeinert benötigt. Anschließend schickt 0 die Koordinaten der Knoten an i. Dieser sucht alle Kinder der Knoten auf der Liste und schickt diese an 0 zurück.
Die beiden obigen Schritte werden so lange wiederholt, bis die Kraftberechnung abgeschlossen ist.
Effizient wird dieses Verfahren vor allem dadurch, dass die Prozesse in Arbeiter- und Kommunikationsthreads unterteilt werden. So kann ein Arbeiterthread mit der Kraftberechnung auf Knoten, die er bereits kennt, fortfahren, während ein Kommunikationsthread die benötigten Knoten kommuniziert. Die Unterteilung der Threads ist dabei dynamisch. So wird ein Kommunikationsthread, der nichts mehr zu tun hat automatisch wieder zum Arbeiterthread. Bekommt er eine neue Anfrage, wird er wieder zum Kommunikationsthread. In [5] wird dieses Verfahren und seine Eigenschaften genauer beschrieben.

Data Shipping

Eine etwas weiter verbreitete Methode wird oft als Data Shipping bezeichnet. Wie der Name schon andeutet, werden bei diesem Verfahren, wie auch bei dem zuerst vorgestellten, die Daten der benötigten Knoten verschickt. Somit kann dann jeder Prozess die Kräfte für seine Partikel selbst berechnen. Dies wird oft als Owner-Computes-Prinzip bezeichnet.
Der Hauptunterschied zum Latency Hiding Scheme liegt in der Trennung der Phasen der Kommunikation und Kraftberechnung: Vor jeder Kraftberechnung müssen also alle benötigten Knoten kommuniziert werden. Die Herausforderung ist hier, dass ein Prozess vor der Kraftberechnung gar nicht wissen kann, wie weit er den Knoten eines anderen Prozesses verfeinert haben muss, da er die genaue Struktur der fremden Bäume nicht kennt. Aus diesem Grund geht man anders herum vor: Jeder Prozess i berechnet, wie weit die anderen Prozesse seinen Baum verfeinert benötigen. Dies ist möglich, da er die Indexknoten der anderen Prozesse kennt. Da alle Partikel eines Prozesses innerhalb der Grenzen seiner Indexknoten liegen, lässt sich so die Distanz zwischen Knoten von Prozess i und Partikeln von anderen Prozessen mit den Grenzen der Indexknoten des jeweiligen, anderen Prozesses abschätzen. Vergleiche dazu Abbildung 3.11.

Nachdem das Verschicken aller benötigten Knoten vollständig abgeschlossen ist, kann nun die Kraft wie im sequentiellen Fall von jedem Prozess für seine Partikel übernommen werden, da er ja alle nötigen Daten dafür hat. In [8], [7, 3.3] und [18] wird dieses Verfahren angewandt.

Function Shipping

Eine grundsätzlich andere Methode ist hingegen das sogenannte Function Shipping. In [8] wird genauer darauf eingegangen, hier soll nur die Idee

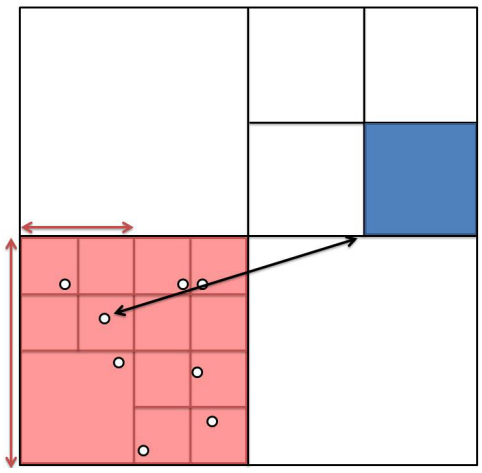

Abbildung 3.11: In dieser Abbildung wird dargestellt, wie Prozess 0, dessen Indexknoten rot hinterlegt dargestellt, werden überprüft, welche seiner Zellen Prozess 1 (blau) zur Berechnung der Kräfte auf die "blauen"Partikel benötigt. In dem gezeichneten Beispiel gibt es für jeden Prozess genau einen Indexknoten, der farblich hinterlegt ist. Für das θ-Kriterium werden der Abstand von Partikel zu Zelle und der Durchmesser der Zelle benötigt. Dies ist exemplarisch als schwarzer beziehungsweise roter Doppelpfeil dargestellt.

