Parallelisierung von MD-Simulationen auf Mehrkernarchitekturen

Bachelorarbeit

am Fachbereich Informatik
der Johannes-Gutenberg-Universität in Mainz

vorgelegt von

Christian Himmelsbach
geboren in Weiterstadt

Mainz 2009

Hiermit erkläre ich, **Christian Himmelsbach**, diese Arbeit selbst verfasst zu haben und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

1. Zusammenfassung

Diese Bachelorarbeit behandelt die Parallelisierung von MD-Simulationen auf Mehrkernarchitekturen. Die Frage im Speziellen ist, ob sich der Cell-Chip, oder ähnliche Architekturen, vorteilhaft bei MD-Simulationen einsetzen lassen. Konkret stellt sich hier die Frage, ob eine Performance-Verbesserung erzielt werden kann und wie groß der Geschwindigkeitszuwachs ist.

Diese Arbeit behandelt zunächst die technischen Gegebenheiten, um darzulegen auf welcher Hard- und Software die Resultate basieren.
Im nächsten Schritt werden alle benötigten physikalischen Grundlagen erklärt, die zum grundsätzlichen Verständnis benötigt werden. Zudem wird erklärt, was eine MD-Simulation ist und welche grundlegenden algorithmischen Werkzeuge bei der Umsetzung einer MD-Simulation benötigt werden.
Darauf folgend werden die verschiedenen Ansätze erläutert, eine MD-Simulation auf eine Cell-Chip-Architektur zu übertragen. Zudem geben Zeitmessungen einen Überblick über das Laufzeit-Verhalten der unterschiedlichen Methoden.
Im Vergleich werden die verschiedenen Verfahren einander gegenübergestellt.
Das Fazit beschreibt die Folgerung aus dem Vergleich.
Zuletzt gibt es noch einen Ausblick, der auf mögliche algorithmische Verbesserungen gegenüber der in dieser Arbeit vorgestellten Herangehensweisen verweist, die in zukünftigen Arbeiten behandelt werden können.

2. Technische Gegebenheiten

Diese Arbeit und deren Resultate basieren auf einem Cell-Chip, wie er in der von Sony entwickelten Spielekonsole „PlayStation 3“ verbaut ist. Entwickelt wurde der Cell-Chip von den drei Firmen Sony, Toshiba und IBM. Der Verbund dieser drei Firmen wird „STI“ genannt.
Die Anwendungsbereiche des Cell lassen sich grob in drei Gebiete einteilen, die den drei Firmen, die an der Entwicklung des Cell-Prozessors beteiligt waren, zugeordnet werden können. Toshiba war von Beginn an daran interessiert, den Cell-Prozessor zur Echtzeitkodierung und Dekodierung von Videosignalen in Bildschirmwiedergabegeräten wie Fernsehern einzusetzen. IBM setzt den Prozessor in Hochleistungsrechnern ein, wobei auch mehrere Cell-Prozessoren miteinander verknüpft arbeiten. Sony zielte auf eine Recheneinheit ab, die in der Spielekonsole „PlayStation 3“ eingesetzt werden sollte.

Im Folgenden werden die wichtigsten Angaben zur Hardware und Software im Zusammenhang dieser Arbeit erläutert. Diese Angaben basieren auf dem „Cell Broadband Engine Programming Handbook“ [IBM07]. Aus diesem Dokument können weitere Details entnommen werden.

2.1 Hardware

2.1.1 Cell Broadband Engine

Im ursprünglichen Konzept des Cell (Abb. 2.1) besitzt dieser einen dualen Hauptkern sowie acht kleine Rechenkerne, eine Arbeitsspeicher-Schnittstelle sowie eine flexible System-Schnittstelle, die aus zwei I/O Schnittstellen besteht. Des Weiteren existiert ein Daten-Bus, der alle diese Komponenten verbindet. So können die Kerne zum Beispiel mit dem Arbeitsspeicher oder untereinander kommunizieren.

Bezeichnungen:
Der Cell-Chip wird Cell BE oder CBE (Cell Broadband Engine) abgekürzt. Es ist aber auch gelegentlich von der CBEA (Cell Broadband Engine Architecture) die Rede. Der Hauptkern wird PPE (Power Processor Element) genannt. Jeder der acht kleinen Rechenkerne wird als SPE (Synergistic Processing Element) bezeichnet. Die Schnittstelle zum Arbeitspeicher wird MIC (Memory Interface Controller) abgekürzt. Die System-Schnittstelle wird mit BEI (Cell Broadband Engine Interface) abgekürzt. EIB (Element Interconnect Bus) ist die Bezeichnung des Daten- und Befehls-Busses, der alle Elemente auf dem CBE verbindet.

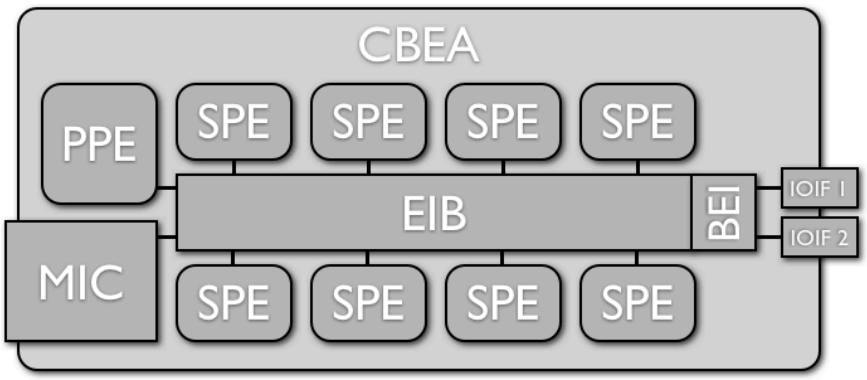

Abb. 2.1 - CBEA Layout

Power Processor Element (PPE):
Der Hauptkern ist ein 64 Bit „Power Prozessor“ mit einer SIMD Altivec-Einheit und einer Taktung von 3,2 GHz. Er besitzt einen L1-Cache von 32 KB, wie einen L2-Cache der Größe 512 KB. Dieser Hauptkern ist in der Lage ein Betriebssystem, wie z.B. Linux, auszuführen und dient in der Regel als Kontrolleinheit für die SPEs. Der Hauptkern verteilt also üblicherweise die anstehenden Aufgaben an die SPEs, empfängt die Ergebnisse und setzt diese zusammen.

Synergistic Processing Element (SPE):
Die Synergistic Processing Elements sind RISC-Kerne, also Recheneinheiten mit eingeschränktem Befehlsumfang. Die Taktung dieser Kerne ist mit 3,2 GHz gleich der Taktung des Hauptkerns. Ein SPE besteht aus einer Synergistic Processing Unit (SPU) (die eigentliche Recheneinheit), einem lokalen Speicher (LS) der Größe 256 Kilobyte und dem Memory Flow Controller (MFC) der den Datenaustausch zwischen SPE und Hauptseicher steuert. Wie das PPE ist jedes SPE in der Lage SIMD-Operationen auszuführen. Allerdings spielt die SIMD-Fähigkeit der SPEs eine wesentlich größere Rolle als bei dem PPE, da die SPEs für das Berechnen von großen Datenmengen gedacht sind und das PPE hauptsächlich Kontrollaufgaben übernehmen soll. Die Größe der SIMD-Operanden beträgt 128 Bit. Abhängig von der Operation können somit Datentypen mit einer Größe von 8 Byte - wie zum Beispiel zwei Double-Werte - bis hinab zu einer Größe von 1 Byte - zum Beispiel 16 Char-Werte - pro Operand verarbeitet werden. Der LS dient nicht nur als Daten-, sondern auch als Programmspeicher. Das heißt, dass das Programm, welches auf einem SPE ausgeführt wird, auch im LS liegt.
Wie bereits erwähnt, dient der MFC dem Datenaustausch zwischen LS und Hauptspeicher.

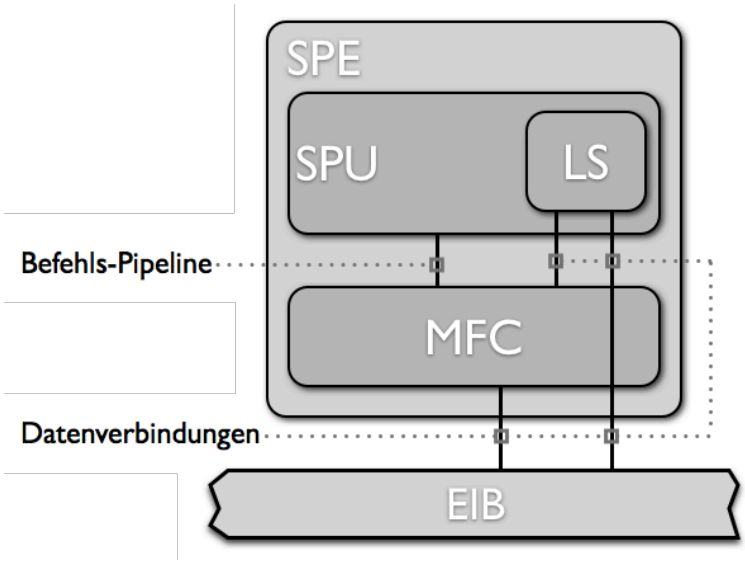

Abb. 2.2 - SPE Layout

Die Kommunikation zwischen SPE und Hauptspeicher verläuft asynchron. Durch den MFC ist es möglich auf einem SPE bereits erhaltene Daten zu verarbeiten, während gleichzeitig neue Daten - über den EIB - aus dem Hauptspeicher empfangen werden können. Umgekehrt ist es natürlich auch möglich, Daten in den Hauptspeicher zu schreiben, während Berechnungen durchgeführt werden. Diese Funktionalität ist ein wichtiges

Alleinstellungsmerkmal der Cell BE. Bisher übliche Mehrkern-Architekturen müssen auf den Abschluss des Speicherzugriffs warten, um weiter arbeiten zu können.

Element Interconnect Bus (EIB):
Der „Element Interconnect Bus“ ist der Kommunikations- und Befehls-Kanal zwischen allen auf der CBEA befindlichen Elementen. Der Bus besteht aus vier Ringen, an die alle Teilnehmer in einer bestimmten Reihenfolge angebunden sind.
Die Reihenfolge spielt in sofern eine Rolle, da Daten bzw. Befehle den EIB von ihrer Quelle zum Ziel entlang wandern müssen. Mit dem PPE, den acht SPEs, dem MIC und den beiden I/O Schnittstellen des BEI sind zwölf Teilnehmer über den EIB verteilt. Da Daten immer auf dem kürzesten Weg ausgetauscht werden, beträgt die maximale Distanz zwischen zwei Teilnehmern sechs Sprünge.

Memory Interface Controller (MIC):
Der „Memory Interface Controller“ stellt die Schnittstelle zwischen dem EIB und dem Arbeitsspeicher dar. Über den MIC können zwei Verbindungen zum Arbeitsspeicher genutzt werden. Diese Verbindungen sind vom Typ Rambus XDR. Über diese Verbindungen können zwischen 64 MB und 64 GB Arbeitsspeicher angebunden werden.

Cell Broadband Engine Interface Unit (BEI):
Die „Cell Broadband Engine Interface Unit“ ist die Schnittstelle zur Außenwelt der CBE. Sie beinhaltet wiederum zwei Rambus FlexIO Schnittstellen. Eine dieser beiden Schnittstellen dient zum Beispiel zur Anbindung von Ein-/Ausgabegeräten.

2.1.2 PlayStation 3
Da für diese Arbeit eine PlayStation 3 zur Verfügung stand, werden hier kurz deren Spezifikationen beschrieben. Das Herzstück der PS3 - der Prozessor - ist eine CBE. Allerdings sind hier nur sieben der acht SPEs aktiv.
Zur grafischen Darstellung wird eine RSX-Grafikkarte verwendet, die Sony zusammen mit NVidia entwickelt hat. Der Grafikspeicher hat eine Größe von 256 MB. Auf die Grafikbeschleunigung kann allerdings nur zurückgegriffen werden, wenn man eine Lizenz von Sony erwirbt, also berechtigt ist Spiele für die PS3 zu entwickeln und zu vertreiben. Da für die Implementationen der MD-Simulationen im Rahmen dieser Arbeit die Grafikkartenbeschleunigung nicht genutzt wird, ist die Grafikleistung nicht von Relevanz.
Der Arbeitsspeicher der PS3 ist auf 256 MB limitiert. MD-Simulationen können unter Umständen große Mengen an System-Ressourcen benötigen. Für die Messungen im Rahmen dieser Bachelorarbeit reicht der verfügbare Speicher von 256 MB allerdings aus.

2.2 Software
Im Folgenden wird auf die Software-Umgebung, im Besonderen auf die Gegebenheiten in Verbindung mit der PlayStation 3, eingegangen.

2.2.1 System
Üblicherweise läuft auf Maschinen die als Recheneinheit eine CBE haben eine Linux-Distribution. Die „Playstation 3“ wird mit der so genannten „PlayStation 3 system software“ als Betriebssystem ausgeliefert. Neben dem auf Multimediaanwendungen abgestimmten Betriebssystem, ist es auch möglich z.B. ein Linux zu installieren. Allerdings gilt dies nicht für Besitzer einer „PlayStation 3 Slim“. Für die „PlayStation 3“ sind bereits mehrere Linux-Distributionen verfügbar. Dazu gehören bekannte Systeme wie Ubuntu, Debian und

openSUSE. Die von Anfang an für die PS3 verfügbare Distribution ist Yellow Dog Linux. Yellow Dog Linux (YDL) beinhaltet seit der Version 6.1 einen Treiber, der es ermöglicht auf den schnellen Grafikkarten-Speicher zuzugreifen und diesen als Auslagerungs-Partition einzubinden. Somit wird der Arbeitsspeicher mit einer Größe von 256 MB auf 512 MB vergrößert. Allerdings wird der zusätzliche Speicher der Grafikkarte keinen so hohen Durchsatz bieten, wie es bei dem Arbeitsspeicher der Fall ist, da die Auslagerung der Daten vom Betriebssystem gesteuert wird.

2.2.2 Entwicklungsumgebung

Die Entwicklungsumgebung zur Programmierung von CBE-Basierten Programmen enthält alles was man benötigt, um Quelltext zu kompilieren, zu debuggen und zu analysieren. Das SDK installiert alle benötigten Bibliotheken, spezielle Compiler wie auch Debugger für PPE- und SPE-Programme. Darüber hinaus enthält das SDK eine spezielle Version der Entwicklungsumgebung Eclipse, die einen CBE-Simulator enthält, Beispiel-Quelltext und ein Paket an Standard-Makefiles, welches das Kompilieren von Quelltext wesentlich erleichtert.
Das gesamte Entwicklungspaket, momentan in der Version 3.1, ist auf den Webseiten von IBM (http://www.ibm.com/developerworks/power/cell/) unter „Downloads“ erhältlich. Das SDK kann nur heruntergeladen werden, wenn man sich zuvor registriert hat. Es liegt in zwei Versionen vor. Zum einen für Systeme, die auf RedHat basieren und zum anderen für Fedora-Derivate. Wenn man also auf einer „PlayStation 3“ mit YDL entwickeln möchte, dann ist das SDK für Fedora zu wählen, da YDL ein Fedora-Derivat ist. Eine genaue Installationsanleitung ist auf der oben genannten SDK-Seite unter „Documentation“ zu finden.

Buildutils:
Zur Entwicklung von CBEA-Programmen ist das bereits erwähnte Paket an Standard-Makefiles, die Buildutils zu empfehlen. Dieses Paket vereinfacht das Kompilieren von Programmen erheblich. Es muss ein eigenes Makefile angelegt werden, in dem die nötigsten Angaben getroffen werden und ein so genannter `make.footer` am Ende des eigenen Makefiles eingebunden werden. Der `make.footer` überprüft, welche Variablen gesetzt sind, um zu entscheiden, welche Compiler mit welchen Optionen aufgerufen werden. Wenn man zum Beispiel in seinem Makefile die Variable `PROGRAM_ppu64` angibt und ihr die Zeichenkette „`Prog`“ zuweist, führt das dazu, dass für den Hauptprozessor ein 64-Bit Programm kompiliert werden soll und den Namen „`Prog`“ tragen soll. Die Buildutils sind - nach der Installation des SDK - in dem Verzeichnis `/opt/cell/sdk/buildutils/` zu finden. Dort findet sich auch die Datei `README_build_env.txt`, die über alle Buildutil-Variablen informiert und deren Verwendung erklärt. Des Weiteren gibt es Beispiele zur Verwendung der Buildutils in dem Archiv `/opt/cell/sdk/src/tutorial_source.tar`.
Dieses Archiv zeigt auch, wie man die Struktur für seinen Quelltext wählen kann. Da man separate Programme für PPE und SPEs schreibt, teilt man auch den Quelltext entsprechend auf. Hat man zum Beispiel ein Programm für das PPE und ein Programm für alle SPEs, dann ist es sinnvoll, im Projekt-Verzeichnis einen Unterordner für den PPE-Quelltext und ein Verzeichnis für den SPE-Quelltext anzulegen. Im Basisverzeichnis des Projektes liegen die Dateien, die vom PPE- wie auch vom SPE-Quelltext benötigt werden.

2.2.3 Programmierung

CBEA-Programme werden üblicherweise in C oder C++ geschrieben. Zudem wird die gewählte Programmiersprache um die so genannten „CBEA Language Extensions“

erweitert. Diese definieren zum Beispiel Vektor-Datentypen für SIMD-Operationen oder MFC-Befehlssätze. Weitere Informationen sind in dem Dokument „C/C++ Language Extensions for Cell Broadband Engine Architecture“ [IBM08] zu finden. Programme für das PPE können in C++ geschrieben werden. Programme für die SPEs hingegen sollten in C geschrieben werden, da C-Code wesentlich weniger Platz benötigt. Zur Erinnerung: Jedes SPE hat einen lokalen Speicher von 256 KB, in dem auch der Programmcode gespeichert wird. Ein SPE-Programm was in C++ geschrieben ist, benötigt ein vielfaches dessen, was sein C-Äquivalent an Speicherplatz verbraucht. Zudem können verschiedene Konzepte von C++, wie Polymorphismus, die Geschwindigkeit eines Programms verlangsamen. Wenn man direkt in C programmiert, besteht gar nicht die Möglichkeit solche Konzepte in seinem Quelltext zu verwenden.
Programme für das PPE werden mit dem ppu-gcc beziehungsweise mit dem ppu-g++ kompiliert. Programme für ein SPE werden mit dem spu-gcc / spu-g++ kompiliert. Um Programme für das PPE zu kompilieren, kann man auch auf die Standart-Kompiler gcc und g++ zurückgreifen. Wie bereits erwähnt, muss man die Kompiler nicht direkt einsetzen, wenn man auf die vom SDK mitgelieferten „Buildutils“ zurückgreift.

Beispiel eines CBE-Programms:
In der Regel schreibt man ein PPE-Programm und ein oder mehrere SPE-Programme.
Das PPE-Programm startet für jedes SPE einen Thread. Hierfür bieten sich Posix-Threads an. Es muss für jedes SPE ein Kontext angelegt werden, über den dann ein SPE-Programm in den Speicher des jeweiligen SPE geladen wird. Zudem wird in der Regel für die SPEs jeweils eine Kontrollstruktur benötigt, über die der initiale Informationsaustausch zwischen PPE und SPEs stattfindet. Jeder SPE-Thread startet sein zugehöriges SPE und übergibt einen Zeiger auf die jeweilige Kontrollstruktur. Jeder Thread verweilt an dieser Stelle, bis das Programm auf seinem zugewiesenen SPE beendet oder unterbrochen worden ist. Während nun alle SPE-Threads auf die Beendigung ihrer SPEs warten, kann das Hauptprogramm die Koordination der SPEs bzw. der SPE-Programme übernehmen. Im einfachsten Fall wartet das Hauptprogramm darauf, dass die Threads beendet werden, um Resultate, die von den SPEs geliefert worden sind auszugeben oder weiter zu verarbeiten. Es können aber auch Nachrichten an die SPEs gesendet werden. Üblicherweise werden auch Daten für die SPEs vorbereitet und nachbereitet.
Ein SPE welches gestartet wurde, erhält wie bereits erwähnt einen Zeiger auf eine Kontrollstruktur. Über den Zeiger können die Daten der Struktur bezogen werden. Solch eine Struktur enthält zum Beispiel Position - im Hauptspeicher - und Größe der zu verarbeitenden Daten. Zudem wird in der Struktur vermerkt sein, wohin die verarbeiteten Daten zurück geschrieben werden sollen. Mit diesen Informationen kann das SPE-Programm die Daten beschaffen, sie verarbeiten und wieder zurück in den Hauptspeicher schreiben.

Einbinden der SPEs:
Der Zugang zu den SPEs vom Hauptprogramm aus wird über die Bibliothek libspe [IBM06] ermöglicht. Diese ist momentan in der Version 2 verfügbar. Es werden zum Beispiel Funktionalitäten zum Erstellen (spe_context_create) und Zerstören (spe_context_destroy) des SPE-Kontextes geboten. Des Weiteren sind Funktionen vorhanden, mit denen man zur Laufzeit eine SPE-Programm-Datei öffnen (spe_image_open) und schließen (spe_image_close) kann. Man kann die Programm-Datei aber auch statisch in das PPE-Programm einbinden. Das eingebundene bzw. geöffnete SPE-Programm-Abbild wird wie

bereits beschrieben in den SPE-Speicher geladen (spe_program_load). Außerdem ist eine Funktion zum Aktivieren (spe_context_run) eines SPE vorhanden.

Kommunikation:
Im Folgenden wird auf die Kommunikations-Konzepte eingegangen, die durch das CBEA-Framework zur Verfügung stehen. Alle hier erläuterten Konzepte betreffen die Kommunikation zwischen dem PPE, den SPEs und dem Hauptspeicher. Auf die Kommunikation per Signaling wird hier nicht eingegangen, da sie in dieser Arbeit nicht zur Anwendung kommt.

Kommunikation - DMA:
Da jede SPE nur einen lokalen Speicher besitzt, Daten aber eventuell mit anderen SPEs bzw. mit der PPE geteilt werden müssen, existiert der „Direct Memory Access“. Der DMA stellt die Verbindung zwischen einem SPE und dem Hauptspeicher dar. Ein DMA-Transfer wird in der Regel von einem SPE initiiert, kann aber auch von dem PPE veranlasst werden. Auf DMA-Transfers, die von dem PPE gesteuert werden, wird hier nicht eingegangen, da sie in dieser Untersuchung nicht zum Einsatz kommen. Ein DMA-Transfer bewirkt, dass entweder Daten aus dem Hauptspeicher in den lokalen Speicher eines SPE oder Daten aus dem lokalen Speicher des SPE in den Hauptspeicher geschrieben werden. Für diesen Zweck besitzt jedes SPE den „Memory Flow Controller“. Dieser ist allein für die Verarbeitung aller DMA-Transfers des jeweiligen SPE zuständig. Da der MFC den Datenaustausch verwaltet, kann die Recheneinheit des SPE während dessen Berechnungen auf anderen Daten durchführen und muss nicht warten, bis zu empfangende Daten angekommen sind bzw. zu sendende Daten verschickt wurden. Es wird also ein asynchroner Speicherzugriff möglich. Dadurch entsteht natürlich das Problem, dass man erst prüfen muss, ob die Daten, auf die man lokal zugreifen möchte, bereits übertragen wurden. Diese Überprüfung geschieht über ein Gruppen-Konzept.

Ein DMA-Transfer muss einer Gruppe zugewiesen werden. Im einfachen Fall hat ein Transfer eine eigene Gruppe. Es können aber auch, was intuitiv ist, mehrere Transfers einer Gruppe angehören. Nun kann man den Status einer Gruppe abfragen oder an einer Stelle im Programm warten, bis eine Gruppe einen bestimmten Status annimmt. So ist es zum Beispiel möglich zu warten, bis die Daten einer Gruppe übertragen wurden um sie dann zu verarbeiten.

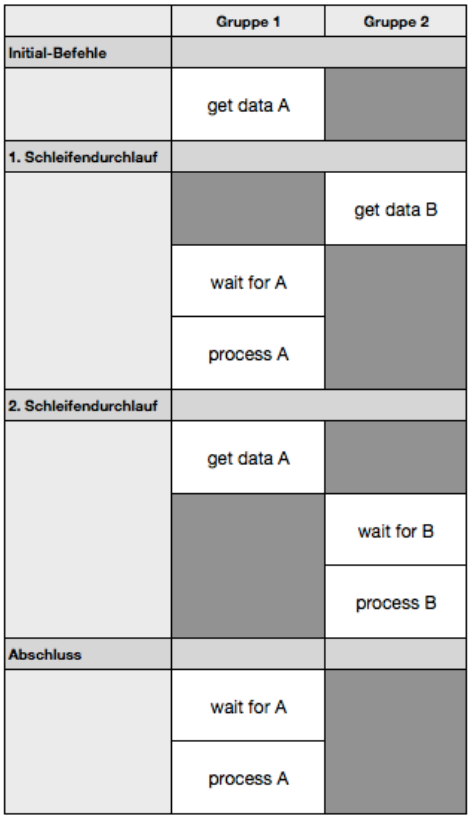

Abb. 2.3 - Doublebuffering

Üblicherweise initiiert man eine Datenübertragung in einer Gruppe 1 für die Daten A und eine weitere Datenübertragung in einer Gruppe 2 für die Daten B (siehe Abb. 2.3). Nun wartet man darauf, dass die A-Daten übertragen wurden und verarbeitet diese Daten. Währenddessen werden weiter die B-Daten übertragen. Wenn man nun die A-Daten verarbeitet hat, kann man in der Gruppe 1 eine weitere Übertragung starten. Nun wartet man darauf, dass B übertragen ist, und kann nun diese Daten verarbeiten. Auf diese Weise werden die ganze Zeit gleichzeitig Daten transferiert und Berechnungen durchgeführt.
Dieses Konzept nennt sich „Doublebuffering“. Es ließe sich auch ein „Tripplebuffering“ oder „Quadbuffering“ implementieren, in dem dann immer zwei oder drei Gruppen Daten transferieren und die Daten einer Gruppe verarbeitet werden. Allerdings wird der Quelltext zur Handhabung der Transfers auch immer komplizierter. Zudem ist die Bandbreite des EIB bzw. des MIC begrenzt und bei einer zu großen Menge an zu übertragenden Daten ausgelastet.

Es gibt verschiedene Varianten von DMA-Transfers. Mit den Befehlen mfc_get und mfc_put werden zusammenhängende Daten transferiert. Die Abbildung 2.4 veranschaulicht die Übertragung eines Datenblocks zwischen Hauptspeicher (RAM) und dem lokalen Speicher (LS) eines SPE.

Abb. 2.4 - Einfacher DMA-Tansfer

Mit den Befehlen mfc_getl und mfc_putl werden so genannte DMA-Listen übertragen. Wie in Abb. 2.5 zu sehen ist, werden im RAM verteilte Daten sequentiell in den LS eines SPE geschrieben, bzw. ein im LS sequentiell liegender Datenblock an unterschiedliche Adressen im RAM geschrieben.

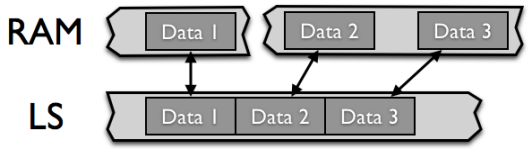

Abb. 2.5 - DMA-Listen-Transfer

Dadurch kann man eine Datenübertragung von mehreren im Hauptspeicher verteilten Datenpäckchen durch einen Übertragungsbefehl initiieren. Ein Listen-Element entspricht im Prinzip einem normalen DMA-Transfer. Dem DMA-Listen-Befehl wird ein Array übergeben, in dem jedes Element die benötigten Daten für die Übertragung enthält. Dies sind Informationen wie Quelladresse und Größe der zu übertragenden Daten. Zusätzlich wird in einem Bit gespeichert, ob folgende Listen-Elemente mit der Übertragung warten sollen, bis das aktuelle Element fertig übertragen wurde. Eine Beispiel-Implementation zu DMA-Listen ist in [IBM07] auf Seite 531 zu finden.
Desweiteren gibt es DMA-Transfer-Arten, die steuern, wann eine Übertragung relativ zu anderen Übertragungen derselben Gruppe stattfinden soll. Diese Varianten gibt es für normale DMA-Transfers, wie auch für DMA-Listen-Transfers. So kann man mit den „Fence“-Varianten (z.B. putf) veranlassen, dass der aktuelle Transfer nach allen bisherigen Transfers derselben Gruppe ausgeführt wird. Mit den „Barrier“-Varianten (z.B. getlb) kann veranlasst werden, dass der aktuelle Transfer nach allen vorherigen Transfers und zusätzlich auch vor allen folgenden Transfers aus derselben Gruppe ausgeführt wird.

Bei der Übertragung von Daten per DMA müssen einige Regeln beachtet werden. Quell- und Zieladresse müssen durch 16 teilbar sein. Es ist zwar auch möglich Adressen zu wählen, die auf ein, zwei, vier oder acht Byte ausgerichtet sind, dann müssen allerdings die vier niederwertigsten Bits von Quell- und Zieladresse gleich sein. Da es umständlich erscheint Quell- und Zieladresse so zu wählen, empfiehlt es sich alle Daten an 16 Byte-Adressen auszurichten.
Man kann Variablen in ihrer Deklaration leicht an bestimmten Bytegrößen ausrichten.

```
float* data[SIZE] __attribute__((aligned(16)));
```

Hier wird das Array vom Typ `float` automatisch so ausgerichtet, dass dessen Startadresse durch 16 teilbar ist.
Es gibt weitere Einschränkungen. Die Menge der zu übertragenden Daten muss ein Vielfaches von 16 Byte sein. Zudem kann man mit einem DMA-Befehl höchstens 16 KB übertragen. Dies gilt auch für die Daten der Listen-Elemente einer DMA-Liste. Zusätzlich gilt für DMA-Listen-Befehle, dass sie höchstens Listen mit 2048 Elementen verarbeiten können. Somit ergibt sich für Daten, die über eine DMA-Liste übertragen werden eine Höchstmenge von 16 KB * 2048, also 32 MB. Da der LS eines SPE momentan nur 256 KB fasst ist diese Angabe nur theoretisch, aber eventuell später nutzbar, falls der LS der SPEs bei späteren Modellen des CBEA erweitert wird.

Kommunikation - Mailbox:
Über das Mailbox-Konzept können Benachrichtigungen zwischen den SPEs und dem PPE versendet werden. So kann zum Beispiel ein SPE darauf warten, dass das PPE ihm eine Aufgabe zuteilt, indem das PPE eine Nachricht in der Inbox des SPE hinterlässt.
Eine Nachricht ist eine vorzeichenlose 32-Bit-Ganzzahl. Jedes SPE hat einen Mailbox Ein- und Ausgang. In den Eingang eines SPE können andere SPEs oder das PPE Nachrichten hinterlegen, die das SPE der Mailbox abfragen kann. Umgekehrt kann ein SPE eine Nachricht in den Ausgang der eigenen Mailbox legen und andere SPEs bzw. das PPE können die Nachricht auslesen.
Das PPE verfügt nicht über eine Mailbox. Eine Mailbox für das PPE kann aber über normale DMA-Transfers realisiert werden. Wenn das SPE dem PPE zum Beispiel signalisieren möchte, dass es seine Aufgabe erledigt hat, kann es im Hauptspeicher an einem dafür vorgesehenen Platz die Nachricht hinterlassen. Das PPE kann den Speicher in regelmäßigen Abständen auslesen, um zu sehen, ob das SPE seine Aufgabe erledigt hat.

Single Instruction Multiple Data auf einem SPE:
Um die eigentliche Rechen-Performance des CBE voll auszunutzen, ist es notwendig die SIMD-Befehlssätze der SPEs sinnvoll einzusetzen. Wie bereits erwähnt, haben die Operanden immer eine Breite von 128 Bit. Hierzu bieten die „CBEA Language Extensions“ [IBM08] den `vector` Modifikator für alle elementaren Datentypen an. Ein `vector` ist immer 128 Bit bzw. 16 Byte groß und ist auch immer an einer 16 Byte Adresse ausgerichtet. Ein `vector float` besteht zum Beispiel aus 4 Float-Werten, da pro Float-Wert 4 Byte benötigt werden. Dementsprechend nimmt ein `vector double` 2 Double-Werte auf. Üblicherweise und am performantesten rechnet man mit vier Float-Werten pro Operand, da die Recheneinheit eines SPE darauf angepasst ist. Es ist abhängig von der Operation, welche Datentypen unterstützt werden.

In dieser Untersuchung werden vor allem die arithmetischen und bitweisen Operationen und Vergleichsoperationen verwendet.
Für gängige Rechenoperationen wie Addition (`spu_add(a,b)`), Subtraktion (`spu_sub(a,b)`), Multiplikation (`spu_mul(a,b)`) sind SIMD-Varianten auf den SPEs vorhanden. Division muss durch eine Näherungsfunktion (12-Bit-Genauigkeit bei einem 32-Bit-Float Eingabewert), die 1/x approximiert (`spu_re(x)`), in Kombination mit einer Multiplikation umgesetzt werden. Es gibt auch zusammengesetzte arithmetische Operationen wie zum Beispiel `a*b+c` (`spu_madd(a,b,c)`).

$$\begin{pmatrix} a_0 \cdot b_0 + c_0 \\ a_1 \cdot b_1 + c_1 \\ a_2 \cdot b_2 + c_2 \\ a_3 \cdot b_3 + c_3 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} + \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

Abb. 2.6 - SIMD verknüpfte Multiplikation und Addition

Der Vorteil ist, dass diese kombinierten Operationen, wie es auch bei den einfachen Operationen der Fall ist, direkt von der Hardware unterstützt werden.

Für Vergleiche sind Operationen wie „größer als“ (`spu_cmpgt(a,b)`), „gleich“ (`spu_cmpeq(a,b)`) und Operationen, die Absolutbeträge miteinander vergleichen

(z.B: `spu_cmpabsgt(a,b)`) vorhanden. Die Rückgabe dieser Funktionen ist eine Bitmaske. Das folgende Beispiel zeigt die Funktionsweise an einem „größer als“ Vergleich. Die Variablen `a` und `b` sind vom Typ `vector float`, während `c` vom Typ `vector unsigned int` ist.

$$\begin{matrix} c \\ \begin{pmatrix} 1s \\ 0s \\ 1s \\ 1s \end{pmatrix} \end{matrix} = \begin{matrix} a \\ \begin{pmatrix} 1.3 \\ 4.1 \\ -2.0 \\ 123.8 \end{pmatrix} \end{matrix} > \begin{matrix} b \\ \begin{pmatrix} 0.0 \\ 4.3 \\ -15.36 \\ 50.25 \end{pmatrix} \end{matrix}$$

Abb. 2.7 - SIMD Größer-Vergleich

Die Einträge 1s und 0s in der Bitmaske c stehen jeweils für eine Menge an Bits, die auf 0 oder 1 gesetzt sind. In diesem Beispiel werden vier Floatwert-Paare miteinander verglichen. In dem ersten, dritten und vierten Vergleich ist die Größer-Bedingung erfüllt, weswegen im Ergebnis-Vektor dort alle Bits auf Eins gesetzt sind. Die Aussage des zweiten Vergleichs ist nicht erfüllt, daher sind dort im Ergebnis alle Bits auf Null gesetzt.

Zusätzlich wird hier noch der Selektions-Operator vorgestellt, da auch von diesem Gebrauch gemacht wird. Der Selektions-Operator (`spu_sel(a,b,c)`) kann in Verbindung mit den eben vorgestellten Vergleichs-Operatoren genutzt werden, um Daten abhängig von einem Vergleich zu wählen. Generell ist die Rückgabe des Selektions-Operators eine bitweise Kombination von `a` und `b` in Abhängigkeit von der Bitmaske `c`. Der folgende Pseudocode soll die Funktionsweise von `d = spu_sel(a,b,c)` veranschaulichen. Die hier verwendete Indizierung ist bitweise.

```
for ( i = 0 to 128 )
  if( c[i] == 0 ) d[i] = a[i]
  else d[i] = b[i]
```

In Verbindung mit der Bitmaske `c` einer Vergleichsopertation liefert `spu_sel` pro Komponente den Wert von `a` (wenn der Vergleich nicht erfüllt ist) oder `b` (wenn der Vergleich erfüllt ist).

In dieser Untersuchung sind auch Operationen nötig, die es erlauben, einen Vektor mit einzelnen Werten zu füllen und umgekehrt einzelne Werte aus einem Vektor zu extrahieren. Zu diesem Zweck sind die Befehle `spu_insert(a,b,c)`, `spu_extract(a,b)` und `spu_splats(a)` hilfreich. Der Befehl `spu_insert(a,b,c)` fügt das Skalar `a` in den Vector `b` an der Stelle `c` ein und liefert den modifizierten Vector zurück. Der Befehl `spu_extract(a,b)` liefert aus `a` das Skalar an der Stelle `b` zurück. Von dem Befehl `spu_splats(a)` wird ein Vector zurückliefert, in dem jedes Feld den Wert des Skalars `a` trägt.

Zusätzlich zu den bereits genannten Operationsarten sind auch logische Operationen, Sprung-Operationen sowie Operationen zum Shiften von 128-Bit-Werten vorhanden.

3. MD-Simulation

Im Folgenden wird erläutert, was eine MD-Simulation ist. Zudem werden die physikalischen Grundlagen und Werkzeuge erklärt, die benötigt werden, um eine MD-Simulation zu implementieren.

3.1 Physikalische Grundlagen

Den Ausgangspunkt einer klassischen Molekulardynamik(MD)-Simulation bilden die Newtonschen Bewegungsgleichungen eines Systems mit N Teilchen:

$$m_i \cdot \ddot{\overrightarrow{r_i}} = -\nabla_i \Phi\left(\{\overrightarrow{r_j}\}\right) = \overrightarrow{F_i}$$

Hierbei ist $\overrightarrow{r_i}$ die Position und m_i die Masse des Teilchens i. $\Phi\left(\{\overrightarrow{r_j}\}\right)$ ist das Potential und $\overrightarrow{F_i}$ ist die Kraft auf das Teilchen mit dem Index i. Die Menge $\{\overrightarrow{r_j}\}$ beschreibt die Positionen aller Teilchen $j = 1...N$ und somit die Konfiguration des Systems.

Für N-Teilchen-Systeme ($N > 2$) können die Bewegungsgleichungen im Allgemeinen nicht analytisch gelöst werden. Deshalb müssen die Bewegungsgleichungen für mehr als zwei Teilchen numerisch gelöst werden. Dafür bietet sich das Velocity-Verlet-Verfahren [ALLEN89a] an, auf das im folgenden Abschnitt näher eingegangen wird.

3.1.1 Velocity-Verlet-Verfahren

Die wesentlichen Eigenschaften dieses Verfahrens sind dessen numerische Stabilität und seine Zeitumkehrinvarianz [FRENKEL02a] .
Die folgenden Gleichungen beschreiben in ihrer Reihenfolge den Ablauf eines Integrationsschrittes.

$$\begin{aligned}
\overrightarrow{r_i}(t+\Delta t) &= \overrightarrow{r_i}(t) + \Delta t \cdot \overrightarrow{v_i}(t) + \frac{\Delta t^2 \cdot \overrightarrow{a_i}(t)}{2} \\
\overrightarrow{v_i}(t+\frac{\Delta t}{2}) &= \overrightarrow{v_i}(t) + \frac{\Delta t \cdot \overrightarrow{a_i}(t)}{2} \\
\overrightarrow{a_i}(t+\Delta t) &= \frac{-\nabla_i \Phi\left(\{\overrightarrow{r_j}\}\right)}{m_i} \\
\overrightarrow{v_i}(t+\Delta t) &= \overrightarrow{v_i}(t+\frac{\Delta t}{2}) + \frac{\Delta t \cdot \overrightarrow{a_i}(t+\Delta t)}{2}
\end{aligned}$$

Hier stellt Δt die Zeitdifferenz zwischen zwei Zeitschritten t_0 und t_1 dar ($t_1 - t_0 = \Delta t$). Des Weiteren ist a_i die Beschleunigung und v_i die Geschwindigkeit des Teilchens mit dem Index i.