angerissen werden. Das zugrundeliegende Prinzip besteht darin, nicht die Knoten der Bäume zu verschicken, sondern die Partikeldaten. Der empfangende Prozess berechnet dann den Einfluss seines Baumes auf die Partikel. Auch hier gibt es verschiedene Ansätze, die Kommunikation zu organisieren. Entweder macht man es wie im Latency Hiding Scheme und kommuniziert immer, wenn es in der Kraftberechnung notwendig ist. Hierbei bietet es sich trotzdem an, Partikel zu bündeln, um nicht zu viele Einzelkommunikationen zu tätigen. Oder aber man kommuniziert jedem Prozess alle Partikelpositionen vor der Kraftberechnung und jeder Prozess berechnet den Einfluss seines Teilbaumes auf alle Partikel. Danach werden die berechneten Kräfte an den für das Partikel zuständigen Prozess zurückgesendet.
Dieses Verfahren verletzt das Owner-Computes-Prinzip, da auch fremde Prozessen Kräfte berechnen. Gegebenenfalls sollte man die oben beschriebenen Metriken überdenken, da die Last eines Partikels nicht mehr nur von dem ihm zugewiesen Prozess getragen wird. Eine Möglichkeit wäre in diesem Fall, die Last mit Baumknoten zu assoziieren. Metriken 2,3 und 4 lassen sich dann sehr einfach adaptieren. Metrik 2 assoziiert jeden Baumknoten mit einer gleichen, konstanten Last, Metrik 3 assoziiert die Last mit der Tiefe des Knotens im Baum und Metrik 4 zählt für jedem Baumknoten die Interaktionen.

5.c Update der Partikelpositionen

Nachdem die Kräfte für alle Partikel erfolgreich berechnet wurden, wird berechnet, wie sich die Partikel als Reaktion auf die Krafteinwirkung bewegen. Als Resultat dieser Bewegung nimmt jedes Partikel eine neue Position ein, die dann als Ausgangspunkt für den nächsten Simulationsschritt dient. Dies läuft analog zu dem Update für Shared-Memory-Architekturen (vgl. Kapitel 3.4d).

5.d Inkrementelles Baumupdate

Dieser Schritt läuft prinzipiell so, wie in Kapitel 2 beschrieben. Der Unterschied ist nur, dass die Partikel gegebenenfalls neu zugewiesen werden müssen, falls sie die Domäne ihres Prozesss verlassen. Dazu kommuniziert jeder Prozess nach dem Update der Partikelpositionen diejenigen Partikel, die seinen Berechnungsbereich verlassen, an den Prozess, in dessen Berechnungsbereich sie eintreten. Welcher Prozess der Empfänger ist, lässt sich über die Indexknoten feststellen, die global bekannt sind. Der Empfänger sortiert die neuen Partikel in seinen Baum ein, wie is in Kapitel 2.1.1a beschrieben ist.
Auch hier muss nun natürlich wieder überprüft werden, dass kein Blatt zu viele Partikel besitzt und auch kein innerer Knoten unnötig verfeinert ist. Anschließend muss nur die Last neu verteilt werden und man kann mit der Phase der Kraftberechnung für den nächsten Schritt weiter machen. [7, 3.2.3] beschreibt das inkrementelle Baumupdate auf Architekturen verteilten Speichers.

6 Das schnelle Multipolverfahren auf Shared-Memory-Architekturen

Die Phasen des Baumaufbaus und Positionsupdates unterscheiden sich für das schnelle Multipolverfahren nicht grundlegend von denen, die der Barnes-Hut-Algorithmus auf Shared-Memory-Architekturen nutzt. Daher wird hier insbesondere auf den Auf- und Abwärts-Schritt eingegangen, der die Kraftberechnung darstellt.