In der folgenden Liste werden die einzelnen Schritte beschrieben:
a) Berechnung der Position aller Teilchen für den nächsten Zeitschritt
b) Berechnung des ersten Geschwindigkeitshalbschrittes aller Teilchen
c) Berechnung der Beschleunigung aller Teilchen für den nächsten Zeitschritt
d) Berechnung des zweiten Geschwindigkeitshalbschrittes aller Teilchen

Hier ist zu sehen, dass die Berechnung der Beschleunigung einen wesentlichen Anteil des Rechnungsaufwands darstellt, da die Berechnung der Kraft für jedes Teilchen im Allgemeinen über alle anderen Teilchen des Systems erfolgt. Das heißt, dass der Aufwand quadratisch mit der Anzahl der Teilchen wächst.

3.1.2 Energie und Temperatur eines Systems

Die Gesamtenergie E eines Systems ist die Summe aus der potentiellen Energie E_{pot} und der kinetischen Energie E_{kin}.

$$E = E_{pot} + E_{kin}$$

Die potentielle Energie E_{pot} eines Systems setzt sich zusammen aus den potentiellen Energien E_{pot_i} aller Teilchen. Das Selbe gibt für die kinetische Energie E_{kin}.

$$E_{pot} = \sum_{i=1}^{N} E_{pot_i} \qquad E_{kin} = \sum_{i=1}^{N} E_{kin_i}$$

Die potentielle Energie ist der Anteil der Energie, den ein Teilchen durch seine Lage im Potentialfeld Φ erhält. Für ein Masseteilchen berechnet sich die potentielle Energie

$$E_{pot_i} = m_i \cdot \Phi\left(\overrightarrow{r_i}\right)$$

Die kinetische Energie eines Masseteilchens ist:

$$E_{kin_i} = m_i \cdot \frac{{v_i}^2}{2}$$

Die Temperatur des in dieser Arbeit betrachteten Systems berechnet sich wie folgt:

$$T = \frac{2 \cdot E_{kin}}{3 \cdot k_B \cdot N}$$

3.2 MD-Simulation in dieser Untersuchung

In dieser Untersuchung werden einige Einschränkungen bezüglich der Systemeigenschaften vorgenommen. Das Simulations-System wird durch einen Quader mit einer festen Seitenlänge L räumlich begrenzt, dessen Ursprung $\overrightarrow{0}$ ist. Des Weiteren sind die räumlichen Grenzen periodische Ränder, das heißt, dass Teilchen über den Rand der Simulationsbox austreten können und auf der gegenüberliegenden Seite wieder eintreten. Durch das Verfahren der periodischen Randbedingungen können Oberflächeneffekte in der Simulation ausgeschlossen werden. Ein Teilchen wird durch eine Position und eine Geschwindigkeit beschrieben. Jedes Teilchen hat die Masse Eins. Die Wechselwirkungen geschehen paarweise zwischen den Teilchen p_i und p_j. Die Wechselwirkung findet nur dann statt, wenn der Abstand $r_{ij} = |\overrightarrow{r_i} - \overrightarrow{r_j}|$ dieser zwei Teilchen kleiner als der Cutoff c ist.

3.2.1 Lenard Jones Potential:

In dieser Arbeit wird das Lenard Jones Potential verwendet. Es stellt eine starke Vereinfachung der Wechselwirkung zwischen nicht geladenen Teilchen dar. Das Potential ist in zwei Teile aufgeteilt. Für sehr kurze Abstände stoßen sich Teilchen immer stärker voneinander ab. Für größere Abstände ziehen sich Teilchen an. Verwendet wird hier das

Lenard Jones (12,6) Potential $\Phi(r_{ij})$. Die Konstanten ϵ und σ definieren Längen- und Energieeinheiten. Sie werden in dieser Arbeit auf Eins gesetzt.

$$\Phi = 4 \cdot \epsilon \cdot \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right)$$

Der Gradient des Potentials ist die Kraft $\overrightarrow{F}$. Der Vektor $\overrightarrow{r_{ij}}$ ist der Abstandsvektor der beiden Teilchen, zwischen denen die Kraft berechnet wird.

$$\overrightarrow{F} = 4 \cdot \epsilon \cdot \left(6 \cdot \frac{\sigma^6}{{r_{ij}}^7} - 12 \cdot \frac{\sigma^{12}}{{r_{ij}}^{13}} \right) \cdot \frac{\overrightarrow{r_{ij}}}{\left| \overrightarrow{r_{ij}} \right|}$$

In dieser Arbeit wird das Lenard Jones Potential so eingestellt, dass sich Teilchen gegenseitig nur abstoßen, aber nicht anziehen. Deshalb wird in Verbindung mit $\epsilon = 1$ und $\sigma = 1$ der Cutoff $c = \sqrt[6]{2}$ gewählt.

3.2.2 Periodische Randbedingungen

Minimum Image Convention
Da in dieser Arbeit ein System mit endlicher Größe und mit periodischen Rändern verwendet wird, muss die Minimum Image Convention [ALLEN89b] beachtet werden. Diese besagt, dass zwei Teilchen in einem System mit periodischen Rändern über die kürzeste Distanz miteinander wechselwirken. Daraus folgt, dass der Cutoff c nicht größer als die Hälfte der Seitenlänge L der Simulationsbox sein darf.

Besonderheiten von Systemen mit periodischen Randbedingungen:
Abstandsmessungen zwischen zwei Teilchen p_i und p_j müssen die periodischen Ränder berücksichtigen [ALLEN89c]. Eines der Teilchen, zwischen denen die Abstandsmessung durchgeführt wird, muss temporär zu dem anderen Teilchen verschoben werden. Man überprüft für jede Koordinatenachse, ob die Teilchen weiter als eine halbe Box-Seitenlänge voneinander entfernt sind. Wenn dies der Fall ist, dann wird je nach Vorzeichen des Abstands die Seitenlänge der Simulationsbox subtrahiert oder addiert.

```
distance(r_i,r_j){
  d_x = r_ix - r_jx
  d_y = r_iy - r_jy
  d_z = r_iz - r_jz

  if(d_x > L/2) d_x -= L
  else if(d_x < -L/2) d_x += L
  if(d_y > L/2) d_y -= L
  else if(d_y < -L/2) d_y += L
  if(d_z > L/2) d_z -= L
  else if(d_z < -L/2) d_z += L

  return sqrt(d_x*d_x + d_y*d_y + d_z*d_z)
}
```

Zudem muss die Positionierung von Teilchen beim Austreten aus der Simulations-Box korrigiert werden. Dies ist intuitiv eine Modulorechnung, wird aber aus Kostengründen durch Substrahieren bzw. Addieren der Box-Seitenlänge realisiert. Dabei ist allerdings zu beachten, dass ein Teilchen sich in einem Integrationsschritt nicht weiter als eine Boxlänge bewegen darf, da die folgende Funktion, nur direkte Nachbarboxen auf die eigentliche Simulationsbox abbildet.

```
foldPoint(r_i){
  if(r_ix < 0) r_ix += L
  else if(r_ix >= L) r_ix -= L
  if(r_iy < 0) r_iy += L
  else if(r_iy >= L) r_iy -= L
  if(r_iz < 0) r_iz += L
  else if(r_iz >= L) r_iz -= L
}
```

3.2.3 Verlet-Listen:

Ein wichtiges Konzept zur effizienten Kraftberechnung bei kurzreichweitigen Wechselwirkungen sind Verlet-Listen [FRENKEL02b].
Bei kurzreichweitigen Wechselwirkungen hat jedes Teilchen nur eine begrenzte Anzahl an Nachbarn, mit denen es interagiert. Zudem kann man davon ausgehen, dass sich die Nachbarschaften zwischen Teilchen von Integrationsschritt zu Integrationsschritt nicht dramatisch ändern. Es ist daher nicht sinnvoll, in jedem Schritt für jedes Teilchen die möglichen Wechselwirkungen mit allen anderen Teilchen des Systems, die unter Umständen gar nicht nahe genug sind, zu untersuchen.
Jedes Teilchen besitzt eine Verlet-Liste, in der alle Teilchen vermerkt sind, deren Nähe mit diesem Teilchen Wechselwirkungen zulässt. Dadurch wird der durchschnittliche Aufwand bei der Kräfteberechnung in Abhängigkeit von der Gesamtzahl der Teilchen von einer quadratischen Laufzeit in eine lineare Laufzeit umgewandelt.
Da sich die Teilchen kontinuierlich bewegen, müssen die Verlet-Listen aktualisiert werden. Die Frage ist nun, wann die Verlet-Listen erneuert werden müssen. Prinzipiell müsste in jedem Integrationsschritt eine Aktualisierung statt finden, da sich auch die Sichtbarkeiten der Teilchen in jedem Schritt ändern können. Diese ständige Aktualisierung würde aber zu viel Aufwand bedeuten.
Man muss einen Trick anwenden, indem man die Sichtweite - den Cutoff c - um eine Übergangsschicht (Skin s) erweitert (Abb. 3.1). In dieser Übergangsschicht befinden sich Teilchen, die noch nicht oder gerade nicht mehr in einer Wechselwirkungsbeziehung mit dem aktuellen Teilchen stehen. So lange sich die Teilchen nicht weiter bewegen als Skin/2, müssen die Verlet-Listen nicht aktualisiert werden. Skin kann frei gewählt werden, insofern dadurch die Minimum Image Convention nicht verletzt wird.

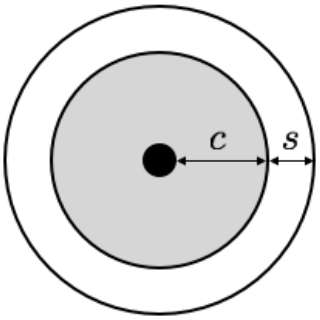

Abb. 3.1 - Sichtweite eines Teilchens

Berechnung von Kräften:
Bei der Berechnung der Kräfte muss berücksichtigt werden, dass nun auch Teilchen in den Verlet-Listen von benachbarten Teilchen stehen, obwohl sie nicht miteinander wechselwirken. Also muss bei der Kraftberechnung noch eine Abstandsprüfung ($r_{ij} < c$) für jedes Teilchen in der Liste durchgeführt werden.

Die Übergangsschicht realisiert man wie folgt: Zum Einen wählt man den Cutoff c ein wenig größer, indem man den Skin s - die Dicke der Übergangsschicht - aufaddiert, zum Anderen speichert man die Positionen aller Teilchen seit der letzen Verlet-Listen-Aktualisierung. Über die Positionen der Teilchen zum Zeitpunkt der letzten Aktualisierung der Verlet-Listen und die Positionen der Teilchen im aktuellen Integrationsschritt kann man sehen, wie weit sich jedes Teilchen seit der letzten Verlet-Listen-Aktualisierung bewegt hat. Ist diese Bewegung seit der letzten Aktualisierung größer als Skin/2, müssen die Listen aktualisiert werden.
Man muss Skin/2 wählen, da sich die Übergangsschicht auf den Abstand zwischen zwei Teilchen bezieht. Wenn sich nun zwei Teilchen, auf den Cutoff bezogen gerade noch sehen und sich jedes der Teilchen um Skin/2 von seiner Position von dem anderen Teilchen entfernt, ist die Distanz von beiden Teilchen um Skin gewachsen. Genauso dürfen sich zwei Teilchen jeweils höchstens um Skin/2 aufeinander zu bewegen, dass ihr Abstand gerade noch größer als Cutoff ist, wenn sie sich vorher nicht gesehen haben.

Wie werden Verlet-Listen aufgebaut?
Man kann die Verlet-Listen in quadratischer Laufzeit aufbauen. Für jedes Teilchen p_i geht man über alle anderen Teilchen p_j und überprüft, ob der Abstand r_{ij} kleiner als $c + s$ ist. Ist dies der Fall, wird das Teilchen p_j in der Verlet-Liste von p_i hinzugefügt.

Es ist aber auch möglich die Verlet-Listen in nahezu linearer Zeit aufzubauen. Hierzu bedarf es einer Gitterstruktur (Abb. 3.2), die die Simulationsbox in eine Menge gleich großer Unterboxen - auch Knoten genannt - unterteilt, die jeweils mindestens eine Seitenlänge von $c + s$ haben. In dieser Gitterstruktur werden die Teilchen abhängig von ihrer Position abgelegt. Da jedes Teilchen nur mit Teilchen interagiert, die maximal c entfernt sind, kann es höchstens mit Teilchen aus einer Nachbarbox wechselwirken. In einem dreidimensionalen System hat jede Box - sich selbst eingenommen - 27 Nachbarboxen. Somit hat man für jedes Teilchen nur eine eingeschränkte Anzahl an in Frage kommenden Nachbarteilchen, die überprüft werden müssen. Man geht über jede Box der Gitterstruktur und überprüft für jedes Teilchen aus einer Box die Abstände zu den

Teilchen aus den 27 genannten Nachbar-Boxen und trägt die Teilchen gegebenenfalls in die jeweiligen Verlet-Listen ein.

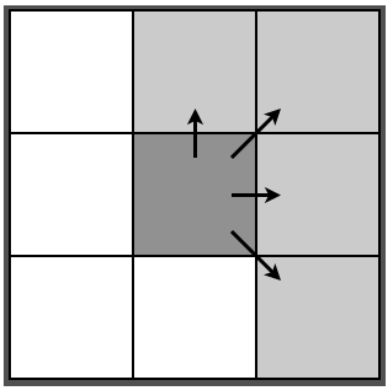

Abb. 3.2 - Gitter für die beschleunigte Erzeugung von Verlet-Listen

Eine weitere Optimierung besteht darin die Verbindungen zwischen Teilchen nur in einer Richtung aufzubauen. Der Hintergrund ist, dass die Kraftberechnung zwischen zwei Teilchen nur einmal durchgeführt werden soll, um Rechnungen einzusparen. Dies ist möglich, da nach dem 2. Newtonschen Axiom gilt: Kraft = Gegenkraft ($\overrightarrow{F_{ij}} = -\overrightarrow{F_{ji}}$). Das heißt, die Kraft die das Teilchen p_i auf das Teilchen p_j ausübt, ist genauso groß wie die Kraft, die das Teilchen p_j auf das Teilchen p_i ausübt, nur in umgekehrter Richtung. Somit muss die Kraft nur einmal berechnet werden.

Man muss zwei Fälle beachten, wenn man die oben genannte Gitterstruktur verwendet.
1. Wie werden Teilchen in derselben Raum-Box miteinander verknüpft?
2. Wie werden Teilchen der aktuellen Box mit Teilchen aus den Nachbar-Boxen verknüpft?

1. Bei Teilchen in derselben Box überprüft man einfach, ob eine Verbindung zwischen dem aktuellen Teilchen und allen nachfolgenden Teilchen in der Teilchen-Liste der Box besteht. So wird jede mögliche Verbindung nur einmal betrachtet.
Hat man zum Beispiel 3 Teilchen in einer Raum-Box, so werden folgende Überprüfungen durchgeführt: (1 nahe 2)?; (1 nahe 3)?; (2 nahe 3)?.

2. Bei Nachbarschaftsprüfungen von Teilchen von benachbarten Boxen kann man die räumliche Ordnung der Boxen nutzen, um zu verhindern, dass zwei Teilchen gegenseitig in ihren Verlet-Listen auftauchen. Ein Beispiel zur Überprüfung von Nachbarteilchen in benachbarten Boxen wird im Folgenden an einem 2D-Gitter veranschaulicht.

Das Gitter in Abbildung 3.2 zeigt ein 2D-Simulations-System bei dem der Raum / die Fläche in neun Bereiche aufgeteilt ist. Die Nachbarschaftsprüfung zur Erzeugung der Verlet-Listen muss nun für alle Teilchen in allen Boxen, wie bereits beschrieben, durchgeführt werden. Allerdings wird nun nur ein Teil aller benachbarten Boxen betrachtet. Das mittlere Kästchen stellt die aktuelle Box dar, von der die Nachbarschaftsprüfungen ausgehen. Die von der mittleren Box ausgehenden Pfeile zeigen, zu welchen Nachbarboxen die Überprüfung stattfindet. Es könnten auch andere Nachbar-Boxen gewählt werden. Wichtig ist, dass zwei Boxen nur einmal gegeneinander verglichen werden.

In einem System mit periodischen Randbedingungen müssen für Boxen am Rand auch die Nachbarboxen über den periodischen Rand hinweg berücksichtigt werden. Wenn zum Beispiel die untere mittlere Box die aktuelle Box ist, muss in dem Beispiel (Abb. 3.2) auch die Box rechts unterhalb überprüft werden, was in diesem Fall der rechten oberen Box entspricht.

Nach einer Menge von Integrationsschritten werden die Verlet-Listen neu erzeugt. Hierbei müssen für jede Box ihre Nachbarboxen bestimmt werden. Dies kann je nach Menge der Boxen einen signifikanten Anteil der Berechnungszeit ausmachen. Man kann dies aber verhindern, indem man die Nachbarschaften für jede Box einmal am Anfang bestimmt und sie für jede Box speichert. Dies nimmt natürlich zusätzlichen Speicher in Anspruch. Bei einer 3D-Simulation besitzt jede Box 13 Nachbarn - ohne sich selbst. Somit muss jede Box 13 Referenzen speichern, was aber gegenüber dem Performance-Gewinn bei der Verlet-Listen-Erzeugung vertretbar ist.

Teilchenpositionen im Gitter:
Jedes Mal, bevor die Verlet-Listen aktualisiert werden, muss für jedes Teilchen überprüft werden, ob es noch im richtigen Gitterknoten liegt, oder ob es inzwischen in einen Nachbarknoten gewandert ist.

3.2.4 Atomic Decomposition

Die Atomic Decomposition ist ein Ansatz, Teilchendaten parallelisiert zu verarbeiten. Bei der Atomic Decomposition werden die Teilchen über alle Berechnungsprozesse gleichmäßig verteilt, ohne auf ihre räumliche Ordnung zu achten. Hingegen werden zum Beispiel bei der Domain Decomposition die Teilchen nach ihrem Aufenthaltsort aufgeteilt. Bei der Domain Decomposition bekommt also jeder Berechnungsprozess einen Anteil des Simulations-Raumes zugewiesen.

3.2.5 Force Cap

Bei dem Aufsetzen eines Systems mit zufälligen Startpositionen der Teilchen ist es möglich, dass zwei Teilchen sehr nahe beieinander platziert werden. Bei der Kraftberechnung (siehe Lenard Jones Potential) steigt die Kraft für kurze Abstände zwischen Teilchen sehr stark an. Dies würde dazu führen, dass das System „explodiert“, da die Teilchen durch die hohe Krafteinwirkung in eine Bewegung mit sehr großer Geschwindigkeit versetzt werden. Um das zu verhindern, werden die Kräfte während der Äquilibrierungsphase durch den sogenannten „Force Cap“-Wert begrenzt.

4. Herangehensweisen

Das Ziel dieser Bachelorarbeit ist es herauszufinden, in wieweit die Performance einer MD-Simulation durch Verwendung einer CBE gesteigert werden kann.
Zu diesem Zweck gibt es eine Variante des Simulations-Programms, das für einen Prozessor geschrieben ist, wie verschiedene Varianten des Programms, die die SPEs der CBE nutzen. Alle Varianten verwenden den Velocity Verlet-Algorithmus, um für die Stabilität und Zeitumkehrinvarianz zu sorgen. Es werden verschiedene Möglichkeiten untersucht, die MD-Simulation auf eine CBE zu portieren. Grundsätzlich basieren alle Ansätze auf der Atomic Decomposition.
Das Hauptaugenmerk liegt auf der Parallelisierung der Kraftberechnungen, da diese den größten Aufwand in einer MD-Simulation darstellen. In der Kraftberechnung wird das Lenard Jones Potential verwendet.

Zeit-Messungen
Zum Vergleich aller Programm-Varianten, hier eine Zusammenstellung aller Randbedingungen.

1. Es werden alle Systemgrößen von `4x4x4` bis `18x18x18` Teilchen gemessen.
2. Die Ausmaße der Simulations-Box entsprechen der Menge an Teilchen. Zum Beispiel wird für `4x4x4` Teilchen eine Simulations-Box mit den Ausmaßen `4x4x4` verwendet.
3. Die Teilchen sind zu Beginn der Simulation grundsätzlich auf einem regelmäßigen $L \times L \times L$ Gitter platziert, wobei jedes Teilchen zusätzlich auf jeder Koordinatenachse um einen minimalen Zufallswert verschoben wird. L entspricht der Seitenlänge des Systems.
4. Die Zeitmessungen werden durch 16000 Integrationsschritte aufsummiert.
5. Folgende Abschnitte werden, wenn vorhanden, gesondert gemessen:
   a. Verlet-Listen-Erzeugung
   b. Positions-Berechnung
   c. Erster Geschwindigkeitshalbschritt
   d. Kraftberechnung je nach Programm-Variante
      i. Vorbereitung der Daten
      ii. Kraftberechnung selbst (bei SPEs inklusive des Datentransfers)
      iii. Nachbereitung der Daten
   e. Zweiter Geschwindigkeitshalbschritt
   f. Gesamtzeit eines Integrationsschrittes

Darstellung der Zeitmessungen
Zum Abschluss jedes der im Folgenden beschriebenen Verfahren werden die Zeitmessungen präsentiert. Es werden die Zeitmessungen aller Abschnitte eines Integrationsschrittes der jeweiligen Programm-Version dargestellt und eventuelle Besonderheiten beschrieben. Sofern nicht anders angegeben zeigt die x-Achse der Graphen die Anzahl der Teilchen und die y-Achse die Zeit in Sekunden an.

4.1 Grundlage

Das Einzelprozessor-Programm dient auch als Grundlage für alle folgenden Programm-Varianten. Dieses Programm führt eine einfache MD-Simulation aus. Es benutzt Verlet-Listen, um die Kraftberechnung zu optimieren.
Desweiteren werden in den Abstandsmessungen nur quadrierte Längen verwendet, um zu verhindern, dass die zeitintensive Wurzelfunktion benötigt wird. Dies ist möglich, da der Abstand selbst nie erforderlich ist. Wenn zum Beispiel überprüft wird, ob zwei Teilchen sich nahe genug sind, muss lediglich der quadrierte Abstand mit dem quadrierten Cutoff verglichen werden. Wenn der Wert des Abstandes selbst von Interesse wäre, wäre diese Optimierung nicht möglich.

Programmablauf des PPE:
Der grundsätzliche Programmablauf ist bei allen Varianten gleich, was für einen Vergleich vorausgesetzt werden muss.
Wenn das Programm gestartet wird, werden als erstes die dem Programm übergebenen Parameter überprüft. Man kann dem Programm entweder eine Reihe an Parametern übergeben, die das System beschreiben, oder eine Datei angeben, aus der ein Simulations-System geladen wird. Wenn das Simulations-System erfolgreich initialisiert wurde, wird eine feste Anzahl an Integrationsschritten berechnet.
Zum Abschluss der MD-Simulation werden alle Teilchenpositionen und deren Geschwindigkeiten zusammen mit den Eigenschaften des Simulations-Systems in die Datei „out.grid“ geschrieben.

In jedem Integrationsschritt (Abb. 4.1) wird zuerst - falls aktiviert - die Temperatur des Systems an einen Sollwert angepasst. Es ist zu erwähnen, dass die Temperaturregulierung während der Zeitmessungen nicht aktiviert ist. Im nächsten Schritt wird überprüft, ob die Verlet-Listen aktualisiert werden müssen. Wenn dies der Fall ist, werden sie aktualisiert und dann die Position aller Teilchen separat, für die nächste Verlet-Listen-Aktualisierungs-Prüfung, gespeichert. Darauf werden die Positionen aller Teilchen aktualisiert. Im nächsten Schritt wird der erste Geschwindigkeitshalbschritt für alle Teilchen durchgeführt. Nun wird je nach Programm-Variante die Kraft berechnet und die potentielle Energie des Systems bestimmt. Die Einzelprozessor-Variante übernimmt hier die gesamte Arbeit der Kraftberechnung, während die anderen Varianten, die auch auf die SPEs zurückgreifen, lediglich Daten zur Kraftberechnung für die SPEs vorbereiten und nachbereiten. Zum Abschluss eines Integrationsschritts wird der zweite Geschwindigkeitshalbschritt durchgeführt, die kinetische Energie sowie die Temperatur des Gesamtsystems berechnet und die Energien wie die Temperatur in die Datei „energy.txt“ geschrieben.

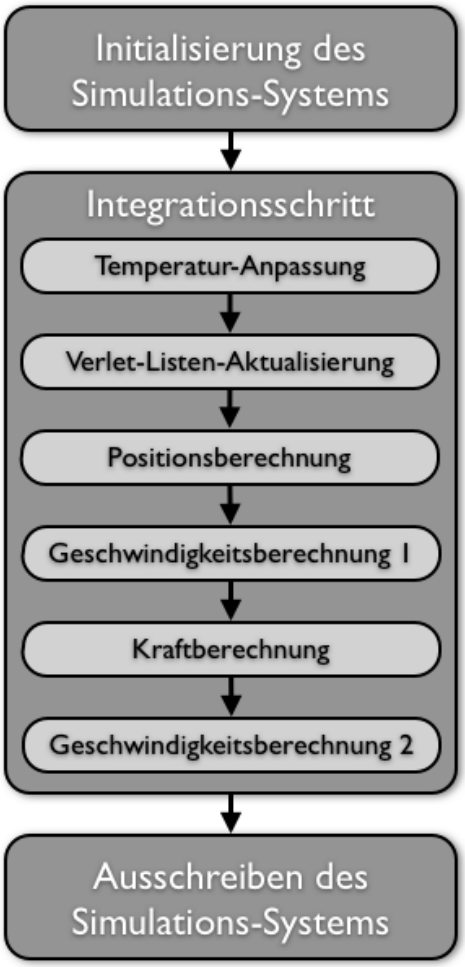

Abb. 4.1 - Genereller Programmablauf PPE

Programmablauf eines SPE:
Auch auf den SPEs ist der grundsätzliche Programmablauf bei allen Programm-Varianten gleich, wobei die SPEs im Einzel-Prozessor-Programm nicht verwendet werden.
Wenn ein SPE-Programm gestartet wird, bezieht es zu allererst die DMA-Kommunikations-Ids. Jede Id steht für eine Gruppe. Im nächsten Schritt holt das SPE seine Kontrollstruktur aus dem Hauptspeicher, in der verschiedene Einstiegspunkte für den Datenaustausch und Eigenschaften des Simulations-Systems gespeichert sind. Die Kontrollstruktur sieht bei allen Programm-Varianten grundsätzlich gleich aus.
Es gibt einen Zeiger auf den jeweiligen SPE -> PPE Mailbox-Eingang, einen Zeiger, über den die potentielle Energie zurückgeschrieben werden kann, und einen Zeiger, der auf eine separate Struktur verweist, die wiederum Zeiger auf die Teilchendaten enthält.
Zudem beinhaltet die Kontrollstruktur Eigenschaften des Systems wie den Cutoff, die Seitenlänge der Simulations-Box und den „Force cap“.
Wenn die initiale Kontrollstruktur empfangen wurde, geht das SPE-Programm in eine Endlos-Schleife. In dieser Endlos-Schleife werden die diversen Aufgaben, hauptsächlich aber die Kraftberechnung, durchgeführt. Zu Beginn jedes Schleifendurchlaufs sendet das SPE eine Nachricht an das PPE, dass es bereit ist, eine Aufgabe entgegen zu nehmen und wartet dann auf eine Nachricht des PPE. Das PPE wird in der Regel - in jedem Integrationsschritt - dem SPE mitteilen, dass Kräfte zu berechnen sind. Das SPE kann aber auch dazu aufgefordert werden, die Kontrollstruktur oder die Teilchendatenstruktur

neu zu empfangen oder die potentielle Energie, die durch die letzte Kraftberechnung aufsummiert wurde, an das PPE zurück zu liefern. Zudem kann das SPE-Programm durch eine Nachricht des PPE terminiert werden.
Wenn das SPE beauftragt wird, die Kraftberechnung durchzuführen, werden die Positionen der Teilchen aus dem Hauptspeicher empfangen, die Kraftberechnungen durchgeführt und die berechneten Kräfte wieder in den Hauptspeicher zurückgeschrieben. Dieser Zyklus - lesen, berechnen, schreiben - ist je nach Programm-Variante mit DMA-Doublebuffering umgesetzt.

Zeitmessung:
Die folgenden Graphen zeigen die Zeitmessungen des Einzelprozessor-Programms. Das Programm wurde auf einer Playstation 3 ausgeführt. Die gesamte Berechnungsarbeit fand somit auf dem PPE eines CBEA statt. Die *Tabelle 1* im Anhang zeigt alle Zeitmessungen des Programms im Detail.

Der Graph in Abb. 4.2 zeigt die Gesamtlaufzeit aller Integrationsschritte. Wie hier zu sehen ist, steigt die Laufzeit linear mit der Anzahl der Teilchen.

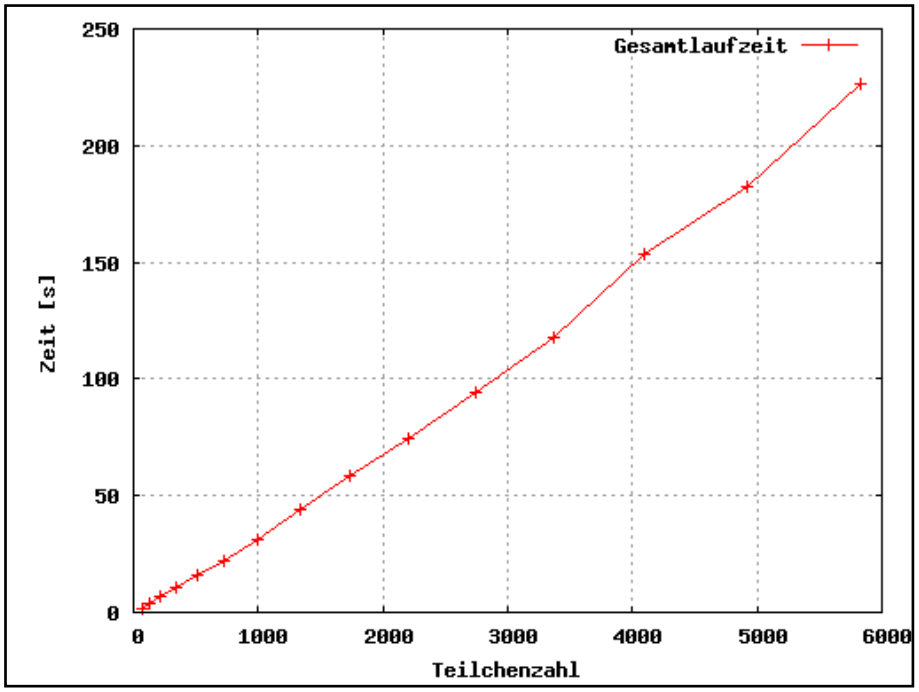

Abb. 4.2 - Gesamtlaufzeit Einzelprozessor

Der folgende Graph (Abb. 4.3) zeigt die einzelnen Zeitanteile, in die sich die Gesamtlaufzeit grob aufteilen lässt. Die Positions-, Geschwindigkeits- und Energieberechnung - rot - benötigt den kleinsten Teil der Zeit. Die Zeit um Verlet-Listen zu erzeugen - grüne Markierung - benötigt einen etwas größeren Teil. Die Kraftberechnung - blau - benötigt mit Abstand die meiste Zeit. Alle Zeiten wachsen nahezu linear mit der Systemgröße.

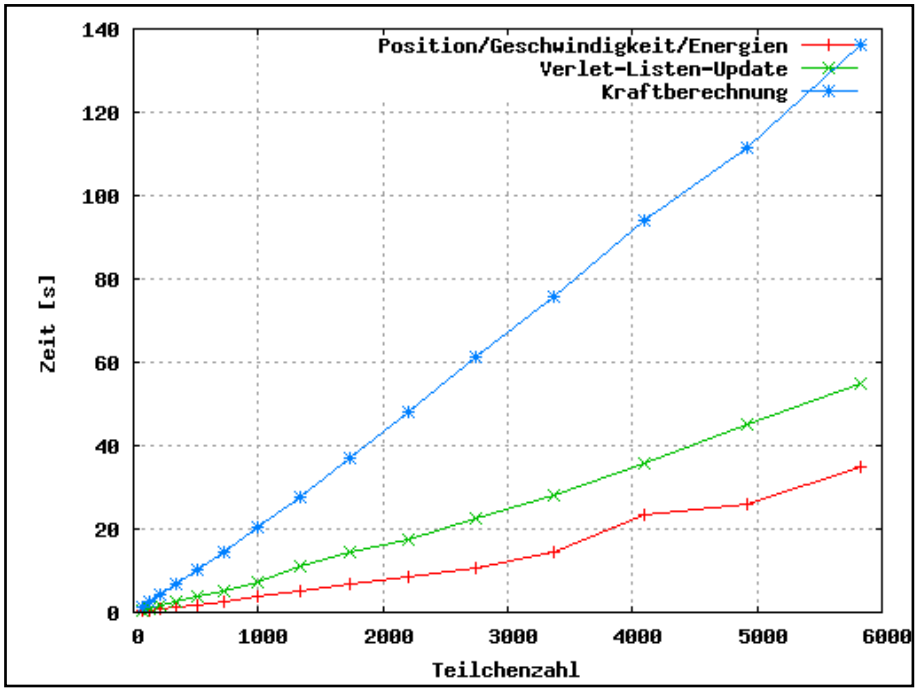

Abb. 4.3 - Aufspaltung der Gesamtlaufzeit

4.2 Parallelisierung auf SPEs

In dem ersten Parallelisierungsansatz wird die Kraftberechnung auf die SPEs verlagert. Alle anderen Berechnungen werden auf dem PPE ausgeführt. Da in diesem Ansatz Teilchendaten zwischen dem PPE und den SPEs ausgetauscht werden müssen, wurde versucht ein optimales Übertragungsprotokoll zu finden.
Der erste Ansatz bestand darin für jedes Teilchen eine SPE-Berechnung zu starten. Für jedes Teilchen p_i wird die Liste aller Nachbar-Teilchen p_j zusammengestellt und einem SPE zugewiesen.
Das Problem bei diesem Ansatz ist, dass die Menge der zu übertragenden Daten sehr gering ist. Geht man von einem angemessen dichtem System aus, hat jedes Teilchen ca. 9 Nachbarteilchen. Für jedes Teilchen müssen x, y und z-Koordinate der Position übertragen werden. Es müssen also 10*3 Float-Werte (120 Byte) übertragen werden. Je mehr Teilchen bei gleich bleibender Dichte in dem System vorhanden sind um so mehr dieser minimalen Daten-Transfers werden durchgeführt, was zu einem großen Funktions-Overhead führt. Zudem wird das Doublebuffering wirkungslos, da die Datenübertragung sehr schnell beendet ist, während die Berechnungen auf den SPEs noch andauern. Im optimalen Fall dauert die Übertragung für das nächste Daten-Paket so lange wie die Abarbeitung des aktuellen Daten-Paketes.
Es war ersichtlich, dass die Menge der an einem Stück zu übertragenden Daten signifikant gesteigert werden muss. Daher wurde der Ansatz modifiziert.
Als Optimierungsmaßnahme wurden alle Daten für jedes SPE jeweils an einem Stück vorbereitet, so dass jedes SPE möglichst große Daten-Mengen in einem DMA-Transfer übertragen und bearbeiten kann.

Daten-Protokolle:
Für jedes SPE wird von dem PPE im Hauptspeicher jeweils ein Buffer für Positionen wie auch für die Kräfte der Teilchen angelegt, in denen die zu übertragenden Daten vom PPE abgelegt bzw. abgeholt werden. Jeder Buffer besteht aus einer Menge Daten-Blöcke. Jeder Daten-Block wird durch einen DMA-Transfer übertragen.

Abbildung 4.4 zeigt den Aufbau des Protokolls für die Positionsdaten. Die Punkte (a) bis (d) zeigen Schritt für Schritt die Aufsplittung des Protokolls. Für jede SPE existiert ein Buffer im Hauptspeicher, der die zu übertragenden Daten beinhaltet (Abb. 4.4 a). Ein Buffer besteht aus einer Menge an Blöcken (Abb. 4.4 b). Die Menge der Blöcke bzw. die Größe aller Blöcke in Byte ist in der Kontrollstruktur des jeweiligen SPE vermerkt. Ein Datenblock hat eine zur Kompilierungszeit feste Größe, die der Größe eines DMA-Transfers entspricht. Jeder zu übertragende Datenblock enthält die Anzahl der enthaltenen Pakete und darauf die Pakete selbst (Abb. 4.4 c). Da die Größe der Datenblöcke fest ist, kann es vorkommen, dass die Größe der Datenblöcke nicht immer voll ausgenutzt wird. Am Anfang jedes Pakets seht die Nummer der Teilchen inklusive dem Referenz-Teilchen, darauf folgend die Teilchen-Positionen (Abb. 4.4 d).

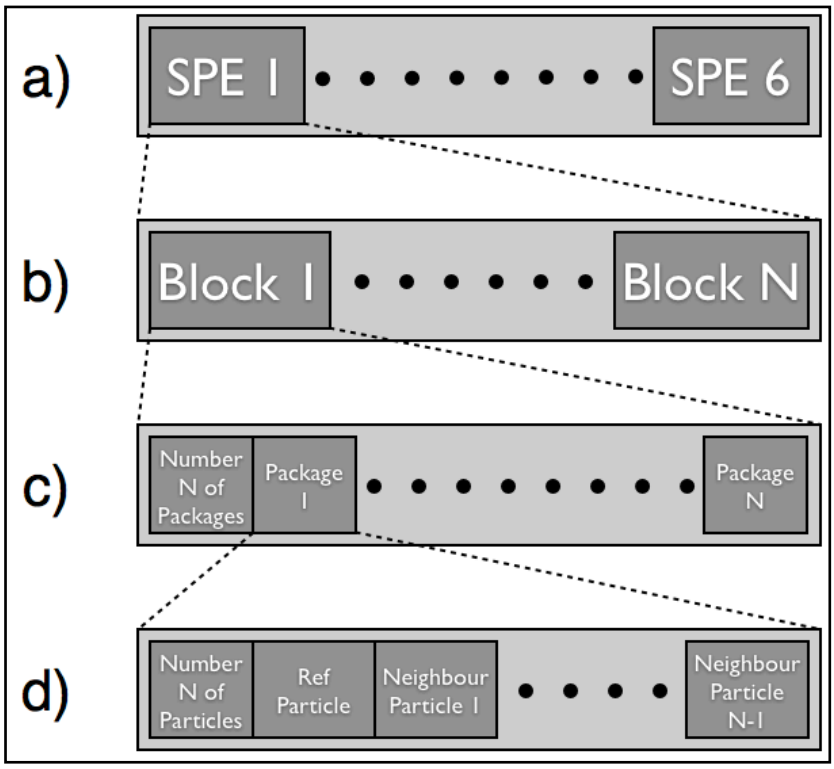

Abb. 4.4 - Positionsdatenprotokoll

Das Format der zurückgesendeten Kräfte ist ähnlich dem der Positions-Daten. Jedes SPE sendet für jeden eingehenden Positions-Daten-Block einen Kräfte-Daten-Block mit derselben Größe zurück. Jeder Kräfte-Daten-Block enthält als erstes einen Wert, der die Anzahl der Float-Werte für den jeweiligen Daten-Block angibt. Die Float-Werte sind die Koordinaten der Kraft-Vektoren in der Form $x_1, y_1, z_1, x_2, y_2, z_2, ..., x_n, y_n, z_n$. Die Kräfte-Daten eines Datenblocks werden nicht in Pakete unterteilt, da sie von dem PPE problemlos sequenziell ausgelesen und den richtigen Teilchen wieder zugeordnet werden können. Dies ist aber nur möglich, da während der gesamten Berechnung die Ordnung der Teilchen nie gestört wird.

Die Übertragung der Daten geschieht per Doublebuffering, während Kraftberechnungen durchgeführt werden. Beispielhaft ist der Ablauf mit zwei Schleifendurchläufen in Abbildung 4.5 dargestellt.
Das erste Daten-Päckchen mit Positionen wird von der Gruppe 1 bezogen. Nun wird eine Schleife betreten, die so lange wiederholt wird, bis keine weiteren Daten-Pakete zu übertragen sind. Falls nur ein Datenpaket vorhanden wäre, würde die Schleife nie betreten werden.
Es ist in der Grafik zu erkennen, dass während eines Schleifendurchlaufs eine Übertragung initialisiert und gleichzeitig eine beendete Übertragung verarbeitet wird. Das Zurücksenden der Kräfte geschieht über dieselbe Gruppe, über die die zugehörigen Positionen empfangen wurden und werden. Nach dem letzten Schleifendurchlauf muss das zuletzt empfangene Positions-Datenpäckchen noch verarbeitet werden und die daraus resultierenden Kräfte zurückgesendet werden. Die Zuweisung der Befehle zu den Gruppen ist hier nur beispielhaft. In jedem weiteren Schleifendurchlauf invertiert sich die Zuweisung der Befehle zu den Gruppen. Dies ist bei den zwei dargestellten Schleifendurchläufen zu sehen. So gehört zum Beispiel jeder Befehl, der im ersten Durchlauf der ersten Gruppe angehört, im zweiten Durchlauf der zweiten Gruppe an.