6.a Aufwärts-Schritt

Mit dem Aufwärts-Schritt werden die Multipolentwicklungen für alle Zellen des Baumes berechnet. Diese Berechnung beginnt in den Blättern des Baumes. Auf Rechnern mit verteiltem Speicher, bei der jeder Prozess alle Daten kennt, werden zuerst für alle Blätter die Multipolentwicklungen berechnet und danach von unten nach oben durch den Baum die Entwicklungen aller anderen Zellen. Welcher Prozess dabei welche Berechnung übernimmt,

spielt keine Rolle. Für eine Berechnung müssen nur die Entwicklungen der Kinderknoten bekannt sein.
Der Aufwärts-Schritt ist jedoch nicht komplett parallelisierbar, was wieder an der Wurzel erkennbar ist: Um die Entwicklung der Wurzel zu berechnen, müssen alle anderen Entwicklungen bereits bekannt sein (vgl. 3.4b). Die parallele Effizienz ist vergleichbar zu der in Kapitel 4.b hergeleiteten, da auch hier der Flaschenhals dann auftritt, wenn es weniger zu berechnende Knoten als Prozesse gibt [10].

6.b Abwärts-Schritt

Der Abwärts-Schritt berechnet nun für alle Zellen, mit der Wurzel beginnend, die Taylorentwicklungen verschiedener Zellen auf andere. Ist man im Blattlevel angekommen, kennt jede Zelle das auf sie wirkende Potenzial. Dieser Schritt läuft prinzipiell genau anders herum als der Aufwärts-Schritt, weshalb er grob die gleiche parallele Effizienz besitzt.

7 Das schnelle Multipolverfahren auf Distributed-Memory-Architekturen

Nachdem die Parallelisierung der schnellen Multipol-Methode für geteilten Speicher diskutiert ist, widmet sich diese Arbeit der Implementierug auf Architekturen verteilten Speichers.

7.a Baumaufbau

Ein einzelner Prozess (Prozess p) baut den kompletten Baum aller Partikel auf. Nachdem der Baum vollständig ist, errechnet Prozess p für jede Zelle die Last. Dazu wird die Metrik aus 3.1.b verwendet. Anhand der Daten dieser Metrik werden dann (zum Beispiel mit raumfüllenden Kurven) alle Zellen auf die Prozesse verteilt, so dass schlussendlich jeder Prozess die gleiche Last erhält. Wichtig ist hierbei, dass nicht nur die Blattzellen verteilt werden müssen, sondern auch die inneren Zellen, da auch diese eine Last mit sich bringen. Unser bisheriges Vorgehen mit raumfüllenden Kurven, wie es für Costzones in 3.3.a beschrieben wird, bezieht sich ausschließlich auf Blattzellen. Eine Adaption, die auch innere Zellen berücksichtigt, ergibt sich aus dem rekursiven Aufbau der Kurven. Jede Zelle hat hierzu nicht nur die eigene Last gespeichert, sondern auch die Gesamtlast aller Nachfahren. Beginnend mit der Wurzel werden nun die Zellen auf die Prozesse verteilt. Ist dabei die Gesamtlast der Zelle zu groß für den aktuellen Prozess, bekommt dieser nur die Zelle zugesprochen und es wird rekursiv mit den Kindzellen fortgefahren. Ist auch die Einzellast der Zelle so groß, dass der Prozess in Summe zu viel Last hätte, so wird mit dem nächsten Prozess fortgefahren. Anschließend wird in einem Kommunikationsschritt jedem Prozess mitgeteilt, welche

Zellen ihm zugewiesen wurden. Da dieser Schritt nur einen verschwindend kleinen Anteil der Gesamtrechenzeit beansprucht, ist es verkraftbar, hier sequentiell zu arbeiten.

7.b Kraftberechnung

Da die meisten modernen Großrechner Architekturen verteilten Speichers besitzen, spielt insbesondere die Parallelisierung auf diesen Architekturen in der aktuellen Forschung eine große Rolle. Aus diesem Grund wird sich diese Arbeit vor allem auf eine vorangegangene Arbeit stützen, die auch auf moderne Probleme, wie zum Beispiel fehlerhafte Berechnungen [13, 2.3], Rücksicht nimmt. Bisher wurden viele dieser Probleme nicht berücksichtigt. Die Quellen, die für das folgende Kapitel zu Rate gezogen wurde, [13], [18], [10] und [11].
Bisher wurde bei der Verteilung der Last davon ausgegangen, dass jede Berechnung korrekt ausgeführt wird. Daraus resultiert, dass kein Prozess größere Wartezeiten hat, wenn die Anzahl der Operationen vor der Partikelverteilung korrekt geschätzt und die Last gut verteilt wurde. In der Praxis werden jedoch nicht alle Operationen korrekt berechnet. Das heißt man kann von vorn herein nicht wirklich abschätzen, wie viel tatsächliche Arbeit ein Prozess zu bewältigen hat. Somit fallen aus den in Kapitel 5 vorgestellten Methoden zur Kommunikation zwei der drei Methoden (nämlich die als *Data Shipping* und *Function Shipping* bezeichneten) für moderne Anwendungen weg, da diese Methoden von einer korrekten Schätzung der Last vor der eigentlichen Kraftberechnung ausgehen. Natürlich ist es auch für das *Latency Hiding Scheme* vorteilhaft, wenn die Last zu Beginn korrekt geschätzt wird, jedoch ist dieses Verfahren durch die permanente Kommunikation auch während der Kraftberechnung in der Lage, Imbalancen auszugleichen.
Natürlich gibt es auch viele Arbeiten, in denen *Data Shipping* und *Function Shipping* verwendet werden. In [18] werden sie zum Beispiel genutzt. In dieser Arbeit wird jedoch wie schon angekündigt vor allem auf die praktische Adaption auf modernen Großrechnern eingegangen, da die gerade angeführten fehlerignoranten Methoden nach [13] nicht beliebig skalierbar sind.
Die Kommunikation wird also während der Kraftberechnung getätigt, um auch spontan auftretende Überlasten einiger Prozessoren abzufedern. Dieses Vorgehen wird in der Literatur oft als *task stealing* bezeichnet. Der größte Vorteil ist wie schon erwähnt, dass die Granularität der Lastbalancierungs-Schritte hier beliebig klein gewählt werden kann. Verfahren, die nicht auf task stealing setzten, können maximal einen Balancierungs-Schritt pro Zeitschritt durchführen. Da nach [13] einer dieser Zeitschritte jedoch über 100 s dauern kann, ist dies zu selten.
Für die eigentliche Parallelisierung wird ein Tool namens QUARK genutzt. Es sorgt dafür, dass wartende Prozesse Prozessen mit Überlast Aufgaben

abnehmen, während gleichzeitig der gesamte Datenstrom minimiert wird. Wie in 3.3.b beschrieben, wird hierzu ein Graph genutzt. Die Knoten des Graphen entsprechen den zu tätigen Berechnungen, die Kanten stehen für Daten-Abhängigkeiten. Für nähere Informationen zu QUARK ist [28] zu konsultieren. In [13] sind Performance-Messungen zu dem hier vorgestellten Verfahren zu finden.

Kapitel 4

Messergebnisse

In diesem Kapitel werden die gemessenen Ergebnisse der hiesigen Implementierung dargelegt. Dabei wird anfangs auf die Motivation dieser Arbeit eingegangen. Im Zweiten Teil werden die verschiedenen Techniken zur Lastquantifizierung, die in dieser Arbeit Erwähnung finden, verglichen. Alle Messungen sind auf physischen Kernen durchgeführt. Die Anzahl der Prozesse, für die gemessen wurde, geht von 1 bis 64 in Potenzen der 2. Dabei wurden immer 10.000 Partikel pro Prozess simuliert: Es wurde das schwache Skalierungsverhalten gemessen. Die Verteilung der Partikel ist im ersten Szenario gleichmäßig und im zweiten der einer Galaxie ähnlich. Alle Messwerte sind über zehn Zeitschritte gemittelt.