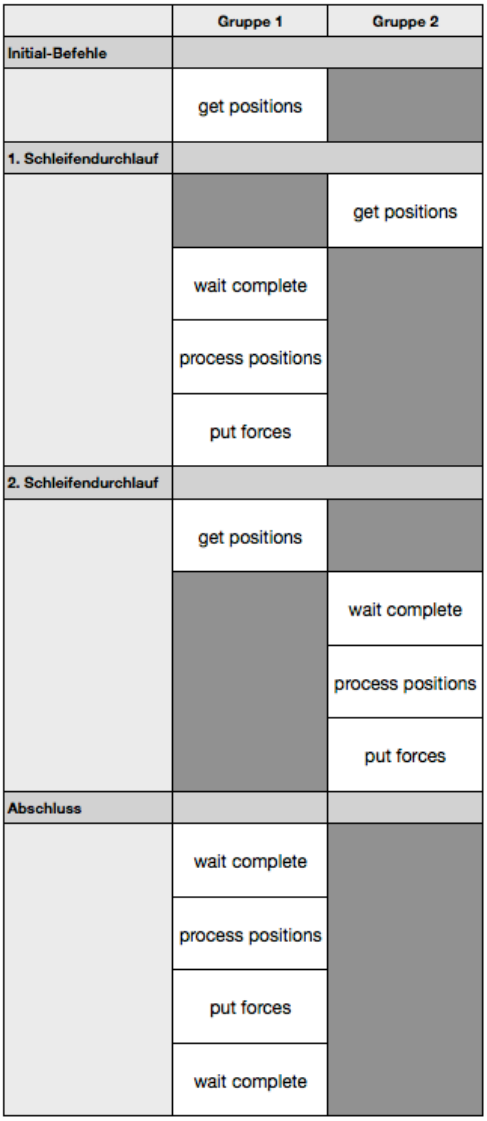

Abb. 4.5 - Beispielhafter Ablauf Kraftberechnung und Doublebuffering

Abwandlung des Programmablaufs:
Während der Initialisierung wird das SPE-Programm zur Berechnung der Kräfte in den Speicher der SPEs geladen und die SPEs aktiviert. Die SPEs warten dann auf eine Aufgabe. Den SPEs wird mitgeteilt, dass sie die Kontrollstruktur aktualisieren sollen, in der Informationen wie Startadresse der Teilchenpositionen, die Startadresse der Teilchenkräfte, Anzahl der zu verarbeitenden Teilchen und Systemeigenschaften wie zum Beispiel den Cutoff c und die Seitenlänge L der Simulations-Box übermittelt werden.
Wenn das PPE-Programm zur Kraftberechnung kommt, beauftragt es alle SPEs, die Kraftberechnung für ihren jeweiligen Anteil durchzuführen, und wartet, bis alle SPEs die Antwort senden, dass sie bereit sind, eine neue Aufgabe anzunehmen und somit die Kraft-Berechnungen beendet haben. Im nächsten Schritt fordert das PPE von allen SPEs deren akkumulierte potentielle Teilenergien an. Diese müssen noch zusammenaddiert werden.

```
calculateForces(){
  for (each spe) {
    prepare data for spe
    activate spe
  }

  wait for all SPEs to finish

  get partial energies from all SPEs and sum them up
  write back forces
}
```

SPE-Programm:
Das SPE-Programm arbeitet wie im letzten Kapitel beschrieben. Die Berechnung der Kräfte auf einem Datenblock wird durch den folgenden Pseudocode veranschaulicht.

```
for ( each reference particle pi ){
  for( each neighbour pj ){
    directionVector = calculateDirection(pi,pj)
    squaredDistance = calculateSquaredLength(directionVector)
    if( squaredDistance < squaredCutoff ){
      force = calculateLenardJonesForce(squaredDistance)
      potEnergySum += calculateLenardJonesEnergy(squaredDistance)
      force = capForce(force)
      piForceVector -= directionVector*force
      pjForceVector  = directionVector*force
    }
  }
}
```

Hier ist zu beachten, dass die Kraftberechnung wie folgt modifiziert wurde (vgl. Kapitel 3.2.1):

$$\overrightarrow{F} = 4 \cdot \epsilon \cdot \left(6 \cdot \frac{\sigma^6}{{r_{ij}}^8} - 12 \cdot \frac{\sigma^{12}}{{r_{ij}}^{14}} \right) \cdot \overrightarrow{r_{ij}}$$

Die Funktion `calculateDirection` berechnet $\overrightarrow{r_{ij}}$ und `calculateLenardJonesForce` berechnet

$$4 \cdot \epsilon \cdot \left(6 \cdot \frac{\sigma^6}{{r_{ij}}^8} - 12 \cdot \frac{\sigma^{12}}{{r_{ij}}^{14}} \right)$$

Die Berechnung der Kraft und der Energie ist im Quelltext kombiniert, da viele Terme in beiden Gleichungen vorkommen. Zum Beispiel ist der Abstand r_{ij} in beiden Gleichungen mit unterschiedlichen Exponenten vorhanden und es ist möglich die Anzahl der Multiplikationen zu minimieren. In der obenstehenden Gleichung ist auch zu sehen, das hier der quadrierte Abstand verwendet werden kann, da die Exponenten von r_{ij} gerade sind.

Zeitmessung:
Die folgenden Graphen zeigen die Zeitmessungen der gerade beschriebenen Multiprozessor-Variante. Die Berechnung der Kräfte wurde auf die SPEs ausgelagert. Alle anderen Berechnungen fanden auf dem PPE statt. Die *Tabelle 2* im Anhang zeigt alle Zeitmessungen des Programms im Detail.

Der Graph in Abbildung 4.6 zeigt die Gesamtlaufzeit aller Integrationsschritte. Wie im Einzelprozessor-Programm steigt auch hier die Laufzeit linear mit der Anzahl der Teilchen, wobei die Steigung hier etwas geringer ist.

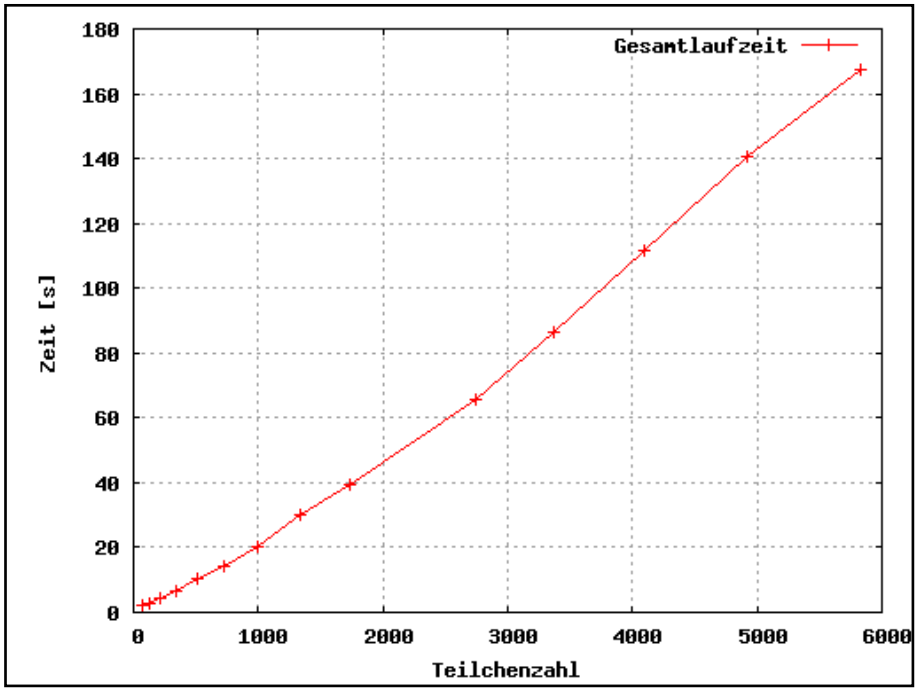

Abb. 4.6 - Gesamtlaufzeit Multiprozessor

Im folgenden Graph (Abb. 4.7) ist die Gesamtlaufzeit wieder in relevante Bestandteile aufgeteilt. Die Berechnung von Positionen, Geschwindigkeiten und Energien - zusammengefasst in der roten Markierung - benötigt nur einen sehr kleinen Teil der Zeit. Die Zeit zum Verarbeiten der Buffer - blau - benötigt einen signifikanten Teil der Zeit. Die Zeit zur Erzeugung der Verlet-Listen - grün markiert - benötigt auch einen großen Teil der Zeit. Mit zunehmender Systemgröße steigt die Zeit für Verlet-Listen-Erzeugung allerdings langsamer als die Zeit für die Datenvorbereitung und Nachbereitung der Buffer. Die Kraftberechnung - violett - fällt beim Zeitverbrauch kaum ins Gewicht.

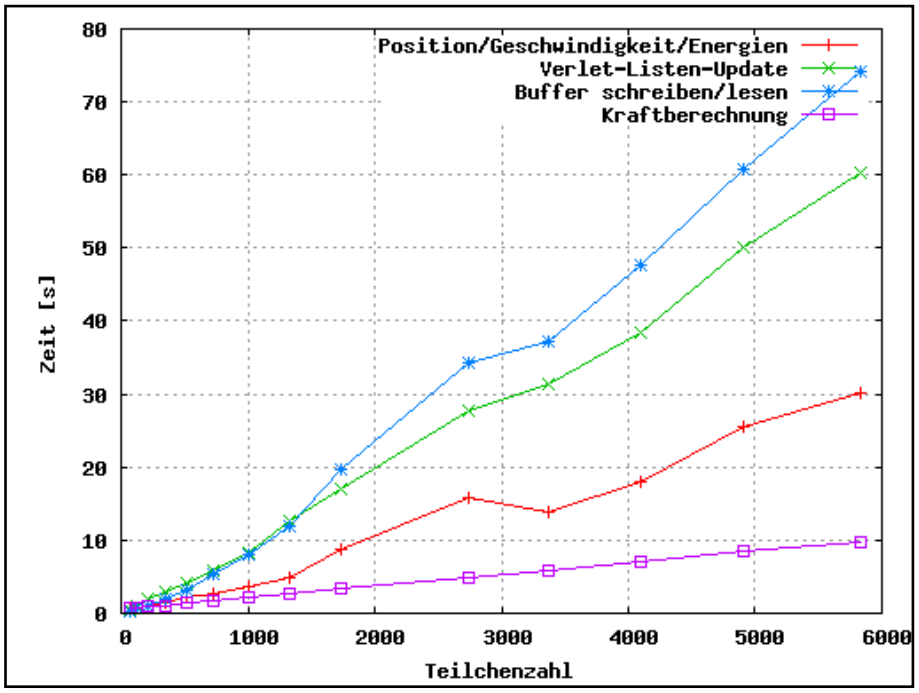

Abb. 4.7 - Aufspaltung der Gesamtlaufzeit

Ein beachtlicher Teil der Rechenzeit wird dafür benötigt, die Positions-Daten in die Buffer für die SPEs zu schreiben und nach der Verarbeitung die zurückgesendeten Kräfte-Daten aus den Buffern auszulesen und aufzuaddieren.
Der Grund für den großen Zeitverbrauch ist die Tatsache, dass die Caches des Hauptprozessors beim Verarbeiten der Buffer nicht optimal ausgenutzt werden. Daten, die aus dem Hauptspeicher geladen werden bestehen immer aus einem oder mehreren Datenblöcken (Cache Lines) fester Größe. Eine Cache Line des PPE ist 128 Byte groß. Die zu übertragenden Daten werden somit, unabhängig von der tatsächlich angefragten Datenmenge, in Vielfachen von 128 Byte aus dem Speicher in den Cache der CPU geschrieben.
In dem hier untersuchten Verfahren besteht jede Teilchenposition aus 3 Float-Werten. Somit belegt jede Teilchenposition 12 Byte im Speicher. Wenn die Positionen der Teilchen in einem Buffer zusammengestellt werden, muss an verschiedene Stellen im Speicher zugegriffen werden. Im schlechtesten Fall werden für jede 12 Byte Position 128 Byte im Cache abgelegt. So lange eine Cache Line noch im Cache liegt kann natürlich auf diese Daten mit erhöhter Geschwindigkeit zugegriffen werden (Cache Hit). Wenn allerdings auf Daten derselben Cache Line erst dann erneut zugegriffen wird, wenn sie bereits aus dem Cache entfernt wurde, muss die Cache Line erneut aus dem Hauptspeicher geladen werden (Cache Miss).

Bei der Zusammenstellung der Positionsdaten muss im Speicher eventuell sehr stark umhergesprungen werden, weil in der Simulation benachbarte Teilchen im Speicher eventuell weit voneinander entfernt liegen.
Ähnlich ist es bei den Kräften. Die von den SPEs zurückgesendeten Kräfte sind Teilkräfte, da wie in Kapitel 3 beschrieben eine Verbindung zwischen zwei Teilchen nur einmal betrachtet wird. Somit muss jede Teilkraft aus dem Buffer zu der eigentlichen Kraft des Teilchens aufaddiert werden. Hier liegen die Schreibpositionen im Hauptspeicher im Durchschnitt weit auseinander, was wieder zum Verwerfen von CPU-Cache-Daten führt.

4.3 Verwendung von bidirektionalen Verlet-Listen

Die Performance einer MD-Simulation kann also gesteigert werden, wenn der Zugriff auf die Daten sequentiell gestaltet werden kann. Die Positions-Daten der Teilchen kann man nur in begrenztem Maße so speichern, dass Nachbarschaften sequentiell im Speicher liegen, da sich Nachbarschaften im Verlauf einer Simulation sehr oft ändern können. Einfacher ist es die Kräfte von den SPEs in sequentieller Form zurück in die Buffer zu schreiben. Voraussetzung dafür ist, dass die Kräfte-Daten von den SPEs in der Reihenfolge zurückgeliefert werden, wie sie nativ vom PPE-Programm verarbeitet werden. Dazu gehört auch, dass ein SPE für jedes verarbeitete Teilchen dessen endgültige Kraft zurück liefert.
Für ein Teilchen kann nur dann seine endgültige Kraft berechnet werden, wenn zu diesem Teilchen alle Nachbarn bekannt sind. Der Aufbau der Verlet-Listen wurde zu Beginn so konzipiert, dass zwischen zwei interagierenden Teilchen nur eine Verbindung besteht, und somit eines der Teilchen bei dem Anderen in der Verlet-Liste verzeichnet ist aber nicht umgekehrt. Der Grund dafür war die Kraft zwischen zwei Teilchen nur einmal berechnen zu müssen. Diese Optimierung muss nun aufgegeben werden.
Hieraus resultiert nicht nur der Vorteil, dass die Kräftedaten vom PPE direkt sequentiell abgearbeitet werden können, sondern auch der Nachteil, dass zum Einen doppelt so viele Kraftberechnungen durchgeführt und zum Anderen auch doppelt so viele Positions-Daten zwischen dem PPE und den SPEs ausgetauscht werden müssen.

Daten-Protokolle:
Es wurde festgestellt, dass zum einen die Positionsdaten in ihrer Menge steigen. Andererseits nimmt aber gleichzeitig die Menge der zurückzusendenden Kräftedaten ab. Die Positionsdaten können weiterhin ohne Probleme mit dem in 4.2 definierten Protokoll übertragen werden. Es wurde auch das Protokoll für Kräftedaten aus 4.2 übernommen, um eine Steigerung der Programm-Komplexität zu vermeiden.
Da im Vergleich zum vorherigen Verfahren die Anzahl der zu übertragenden Positionsdaten steigt und die Anzahl der Kräftedaten sinkt, sind die Kraft-Datenblöcke nur zu einem geringen Teil gefüllt, da nachwievor die Datenblöcke für Positionen und Kräfte dieselbe Größe besitzen. Gegebenenfalls kann hier noch eine Optimierung vorgenommen werden, durch die die Größe der übertragenen Kraft-Datenblöcke separat minimiert wird.

Zeitmessung:
In den folgenden Grafiken sind die Zeitmessungen der Multiprozessor-Variante mit bidirektionalen Verlet-Listen zu sehen. Die *Tabelle 3* im Anhang zeigt alle Zeitmessungen des Programms im Detail.

Im folgenden Graph (siehe Abb. 4.8) ist die Gesamtlaufzeit aller Integrationsschritte für die verschiedenen Systemgrößen zu sehen. Die Laufzeit steigt, wie in den vorherigen Varianten, linear mit der Anzahl der Teilchen. Auch hier ist eine - wenn auch geringe - Verbesserung der Programm-Laufzeit gegenüber den zuvor behandelten Varianten zu erkennen.

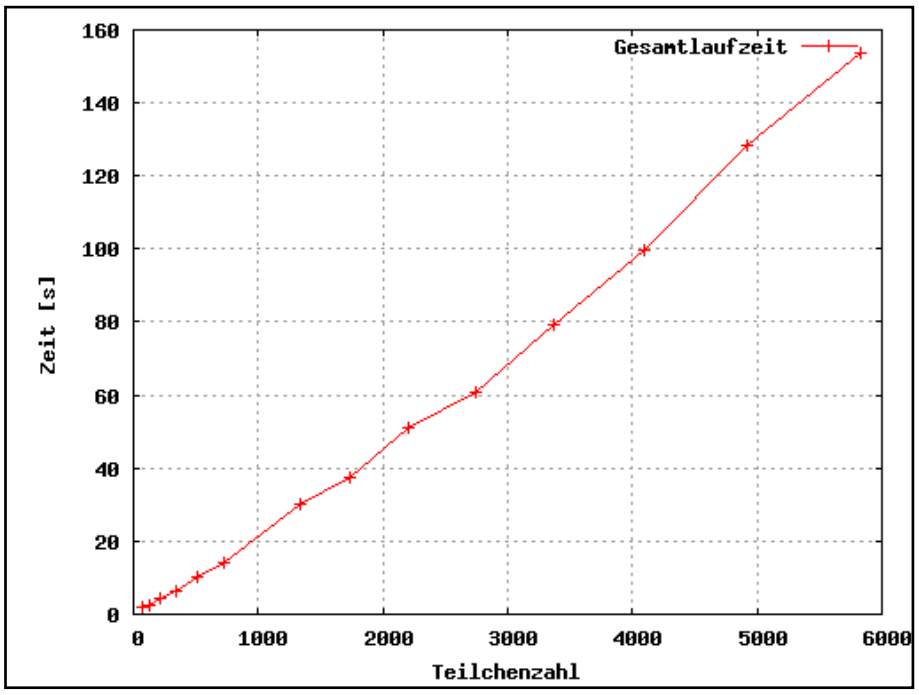

Abb. 4.8 - Gesamtlaufzeit

In der folgenden Grafik (Abb. 4.9), die wieder die Teilzeiten der Gesamtlaufzeit darstellt, ist zu erkennen, dass die Verarbeitungszeit der Buffer um einiges gesunken ist. Im Gegenzug ist die Zeit zur Erzeugung der Verlet-Listen ein wenig gestiegen. Das liegt daran, dass nun auch bei der Erzeugung der Verlet-Listen die doppelte Menge an Teilchenverbindungen überprüft werden muss.

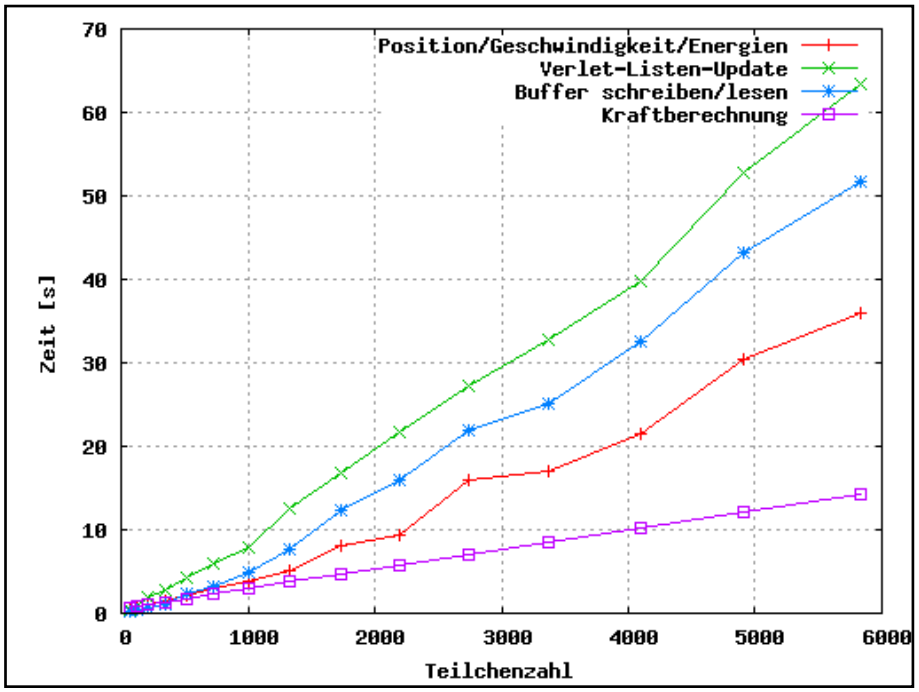

Abb. 4.9 - Aufspaltung der Gesamtlaufzeit

4.4 Verwendung von bidirektionalen Verlet-Listen und SIMD

In diesem Ansatz wurde versucht, die Rechenoperationen auf den SPEs durch SIMD-Operationen zu beschleunigen. Wichtig hierbei ist eine möglichst optimale Nutzung der SIMD-Operationen.
Es gibt zwei übliche Verfahren, die von der Speicherung der Daten abhängen. Gleichförmige Daten können als Array of Structures (AoS) oder als Structure of Arrays (SoA) gespeichert werden.

a) Array of Structures:
   Ein AoS speichert in einem Array pro Element zusammenhängende Daten. So würden zum Beispiel die Positionsdaten $(x, y, z)_i$ eines Teilchens in dem i-ten Element des Arrays gespeichert.
   Vorteil ist, dass die Speicherung der Daten in natürlicher Form vorliegt. Zudem sind zusammenhängende Daten gekapselt.
   Nachteil ist, dass SIMD-Operationen auf Daten in dieser Form schwieriger umzusetzen sind oder die Daten vor der Verwendung in solchen Operationen umorganisiert werden müssen.
b) Structure of Arrays:
   Ein SoA hingegen hält für jedes Datum ein separates Array bereit, wobei die Daten in allen Arrays an einem Index i zusammen gehören. Für Positionsdaten würde das bedeuten, dass für jede Koordinate (x, y, z) ein separates Array vorhanden ist und die Elemente x_i, y_i, z_i eine Position darstellen.
   Der Vorteil dieser Speicherung ist, dass die Daten zum Beispiel durch komponentenweise Berechnung die SIMD-Operationen meist besser ausnutzen können.
   Ein Nachteil ist hier im Gegensatz zu einem AoS, dass die Daten nicht in einer natürlichen Weise gespeichert werden. Zusammengehörige Daten sind über verschiedene Arrays verteilt und somit nicht gekapselt.

In dieser Untersuchung wurde aus folgenden Gründen auf ein AoS zurückgegriffen
1. Die Daten liegen bereits als AoS im Hauptspeicher vor.
2. Umorganisierung der Daten, um somit eine Fehlerquelle und einen beachtlichen Zeitaufwand zu vermeiden.
3. Vermeidung einer erhöhten Fehlerwahrscheinlichkeit durch die Verwendung von SoAs

Um die Daten auf einem SPE direkt in einer SIMD-Operation verwenden zu können, müssen die Daten-Pakete 128 Bit also 16 Byte groß sein. Die zu verarbeitenden Daten (Positionen und Kräfte) haben aber nur 12 Byte. Das Problem wurde einfach behoben, indem die Datenstruktur für Koordinaten um einen Float-Wert erweitert wurde. Dadurch erhöht sich das Datenaufkommen um ein Drittel, was bedeutet, dass ein Viertel der Daten umsonst übertragen werden. Man kann den überschüssigen Float-Wert für zusätzliche Informationen verwenden und zum Beispiel zusätzlich die Ladung des Teilchens unterbringen. Es wäre auch möglich gewesen, die Daten in ihrer ursprünglichen Form zu belassen und dann im SPE-Programm in 16-Byte-Blöcke umzukopieren und somit auch das benötigte Alignment zu erreichen. Dies würde aber einen erheblichen Mehraufwand bedeuten, der für jedes Teilchen (Position und Kraft) durchgeführt werden müsste.

Abwandlung des SPE-Programms:
Die Daten werden nun in Arrays vom Typ `vector float` gespeichert und das Empfangen und Senden von Daten wird auf 16 Byte-Pakete abgestimmt. Die Abstandsberechnung wird in eine SIMD-Operation umgewandelt. Desweiteren werden in der Kraftberechnung alle if-Verzweigungen eliminiert. Dies ist sinnvoll, da die SPE-Architektur für Massenberechnungen konzipiert wurde, aber nicht auf konditionale Sprünge im Code optimiert ist.
In der Kraftberechnung kamen bisher zwei offensichtliche if-Verzweigungen vor. Bei der ersten wird entschieden, ob die Kraft zwischen zwei Teilchen berechnet werden muss. Die zweite entscheidet, ob ein Force Cap angewendet wird. Es gibt noch weiter Verzweigungen in der Berechnung des Richtungsvektors der Kraft, bedingt durch den Code zur Behandlung der periodischen Ränder. In der neuen Variante werden bei allen if-Verzweigungen beide if-Zweige berechnet und das gültige Ergebnis verwendet. So wird der Quelltext Kraftberechnung durch folgenden Quelltext (Pseudocode) ersetzt.

```
for ( each reference particle pi ){
  for( each neighbour pj ){
    directionVector = calculateDirection(pi,pj)
    squaredDistance = calculateSquaredLength(directionVector)
    force = calculateLenardJonesForce(squaredDistance)
    potEnergy = calculateLenardJonesEnergy(squaredDistance)

    caped = select(force,-force cap, force     < -force cap)
    caped = select(caped, force cap, caped     > force cap)
    force = select(force,     caped, force cap > 0)

    potEnergySum +=
           select(0,potEnergy, squaredDistance < squaredCutoff)

    forceVector = mul(directionVector,force)
    piForceVector -=
           select(0,forceVector, squaredDistance < squaredCutoff)
    }
  }
}
```

Da hier einige Daten 128-Bit-Vektoren und andere Daten Floatwerte sind, müssen an manchen Stellen Daten konvertiert werden. Die berechnete Kraft ist zum Beispiel nur ein Floatwert, wird aber in SIMD-Operationen verwendet. Die Konvertierung von einfachen Floatwerten in 128-Bit-Vektoren, wie auch das Herausnehmen einzelner Werte aus einem 128-Bit-Vektor, wurde in 2.2.3 erklärt. Die Konvertierung wurde im Pseudocode ausgelassen, um den Fokus auf die Funktionsweise zu legen.
Die ersten Berechnungen aus dem Pseudocode sind bereits aus den vorherigen Programm-Varianten bekannt.
Der Force Cap wird hier nun als eine Folge von SIMD-Operationen angewendet, wodurch alle if-Verzweigungen aufgehoben werden. Die Kraft wird zuerst nach unten beschränkt, dann nach oben beschränkt und im dritten Schritt wird erst überprüft, ob die beschränkte oder die unbeschränkte Kraft weiter verwendet werden soll. Es ist hier definiert, dass ein negativer Force Cap bedeutet ihn nicht anzuwenden.

Die Anweisung `select` entspricht dem SIMD-Befehl `spu_sel(a,b,c)`. Jeweils der dritte Parameter wird durch eine der in Kapitel 2.2.3 erwähnten SIMD-Vergleichs-Operationen umgesetzt.
Im nächsten Schritt wird die potentielle Energie um den Teilbetrag aus der aktuellen Interaktion erhöht, wenn diese überhaupt statt findet. Genauso wird nur dann die Kraft subtrahiert, wenn die Teilchen sich nahe genug sind.

Zeitmessung:
In den folgenden Grafiken sind die Zeitmessungen der Multiprozessor-Variante mit bidirektionalen Verlet-Listen und der Verwendung von SIMD-Operationen im SPE-Programm zu sehen. Die *Tabelle 4* im Anhang zeigt alle Zeitmessungen des Programms im Detail.

Im folgenden Graph (Abb. 4.10) ist die Gesamtlaufzeit aller Integrationsschritte für die verschiedenen Systemgrößen zu sehen. Die Laufzeit steigt, wie in den vorherigen Varianten, linear mit der Anzahl der Teilchen. Auch hier ist eine - wenn auch geringe - Verbesserung der Programm-Laufzeit gegenüber den zuvor behandelten Varianten zu erkennen.

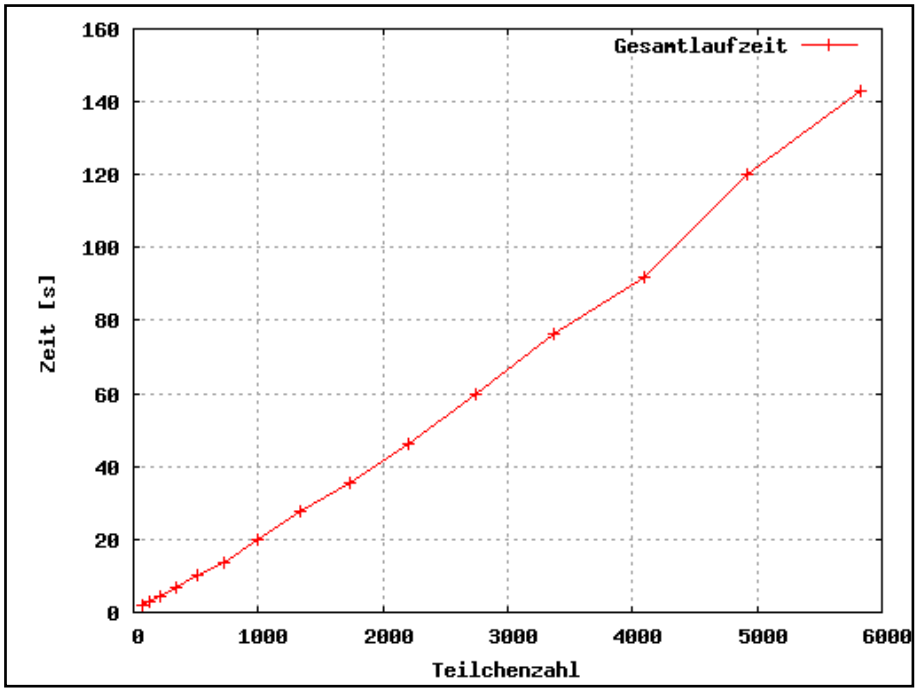

Abb. 4.10 - Gesamtlaufzeit

In der folgenden Abbildung 4.11 sind wieder die Teilzeiten der Gesamtlaufzeit darstellt. Es ist zu erkennen, dass die Verarbeitungszeit der Buffer wieder ein wenig angestiegen ist. Die Berechnung der Kräfte hat sich nur leicht beschleunigt. Der Geschwindigkeitsgewinn fällt nur so gering aus, da die Berechnungszeit der Kräfte im Vergleich zur Gesamtlaufzeit nur einen geringen Anteil ausmacht.

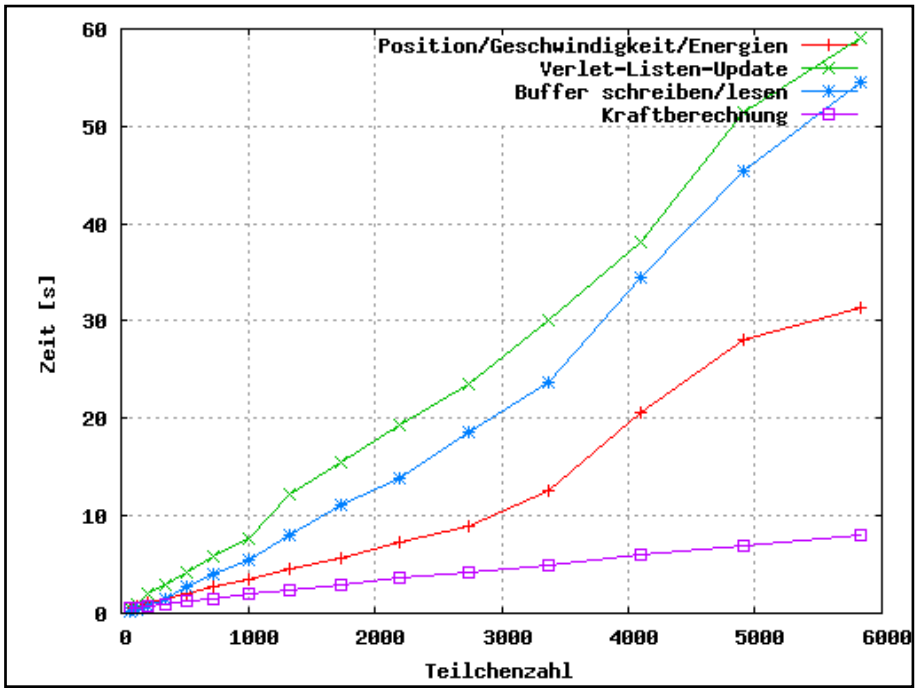

Abb. 4.11 - Aufspaltung der Gesamtlaufzeit

4.5 Quadratisches Abarbeiten der Kräfte

In diesem Ansatz werden alle Optimierungen, die durch die Verlet-Listen erzielt werden verworfen, um zu sehen, ob es sich lohnt, die pure Geschwindigkeit der SPEs in Bezug auf Berechnungen zu nutzen. Dadurch dass keine Verlet-Listen verwendet werden, können die Teilcheninformationen sequenziell aus dem Hauptspeicher geladen werden, wenn dies auch einen quadratischen Aufwand darstellt. Teilchen für mehrere Berechnungen in einem SPE zwischen zu speichern ist nur für eine begrenzte Zahl von Teilchen möglich, da der LS mit 256 Kilobyte begrenzt ist. Daher wurde entschieden auf die Reduzierung des Datenaufkommens für kleine Teilchenmengen zu verzichten.

Ablauf des SPE-Programms:
Bei dem quadratischen Ansatz werden für alle Teilchen p_i die Kräfte aus den Wechselwirkungen zu allen anderen Teilchen $p_j \neq p_i$ berechnet. So muss jedes Teilchen N (Anzahl der Teilchen im System) mal aus dem Hauptspeicher geladen werden. In dem Programm wird zwischen den Teilchen p_i und den Teilchen p_j unterschieden. Für beide Teilchenmengen p_i und p_j sind Arrays vorhanden. Jedes der Arrays nimmt eine Menge von Teilchen durch einen DMA-Transfer auf. Die Daten werden wie üblich durch Doublebuffering übertragen, wobei p_i -Teilchen über zwei separate Gruppen und die p_j -Teilchen über zwei separate Gruppen übertragen werden. Der vereinfachte Ablauf sieht wie folgt aus:

```
while( there is more p_i-data ){
  doublebuffer get p_i-positions
  for( each particle p in p_i ){
    while( there is more p_j-data ){
      doublebuffer get p_j-positions
      calculate forces between p and {p_j}
    }
  }
  doublebuffer send p_i-forces
}
```

Zeitmessung:
In den folgenden Grafiken sind die Zeitmessungen der Multiprozessor-Variante des quadratischen Ansatzes zu sehen. Die folgenden Graphen können optisch nicht mit den vorhergehenden Graphen verglichen werden. Die Zeitmessungen wurden nur bis zu einer Systemgröße von 13x13x13 Teilchen durchgeführt, da die Berechnungsdauer sehr groß wurde. Die *Tabelle 5* im Anhang zeigt alle Zeitmessungen des Programms im Detail.

In folgenden Graph (Abb. 4.12) ist die Gesamtlaufzeit aller Integrationsschritte für die verschiedenen Systemgrößen zu sehen. Die Laufzeit steigt hier quadratisch mit der Anzahl der Teilchen. Hier ist eine drastische Verschlechterung der Programm-Laufzeit zu sehen.

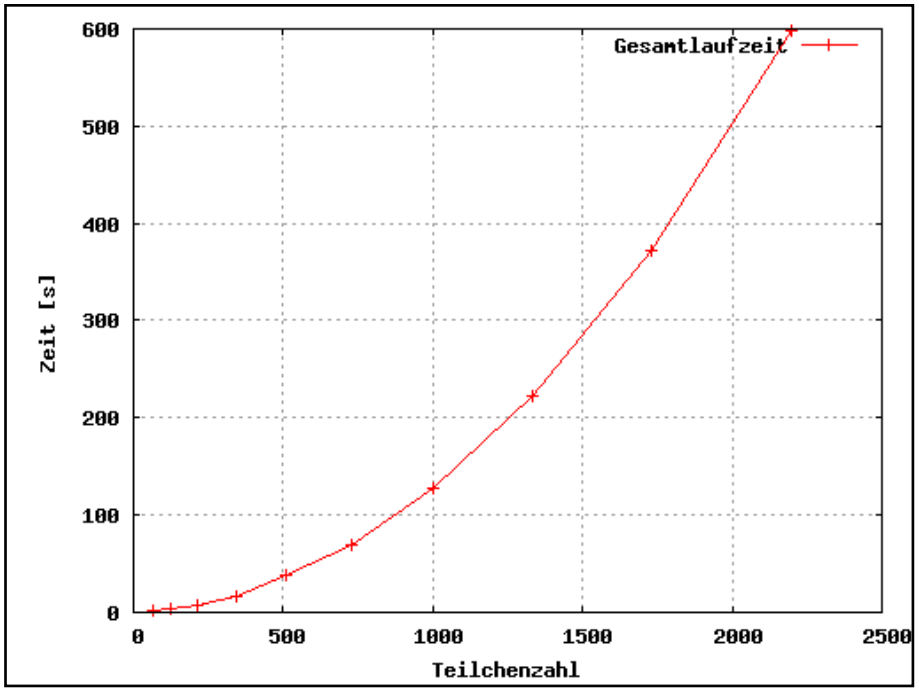

Abb. 4.12 - Gesamtlaufzeit

In dem Graph in Abbildung 4.13 sind die Teilzeiten der Gesamtlaufzeit darstellt. Es ist zu erkennen, dass die Verarbeitungszeit der Buffer stark gesunken ist. Das liegt daran, dass die Buffer nur noch für das Zusammenstellen und Auslesen von sequentiellen Daten verwendet werden. Der größte Teil der Zeit fließt in die Berechnung der Kräfte und den damit verbundenen Austausch von Daten zwischen PPE und SPEs.

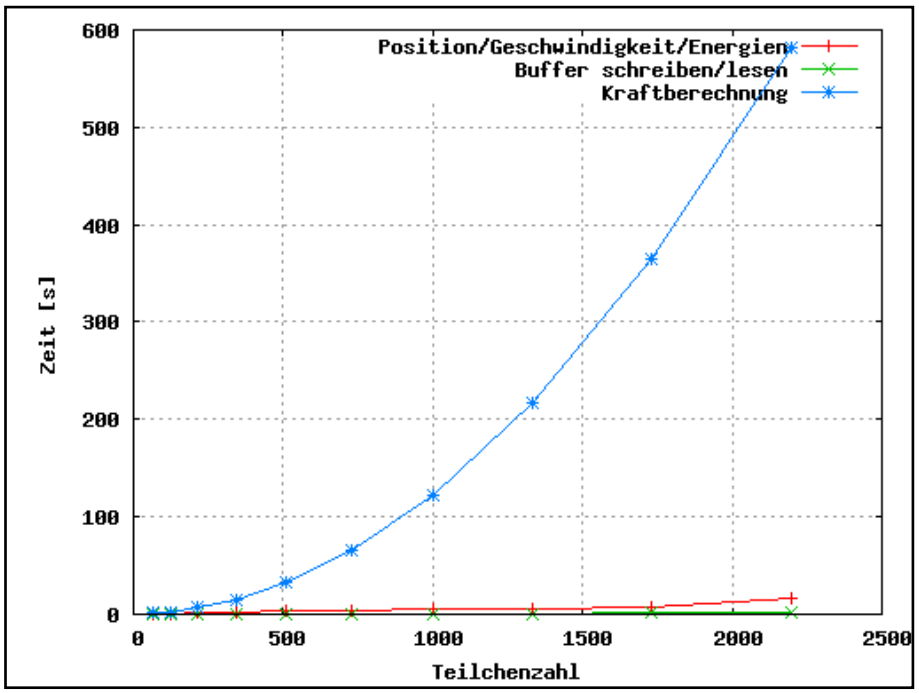

Abb. 4.13 - Aufspaltung der Gesamtlaufzeit

5. Vergleich

Bei einem ersten Überblick über die untersuchten Verfahren ist zu erkennen, dass die Parallelisierung einer MD-Simulation - im Speziellen bei Simulationen mit kurzreichweitigen Wechselwirkungen - auf einer CBEA grundsätzlich sinnvoll ist, da bei allen Verfahren bis auf die quadratische Variante ein Geschwindigkeitsvorteil gegenüber dem Einzelprozessor-Programm erzielt wurde (siehe Abb. 5.1).