1 Diskussion der Ergebnisse

Die grundlegendste Art, Last auf Prozessoren im Rahmen von Molekülsimulationen zu verteilen, wird in Kapitel 3.1.a als Metrik 1 bezeichnet. Dabei weist man jedem zur Verfügung stehenden ein glich großes Stück der physischen Domäne zu. In Abbildung 4.1 sind die gemessenen Zeiten für beide Szenarien zu sehen.

Die Steigung der Kurven gibt das allgemeine Skalierungsverhalten an. Ideal wäre hierbei eine konstante Funktion, da dann die parallele Effizienz bei 1 liegt. Im Falle des gemessenen Barnes-Hut-Algorithmus ist dies jedoch utopisch, da der Algorithmus eine Komplexität von $\mathcal{O}(n \log n)$ in der Anzahl der Partikel n hat und hier das schwache Skalierungsverhalten gemessen wird: die Anzahl der Partikel also mit der Anzahl der Prozesse steigt. Somit ist die leichte Steigung auch Ausdruck der Komplexität des Algorihmus.
Die wichtigste Aussage dieser Bilder liegt in der Lage der Minima und Maxima: Ist ein Verfahren perfekt balanciert, so benötigt jeder Prozess gleich viel Zeit. In diesem Fall liegen also Durchschnittszeit, Maximalzeit und Minimalzeit auf dem gleichen Wert. Umgekehrt zeigt eine große Diskrepanz dieser Werte, dass das Verfahren extrem unbalanciert ist. Auf Abbildung 4.1 bezo-

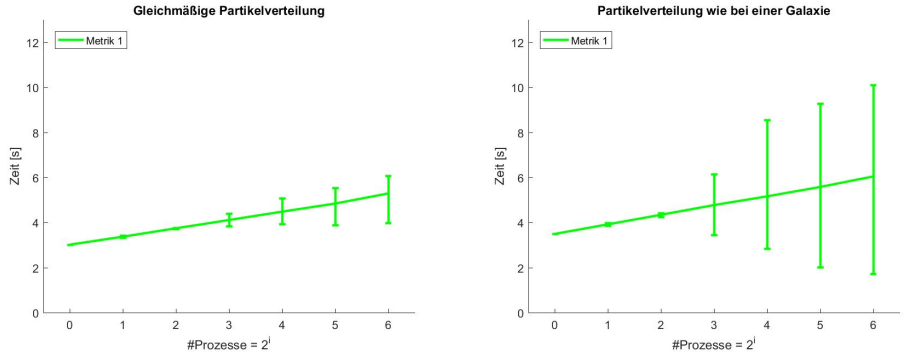

Abbildung 4.1: In diesen Abbildungen ist die zur Kraftberechnung benötigte Zeit in Abhängigkeit zur Anzahl der Prozesse für beide Szenarien dargestellt. Die Zeit-Achse ist linear skaliert, die Achse für die Anzahl der Prozesse logarithmisch zur Basis zwei. Dabei zeigt die durchgängige, grüne Linie die über alle Prozesse gemittelte Berechnungszeit an. Das obere beziehungsweise untere Ende der Balken gibt dabei die maximale beziehungsweise minimale Berechnungszeit aller Prozesse an.

gen stellt man fest, dass Metrik 1 für gleichmäßig verteilte Partikel die Last so gut aufteilt, dass die Differenz in den Zeiten der verschiedenen Prozesse ungefähr 2 s beträgt. Dies ist eine vergleichsweise gute Balance, wenn man das gleiche Diagramm für inhomogen verteilte Partikel betrachtet: Hier liegen die Berechnungszeiten zwischen $1,5$ $sund10,5$ s. Wie in Kapitel 3 schon angesprochen, hat dieses Balancierungsverfahren für inhomogene Partikelverteilungen nur eine sehr schlechte Performance. Dies motiviert die in Kapitel 3 folgenden Überlegungen zur Lastpartitionierung.
Am Ende dieser Überlegungen standen verschiedene Verfahren und Metriken, mit denen auch für inhomogene Anwendungen gute Lastbalancen entstehen sollten. In Abbildung 4.2 werden die Ergebnisse des Costzones- und ORB-Verfahrens mit den in Abbildung 4.1 erreichten Ergebnissen verglichen. Für beide Verfahren wurde dabei Metrik 4 zur Schätzung der Last genutzt.