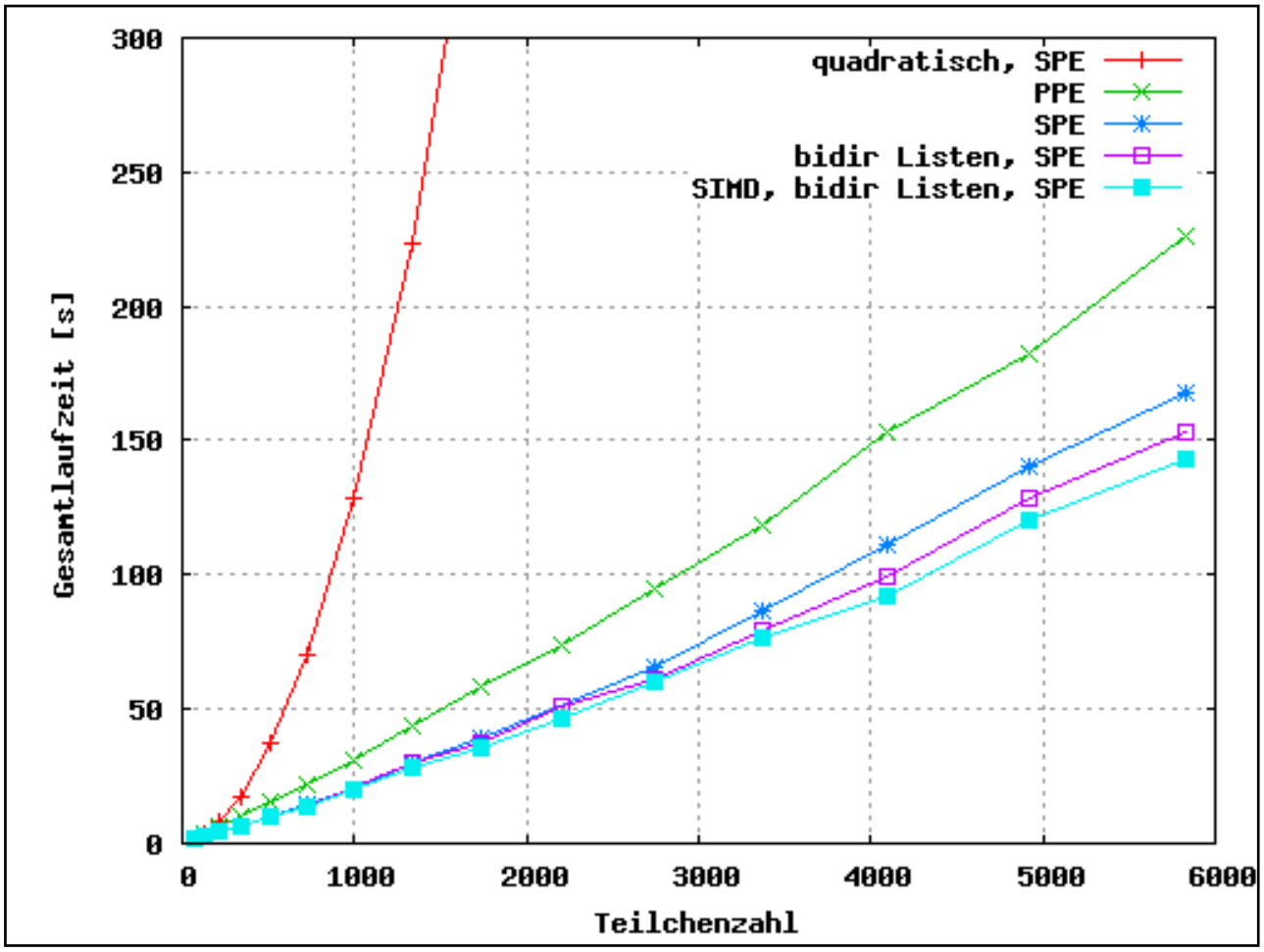

Abb. 5.1 - Vergleich: Gesamtlaufzeit aller Methoden

In den Abbildungen 5.2 und 5.3 ist zu sehen, dass die Verlet-Listen-Erzeugung und die Positions-, Geschwindigkeits- und Energieberechnungen zwischen den verschiedenen Methoden kaum variieren. Somit tragen diese Bereiche nicht relevant zu den Unterschieden in den Laufzeiten bei.

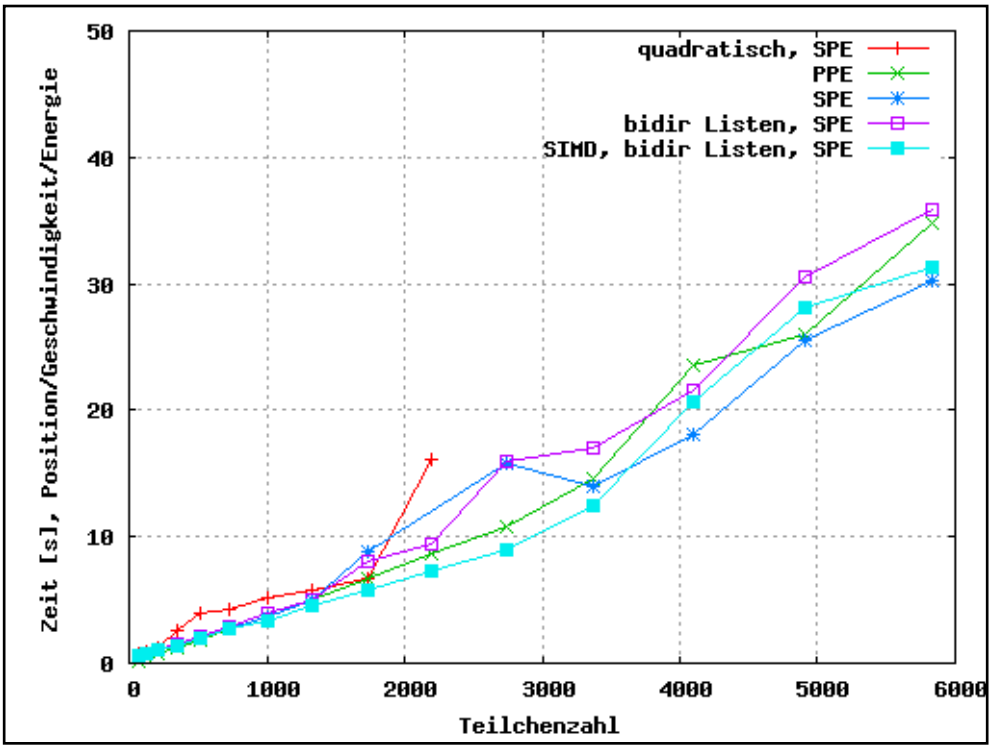

Abb. 5.2 - Vergleich: Positions-/Geschwindigkeits-/ Energieberechnung

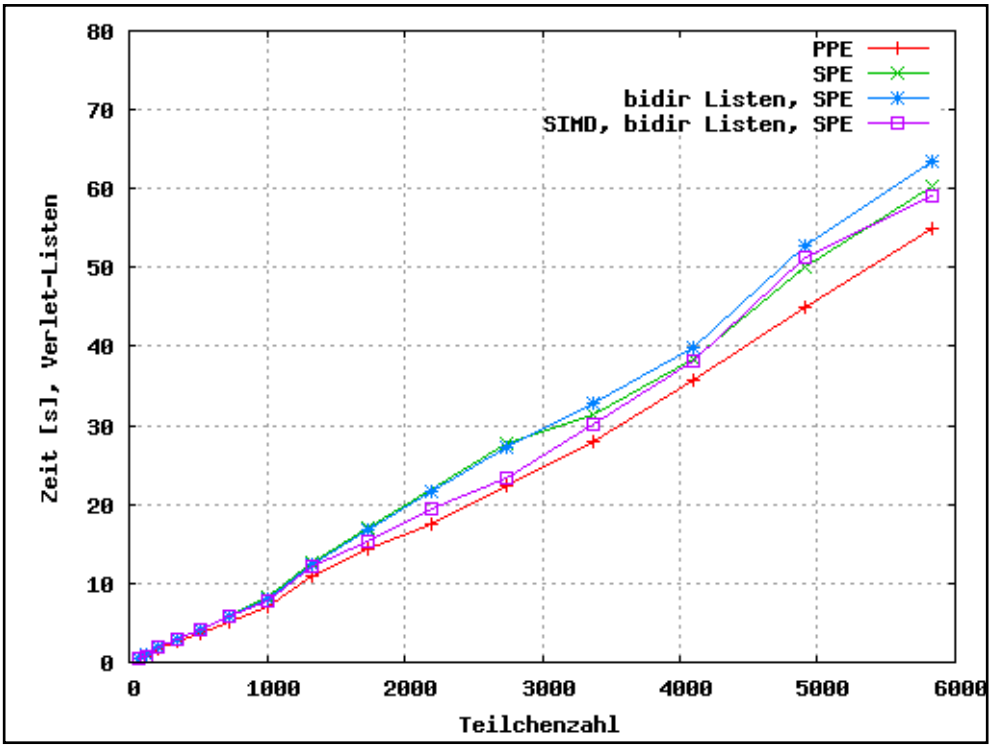

Abb. 5.3 - Vergleich: Verlet-Listen-Erzeugung

Abbildung 5.4 zeigt die Zeitunterschiede in der Verwendung der Buffer. Das Einzelprozessor-Programm ist hier nicht aufgelistet, da es keine Buffer verwendet. Der erste Parallelisierungsansatz (Kapitel 4.2) benötigt hier - grün dargestellt - am meisten Zeit, da das Zurückschreiben der Kräfte noch nicht sequentiell abläuft. Die Programme (Kapitel 4.3, 4.4), welche bidirektionale Verlet-Listen verwenden und somit das sequentielle Schreiben der Kräfte ermöglichen, erreichen einen Geschwindigkeitszuwachs von ca. 34% (Geschwindigkeitsfaktor: 1.5) gegenüber dem ersten Parallelisierungsansatz für das Abarbeiten der Buffer. Die quadratische Methode (Kapitel 4.5) benötigt am wenigsten Zeit für das Auslesen und Schreiben der Buffer, da die Buffer hier nur für einen geringen Teil der zu transferierenden Daten verwendet werden.

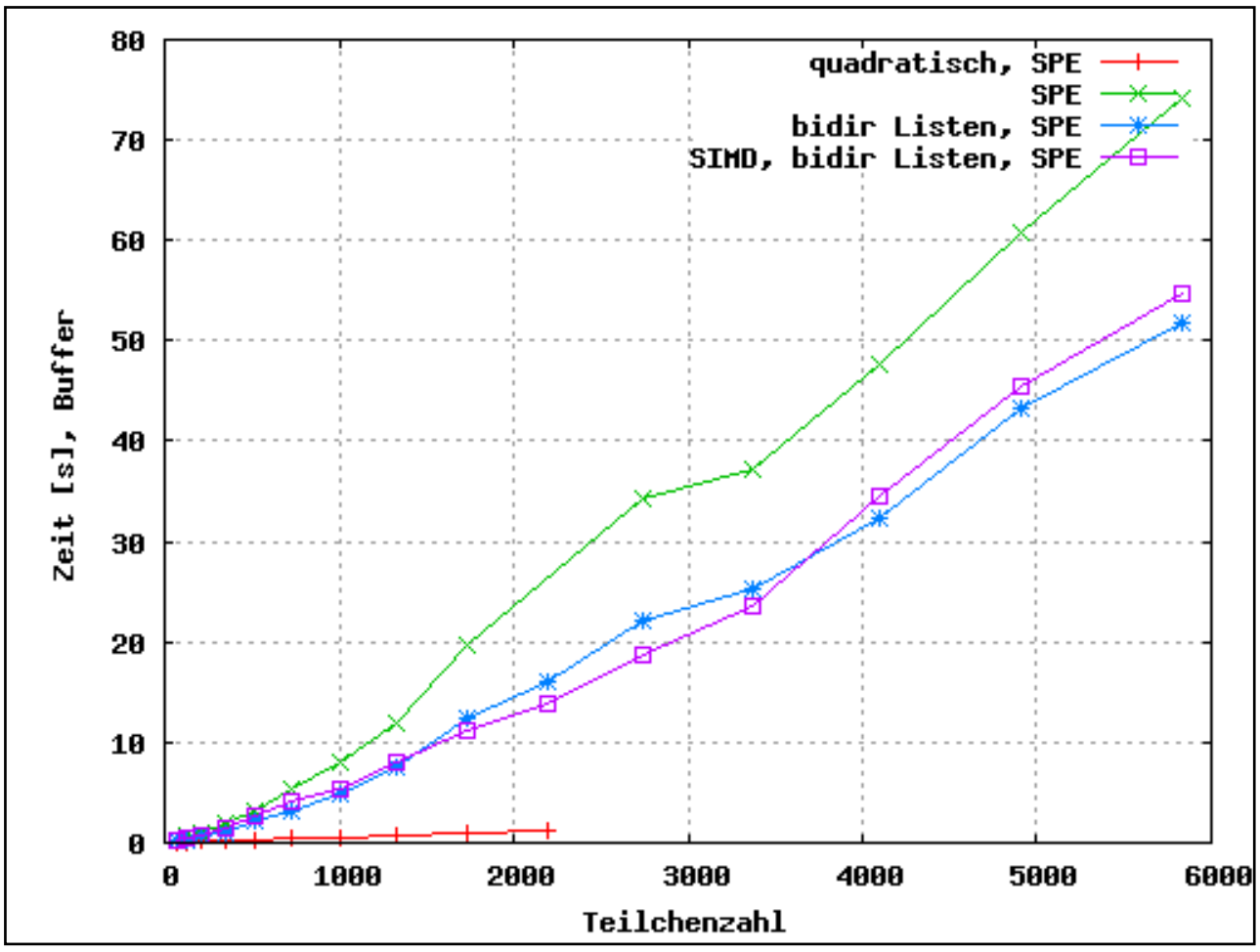

Abb. 5.4 - Vergleich: Buffer schreiben/auslesen

Die größten Unterschiede sind in der folgenden Abbildung 5.5 zu sehen. Es sind die Zeiten für die Kraftberechnung dargestellt. Bei der quadratischen Methode steigt die Zeit bereits bei kleinen Teilchenzahlen immer schneller an und steht in keinem Vergleich zu den anderen Methoden. Die Einzelprozessor-Variante benötigt im Vergleich zu den parallelisierten Ansätzen wesentlich mehr Zeit für die Kraftberechnung. Die Unterschiede zwischen den parallelisierten Programmen sind gering.

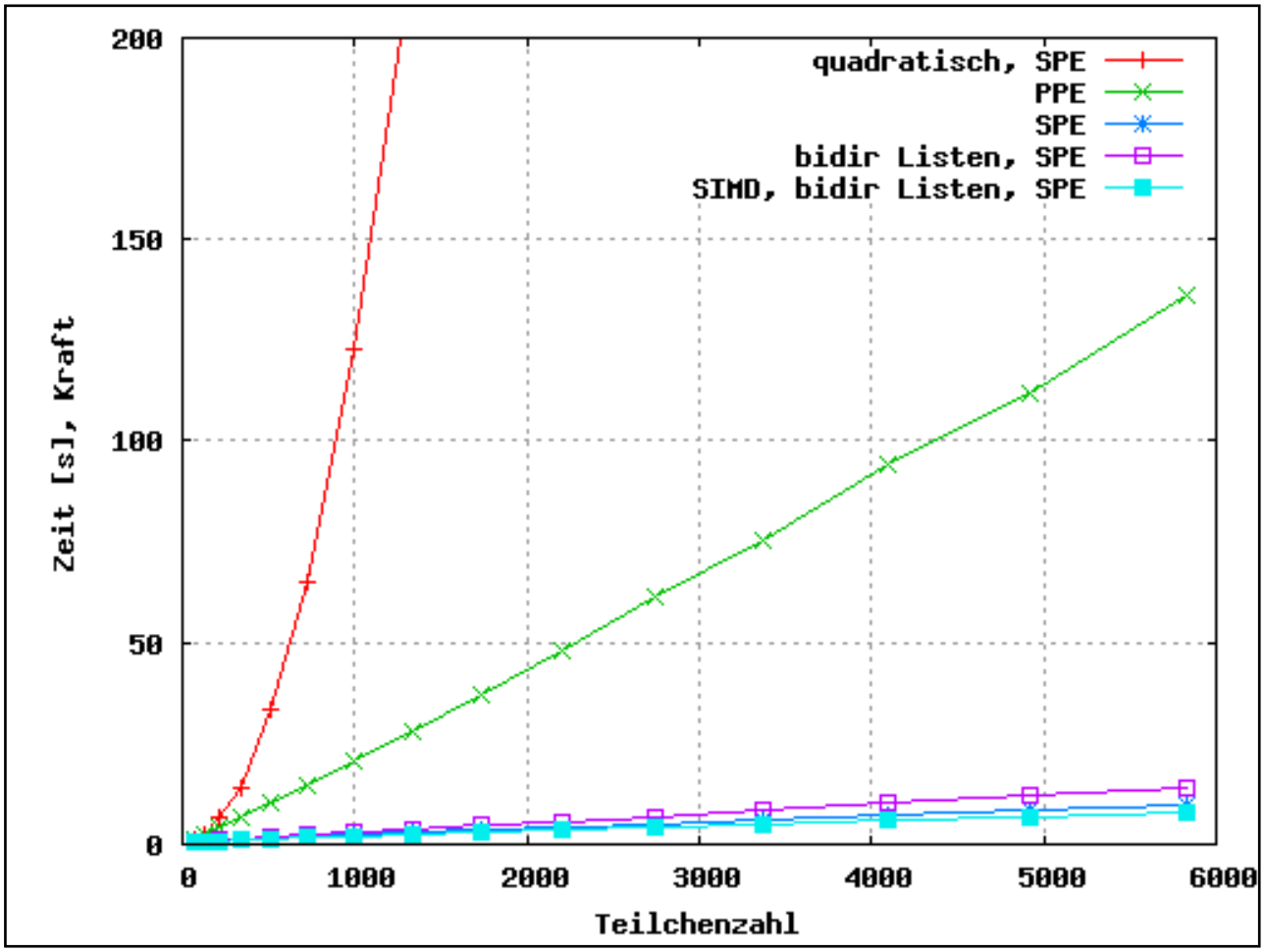

Abb. 5.5 - Vergleich: Kraftberechnung

6. Fazit

Wie der Vergleich der Methoden zeigt, ist es möglich einen Performancegewinn zu erzielen. Dieser fällt bisher allerdings gering aus und bezieht sich nur auf MD-Simulationen mit den in Kapitel 3.2 beschriebenen Eigenschaften. Daher gelten die Resultate grundsätzlich nur für Simulationen die kurzreichweitige Wechselwirkungen behandeln, eine Simulationsbox mit endlicher Größe verwenden und Teilchen mit einheitlichen Massen betrachten.

Die quadratische Programm-Variante benötigt die meiste Zeit und ist wesentlich langsamer als das Referenz-Programm, welches nur auf einer CPU - dem PPE - arbeitet. Für begrenzte Teilchenzahlen ist es aber möglich die Berechnungs-Laufzeit wesentlich zu verkürzen, worauf im Ausblick (Kapitel 7) kurz eingegangen wird.