Es ist links gut zu erkennen, dass beide verbesserten Verfahren für einen bis 32 Prozesse eine deutlich bessere Performance haben: Mittelwert, Minimum und Maximum liegen sehr nah beieinander. Das Verhalten bei 64 Prozessen fällt aus diesem Schema. Für das ORB-Verfahren ist dabei der Mittelwert sehr nah am Minimum, es gibt jedoch mindestens einen Prozess, der deutlich zu viel Last bekommen hat, was sich in dem hohen Maximum äußert. Für das Costzones-Verfahren hingegen verhält sich selbst der Mittelwert sehr untypisch.
Für den homogenen Fall ließen sich bereits gute Ergebnisse der beiden verbesserten Verfahren nachweisen, der wirkliche Vorteil dieser Methoden of-

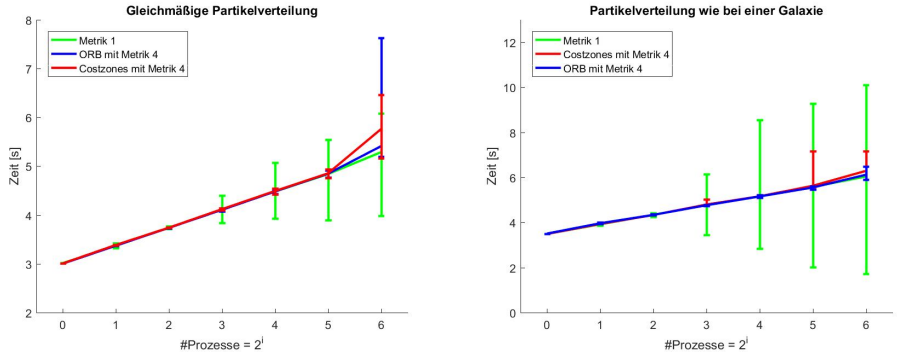

Abbildung 4.2: Die Abbildung ist Abbildung 4.1 ähnlich. Es werden dabei die gemessenen Zeiten für die Verfahren ORB und Costzones – jeweils mit Metrik 4 zur Schätzung der Last – hinzugefügt. Außerdem ist die Skala der Zeit-Achse im linken Diagramm zur besseren Übersicht adaptiert.

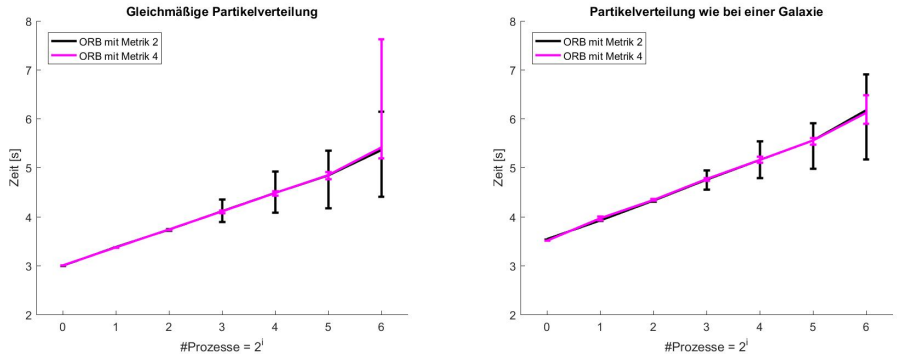

Abbildung 4.3: Zwei Abbildungen, die die Performance des ORB-Verfahrens mit verschiedenen Metriken zeigen.

fenbart sich allerdings erst für inhomogene Partikelverteilungen. Das ORB-Verfahren zeigt hierbei durchweg gute Lastbalance, die insbesondere um Längen besser ist als die Balance, die durch gleichmäßiges Aufteilen der physischen Domäne erreicht wurde. Das Costzones-Verfahren hingegen hat auch hier ein paar Ausreißer, führt jedoch trotzdem zu besserer Lastbalance, als Metrik 1.
Abbildung 4.3 zeigt zwei der vier Metriken im Vergleich. Zum einen ist die Metrik zu sehen, die für alle Partikel eine gleiche, konstante Last annimmt und zum anderen Metrik 4, die alle Interaktionen zählt.

In beiden Diagrammen ist gut zu erkennen, dass Metrik 4 deutlich bessere Ergebnisse liefert, als Metrik 2. Im linken Bild gibt es jedoch wieder einen untypischen Wert für 64 Prozesse. Dies könnte bedeuten, dass der Fehler in Abbildung 4.2 auf die Metrik zurückzuführen ist.
Grundsätzlich bestätigen die gemessenen Werte die in Kapitel 3 aufgestell-

ten Hypothesen: Für gleichmäßige Partikelverteilungen liefert Metrik 1 eine simple und recht performante Lastbalance. Gerade bei inhomogenen Verteilungen lohnt es sich jedoch, mehr Zeit in eine gute Lastbalancierungsmethode zu investieren. Diese zeigen auch im Allgemeinen eine deutlich bessere Aufteilung der Last. Insbesondere die Wahl der Lastmetrik spielt dabei eine entscheidende Rolle.

Kapitel 5

Zusammenfassung und Ausblick

Die grundlegende Motivation dieser Arbeit ist die Lastbalancierung für Molekülsimulationen mit langreichweitigen Kräften. Dazu wurde zuerst die direkte Methode der Kraftberechnung vorgestellt, die jedoch mit einer Komplexität von $\mathcal{O}(n^2)$ in der Zeit und für die Anzahl der Partikel n zu schlecht skaliert, als dass man sie für große Simulationen nutzen könnte. Beim genaueren Betrachten der Potenziale ergibt sich für kurzreichweitige Potenziale eine Lösung, die auf der Idee beruht, das Potenzial gewissermaßen *abzuschneiden*. Für langreichweitige Potenziale ist dabei der Fehler zu groß. Hier lässt sich durch Nutzung von Partikel-Zell-Wechselwirkungen die Anzahl der zu berechnenden Interaktionen deutlich senken. Viele Ansätze, diese Partikel-Zell-Wechselwirkungen in der Praxis zu nutzen, beruhen auf Baumstrukturen. Insbesondere der Barnes-Hut-Algorithmus und die schnelle Multipol-Methode sind bekannte Algorithmen, die auf dieser Idee gründen. Will man diese Algorithmen parallel implementieren, stellt sich vor allem die Frage, wie man den globalen Baum aller Partikel effizient auf die Prozesse verteilen kann, um eine gute Lastbalance zu erreichen. Die Basis für diese Aufgabe ist ein Maß der Last. In dieser Arbeit wurden verschiedene Lastmetriken vorgestellt und hergeleitet, die unterschiedlich komplex in ihrer Berechnung sind. Die Messungen aus Kapitel 4 haben gezeigt, dass sich komplexe Metriken trotzdem auszahlen! Offen ist noch die Frage, wie man die Partikel dann tatsächlich auf die Prozesse verteilt. Dabei wurden verschiedene Partitionierungsmethoden wie graphbasiertes Partitionieren, ORB und Costzones vorgestellt. Diese lassen sich mit einigen Änderungen auch auf die schnelle Multipol-Methode anwenden.
Durch das rasante Wachstum der modernen Rechner wird das Thema der Parallelisierung wird in Zukunft eine immer größere Rolle spielen. Bezogen auf die hiesige Arbeit wäre ein möglicher nächster Schritte zum Beispiel die Parallelisierung über Zeitschritte hinweg. Bisher musste jeder Prozess war-

ten, bis alle anderen Prozesse mit dem jeweiligen Zeitschritt abgeschlossen haben, um mit dem nächsten Zeitschritt zu beginnen. Allerdings führt dieses Vorgehen dazu, dass die Umverteilung der Partikel – sowohl bei statischem als auch bei dynamischem Partitionieren – nicht ohne weiteres nach jedem Zeitschritt vorgenommen werden kann.
Eine weitere Möglichkeit, den Algorithmus zu verbessern, ist eine feinere Granularität in der Partitionierung, die auch innerhalb eines Zeitschritts und nicht nur zwischen Zeitschritten möglich ist. Denkbar wäre auch, nicht alle Partikel zu Beginn zu verteilen, sondern einen neutralen Pool an Partikeln zu haben, für den theoretisch jeder Prozess Berechnungen tätigen können muss. Prozesse, die bereits fertig sind mit ihrer Arbeit, kümmern sich dann um diesen Pool, während die anderen Prozesse ganz normal arbeiten. Insgesamt ist diese Arbeit nicht erschöpfend, bietet aber eine gute Grundlage an Methoden, die eine erhebliche Effizienzsteigerung – gerade auf parallelen Architekturen – mit sich bringt.

Literaturverzeichnis

[1] Dan Sloughter. The two-body problem. 2000.

[2] J. Barnes; P. Hut. A hierarchical O(N log N) force-calculation algorithm. December 1986.

[3] A. Appel. An efficient program for many-body simulation. 1985.

[4] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, December 1987.

[5] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon. A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations. *Computer Physics Communications*, 183:880–889, April 2012.

[6] J. Board; Z.Hakura; W. Elliott; W. Rankin. 1995.

[7] Pangfeng Liu. The parallel implementation of n-body algorithms. 1994.

[8] Ahmed Sameh Ananth Grama, Vipin Kumar. Scalable parallel formulations of the barnes–hut method for n-body simulations. 1998.

[9] Ahmed Sameh Ananth Y. Grama, Vipin Kumar. Scalable parallel formulations of the barnes-hut method for n-body simulations.

[10] W.D.Gropp L. Greengard. A parallel version of the fast multipole method. 1989.

[11] Jaswinder Pal Singh, Chris Holt, John L. Hennessy, and Anoop Gupta. A parallel adaptive fast multipole method. In *Proceedings Supercomputing '93, Portland, Oregon, USA, November 15-19, 1993*, pages 54–65, 1993.

[12] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta, and John L. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal Of Parallel and Distributed Computing*, 27:118–141, 1995.

[13] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *CoRR*, abs/1203.0889, 2012.

[14] Shang-Hua Teng. Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation. 1998.

[15] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM.

[16] Warren M. Lam; Jerome M. Shapiro. A class of fast algorithms for peano-hilbert space-filling curve.

[17] C. Anderson. An implementation of the fast multipole method without multipoles. 1991.

[18] Zumbusch Caglar Griebel, Knapek. *Numerische Simulation in der Molerküldynamik*. 2004.

[19] Ebisuzaki Sugimoto Makino, Taiji. Grape 4: a one-tflops special-purpose computer for astrophysical n-body problem.

[20] Miriam Mehl. Grundlagen des wissenschaftlichen rechnens - lecture notes. 2015.

[21] *The Linked Cell Method for Short-Range Potentials*, pages 37–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[22] Dirk Pflüger. Numerische und stochastische grundlagen. 2016.

[23] Klaas Esselink. The order of appels algorithm. 1992.

[24] Hendrik Hochstetter. Effiziente parallele implementierung von hierarchischen n-body-algorithmen auf multicore-systemen mit gpu-beschleunigung. 2012.

[25] V.Rokhlin J.Carrier, L.Greengard. A fast algorithm for particle simulations. *Journal of Computational Physics*, 135(2):280 – 292, 1997.

[26] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997.

[27] A.Gupta J.Hennessy, M.Heinrich. Cache-coherent distributed shared memory: perspectives on its development and future challenges. 1999.

[28] J. Kurzak A. YarKhan and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. 2011.

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.