Die in 4.2 behandelte Methode erzielt einen Geschwindigkeitszuwachs von 1.40 (ca. 27 %) gegenüber dem Einzelprozessor-Programm. Dieser Performancegewinn wurde durch die Parallelisierung der Kraftberechnung erreicht.

Das in Kapitel 4.3 vorgestellte Programm wurde gegenüber dem Programm aus Kapitel 4.2 so modifiziert, dass die berechneten Kräfte sequentiell zurückgeschrieben werden können, was einen Geschwindigkeitszuwachs von 1.46 (ca. 31%) gegenüber dem Referenz-Programm zur Folge hat. Das Problem, dass die Positionsdaten nicht sequentiell aus dem Speicher in die Buffer übertragen werden können, verhindert einen weiteren Performancegewinn in diesem Schritt.

Die im Durchschnitt schnellste Methode (Kapitel 4.4) verwendet die SIMD-Operationen der SPEs, was im Vergleich zu allen anderen Methoden zu einem weiteren, aber geringen, Geschwindikeitszuwachs führt. Der Zuwachs liegt bei 1.52 (ca. 33%) gegenüber dem Einzelprozessor-Programm.

Es muss festgestellt werden, dass die Multiprozessor-Varianten nicht wie erwartet skalieren. Man kann von einer theoretisch sechsfachen Geschwindigkeit ausgehen, wenn man die Anzahl der SPE-Kerne in einer PS3 berücksichtigt. Zudem ist jede SPE in der Lage vier Floatwerte zur gleichen Zeit zu berechnen. Somit wäre theoretisch zwischen dem Einzelprozessor-Programm und einer Variante, die alle SPEs nutzt und SIMD-Operationen verwendet, ein Geschwindigkeitszuwachs um das 6*4-fache zu erreichen.

Die folgende Grafik (Abb. 5.6) zeigt das Verhältnis der Laufzeit zwischen Einzelprozessor- und SIMD-Variante. Wie dort zu sehen ist, beträgt der durchschnittliche Geschwindigkeitszuwachs nur 1.5 (33%).

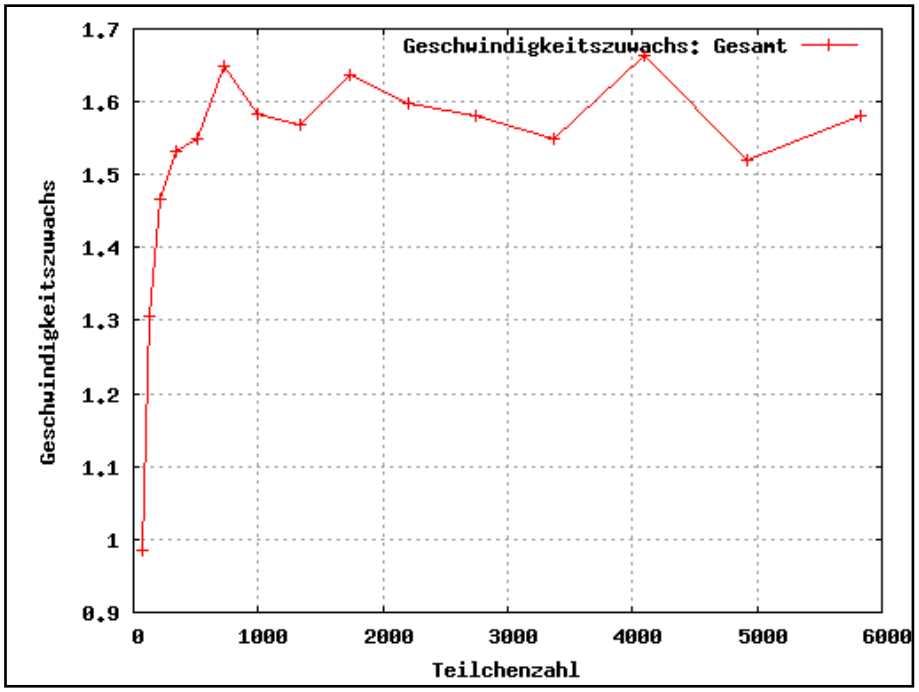

Abb. 5.6 - Geschwindigkeitszuwachs zwischen Einzelprozessor- und SIMD-Variante

In der folgenden Grafik (Abb. 5.7) ist das Verhältnis der Berechnungszeit für die Kraft zu sehen. Der Geschwindigkeitszuwachs beträgt hier durchschnittlich 11. Zudem ist zu sehen, dass mit steigender Teilchenzahl der Geschwindigkeitszuwachs 16 und bei größeren Teilchenzahlen mehr erreicht. Dies liegt wesentlich näher am theoretisch maximal erreichbaren Geschwindigkeitszuwachs von 24.

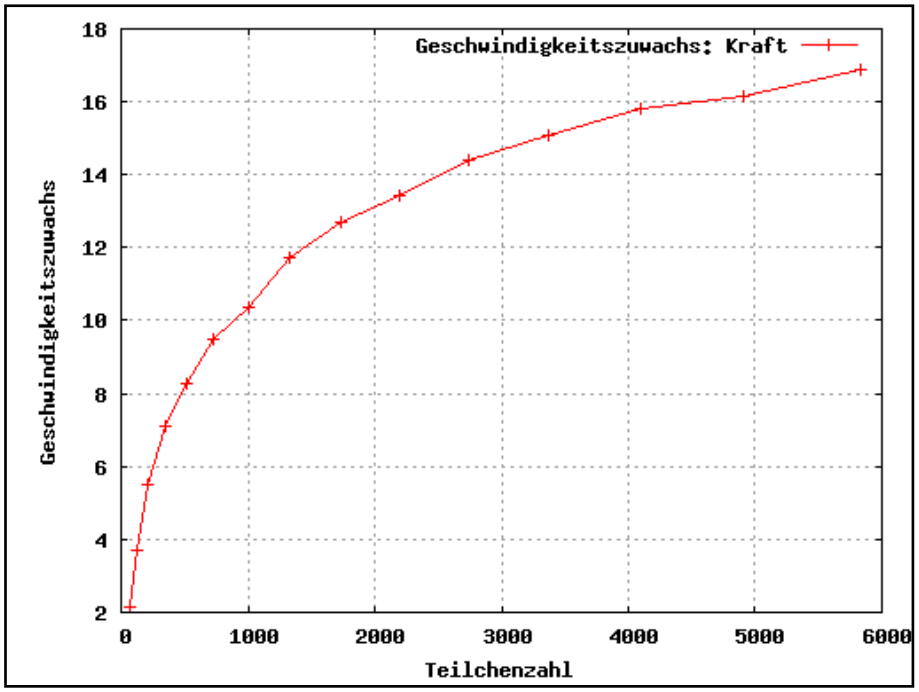

Abb. 5.7 - Geschwindigkeitszuwachs der Kraftberechnung

Der Engpass in der Berechnung ist bei allen Methoden 4.2 - 4.4 der Speicherzugriff. Besonders der nichtsequentielle Zugriff auf den Speicher beansprucht viel Zeit. Zudem sind die SPEs in geringem Maße ausgelastet. Abbildung 5.8 zeigt dies. Dort sind die über alle Methoden (4.2 - 4.4) gemittelten Anteile eines Integrationsschrittes in Prozent angegeben. Es ist zu sehen, dass die Erzeugung der Verlet-Listen und die Verwendung der Buffer den größten Anteil (zusammen ca. 80%) darstellen. Buffer und Verlet-Listen greifen auf nicht sequentiell im Speicher liegende Daten zu. Die restlichen Berechnungen stellen ca. 20% des Aufwandes dar und greifen auf sequentiell im Speicher liegende Daten. Die Krafberechnung stellt wiederum den kleineren Teil der 20% dar.

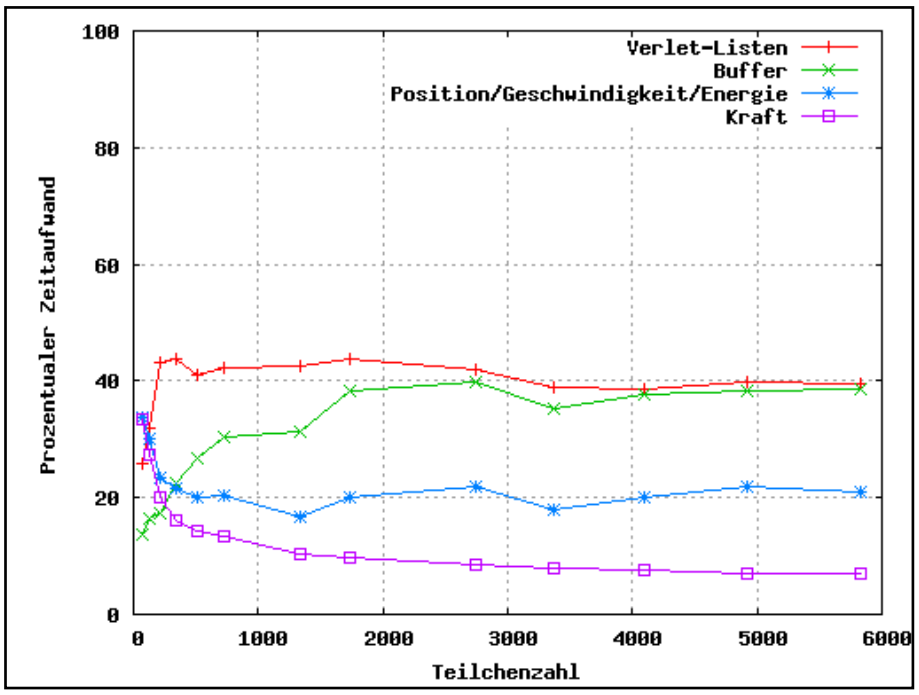

Abb. 5.8 - Durchschnittliche prozentuale Anteile der Gesamtlaufzeit

7. Ausblick

Im Folgenden wird kurz auf mögliche algorithmische Verbesserungen eingegangen, die in zukünftigen Arbeiten auf ihre Einsetzbarkeit, wie auch ihren Nutzen hin untersucht werden können.

Es gibt zwei Ansätze die verfolgt werden können um eine größere Performacesteigerung zu erzielen. Man kann versuchen die Unzulänglichkeiten bei den beschriebenen Methoden zu beseitigen und somit weiterhin bei der Atomic Decomposition bleiben. Oder man wendet die Domain Decomposition an.

Wenn Geschwindigungssteigerungen durch die Optimierung der behandelten Methoden angezielt werden, ist es wichtig das Verhältnis zwischen Speicherzugriffen und Berechnungsaufwand der SPEs zu minimieren. Entweder der Berechnungsaufwand der SPEs wird gesteigert, oder der aufwand für Speicherzugriffe muss verkleinert werden.
Da das PPE Berechnungsaufgaben wesentlich langsamer ausführt, als es mit den SPEs möglich ist, würde die Steigerung des Berechnungsaufwandes - zum Beispiel der Kraftberechnung durch mehrere verschiedene Potentiale - zu einer Vergrößerung des Abstandes der Berechnungsdauer zwischen der Einzelprozessor und der Multiprozessor-Variante führen. Wenn der Berechnungsaufwand der SPEs gesteigert werden soll, darf hierbei der Datenübertragungsaufwand zwischen PPE und SPEs nicht signifikant anwachsen.
Der Aufwand Kräfte aus dem Buffer zurückzuschreiben wurde bereits verkleinert. Das Füllen der Buffer mit den Positionsdaten hingegen geschieht in einer nicht sequentiellen Form. Durch das Umsortieren der Teilchenpositionen im Speicher zu bestimmten Zeitpunkten wäre es möglich, die Lokalität der Daten zu erhöhen und somit „Cache misses“ zu verringern. Hierzu bietet sich zum Beispiel die Verwendung eines Octtree zur Sortierung der Daten an.

Zu Beginn dieser Arbeit, gab es die Überlegung ein Domain Decomposition als Erweiterung der hier behandelten Methoden vorzuschlagen. So könnte die schnellste hier vorgestellte Methode (Atomic Decomposition) auf einer Menge CBEAs ausgeführt werden um jeweils einen Teil des Simulationssystems zu berechnen, während alle CBEAs zusammen das Gesamtsystem darstellen (Domain Decomposition). Dies ist nachwievor eine Möglichkeit.
Als Alternative zum Atomic Decomposition kann es aber bereits vorteilhaft sein eine Domain Decomposition auf CBEA-Ebene - auf einer CBEA - auszuführen. Hierbei würde jede SPE eine Menge an Unterboxen der Simulationsbox zur Verarbeitung zugewiesen bekommen. Jede der Unterboxen in die das Simulationssystem aufgeteilt ist, wird gerade so groß gewählt, dass sie in dem LS eines SPE Platz findet. So könnte eine hohe Lokalität der Daten für eine schnelle Übertragung zwischen PPE und SPE sorgen. Zudem kann ein SPE ohne zusätzlichen Datenaustausch andere Berechnungen wie die Positions-, Geschwindigkeits- und Energieberechnung zusätzlich übernehmen, was das PPE - wenn nötig - weiter entlastet.
Im Zusammenhang einer CBEA Domain Decomposition wäre weiter zu prüfen, ob sich hier das quadratische Abarbeiten bei der Kraftberechnung der Teilchen als vorteilhaft erweisen könnte.

Des Weiteren sind die Ergebnisse dieser Arbeit auf andere Architekturen übertragbar. Dies bezieht sich vor Allem auf Architekturen , die ein der CBEA ähnliches Layout besitzen. Es wäre zum Beispiel in Zukunft interessant, ob sich die hier vorgestellten Ansätze auf einen Intel Larrabee (noch in Entwicklung) übertragen lassen.

Zudem kann es von Interesse sein, welche hier getroffenen Einschränkungen bezüglich des zu simulierenden Systems aufgehoben werden können, und gleichzeitig eine möglichst hohe Performace zu erhalten.

Teilchen pro Achse	Gesamt-laufzeit	Verlet-Listen-Update	Positions-Update	Geschw.-Update 1	Kräfte auf 0 setzen	Kräfte berechnen	Geschw.-Update 2 + Berechnung der kinetischen Energie
4	1.853537	0.443375	0.121032	0.027242	0.009987	1.191210	0.060691
5	3.512175	0.750837	0.233201	0.050329	0.017992	2.343842	0.115974
6	6.676718	1.791895	0.403555	0.085609	0.030296	4.164407	0.200956
7	10.392243	2.565327	0.643923	0.136667	0.047573	6.671073	0.327680
8	15.660926	3.728525	1.024049	0.246613	0.070396	10.084489	0.506854
9	22.304190	5.096497	1.509188	0.387964	0.099142	14.476495	0.734904
10	31.302867	7.057655	2.019950	0.532710	0.135705	20.532926	1.023921
11	43.847289	10.982097	2.759287	0.711982	0.180037	27.821984	1.391902
12	58.339459	14.409832	3.495049	0.931084	0.242731	37.183956	2.076807
13	74.105651	17.590189	4.560249	1.170739	0.297234	47.911788	2.575452
14	94.494833	22.471958	5.592533	1.487419	0.380428	61.189005	3.373490
15	118.090476	27.880397	7.461147	1.922019	0.571384	75.597440	4.658089
16	153.204497	35.631412	10.531271	3.962213	2.533504	94.056185	6.489912
17	182.748020	45.093024	11.786575	3.984855	2.150976	111.650424	8.082166
18	226.120701	54.971980	15.177477	5.941163	3.988440	136.283249	9.758392

Tabelle 1: Laufzeiten Einzelprozessor

Teilchen pro Achse	Gesamt-laufzeit	Verlet-Listen-Update	Positions-Update	Geschw.-Update 1	Positionen in Caches schreiben	Kräfte berechnen	Potentielle Energien empfangen	Kräfte aus Caches auslesen	Geschw.-Update 2 + Berechnung der kinetischen Energie
4	2.161312	0.507713	0.104349	0.026444	0.123149	0.671780	0.467430	0.211829	0.048618
5	3.074829	0.878496	0.199539	0.047577	0.217078	0.773446	0.473273	0.393870	0.091550
6	4.907899	1.894663	0.332308	0.082708	0.392180	0.909494	0.465593	0.672582	0.158371
7	7.400497	2.951969	0.565767	0.138102	0.810593	1.069039	0.460016	1.134818	0.270193
8	10.940578	4.250590	0.911382	0.231763	1.395158	1.342948	0.478109	1.869920	0.460708
9	15.821684	5.889243	1.322909	0.333649	2.294994	1.818907	0.482725	3.037226	0.642031
10	22.199492	8.207614	1.809194	0.471116	3.619499	2.261264	0.468584	4.462221	0.900000
11	32.209340	12.601566	2.531669	0.607864	5.034269	2.737520	0.481217	6.858537	1.356698
12	48.946504	17.011209	4.112168	1.715506	8.487725	3.395699	0.471349	11.228074	2.524774
14	82.881955	27.806243	7.247421	3.650874	14.626680	4.896742	0.477173	19.671577	4.505245
15	88.281433	31.321351	6.970169	2.150652	16.131754	5.932503	0.478099	20.954266	4.342639
16	111.196904	38.309647	8.524511	3.149041	20.278177	7.116524	0.481077	27.478748	5.859179
17	145.060063	50.144588	12.045188	5.431841	25.941490	8.453754	0.477766	34.920544	7.644892
18	174.479206	60.311112	14.225454	6.324303	31.225146	9.839575	0.488890	42.873198	9.191528

Tabelle 2: Laufzeiten SPE +Verlet-Listen

Teilchen pro Achse	Gesamt-laufzeit	Verlet-Listen-Update	Positions-Update	Geschw.-Update 1	Positionen in Caches schreiben	Kräfte berechnen	Potentielle Energien empfangen	Kräfte aus Caches auslesen	Geschw.-Update 2 + Berechnung der kinetischen Energie
4	2.082176	0.497867	0.115337	0.028148	0.166190	0.705108	0.464735	0.055789	0.049002
5	2.881498	0.864484	0.221360	0.054151	0.289703	0.828635	0.464521	0.067485	0.091159
6	4.730771	1.976531	0.390955	0.087984	0.517207	1.026200	0.467467	0.111765	0.152662
7	6.651635	2.863941	0.641447	0.149393	0.881716	1.232353	0.470233	0.170008	0.242544
8	10.377611	4.190927	1.015007	0.248809	1.932171	1.797950	0.469988	0.336590	0.386169
9	14.336708	5.913831	1.472025	0.374810	2.804832	2.275148	0.474203	0.438808	0.583051
10	19.809159	7.967745	2.051828	0.524619	4.279344	2.960577	0.470688	0.681852	0.872506
11	29.014095	12.494148	2.720281	0.682060	6.698299	3.875579	0.470512	0.895750	1.177466
12	41.990539	16.831245	4.103081	1.423371	10.717253	4.735155	0.480378	1.724961	1.975095
13	52.851637	21.728704	5.257964	1.589274	13.912938	5.749850	0.481326	2.095571	2.036010
14	72.100837	27.187508	7.678205	3.443326	18.895645	6.962495	0.478132	3.118531	4.336995
15	83.632222	32.856706	8.620226	3.552947	21.679309	8.503513	0.499013	3.531857	4.388651
16	104.111029	39.874379	10.699918	4.615930	28.020015	10.226517	0.491306	4.430773	5.752191
17	138.615645	52.798468	14.358779	7.155523	37.552416	12.046772	0.487645	5.714084	8.501958
18	165.235712	63.499749	16.925314	8.458865	44.933764	14.159636	0.485753	6.742948	10.029683

Tabelle 3: Laufzeiten SPE +Verlet-Listen+Bidirektional

Teilchen pro Achse	Gesamt-laufzeit	Verlet-Listen-Update	Positions-Update	Geschw.-Update 1	Positionen in Caches schreiben	Kräfte berechnen	Potentielle Energien empfangen	Kräfte aus Caches auslesen	Geschw.-Update 2 + Berechnung der kinetischen Energie
4	1.884872	0.478766	0.096009	0.025862	0.170542	0.551719	0.460308	0.052269	0.049397
5	2.658694	0.848369	0.191111	0.050177	0.308894	0.630033	0.460944	0.076542	0.092624
6	4.426376	1.958454	0.327092	0.080952	0.547701	0.759734	0.475569	0.120316	0.156558
7	6.723839	2.911733	0.535477	0.136024	1.263888	0.941620	0.467155	0.217146	0.250796
8	10.040935	4.152977	0.847314	0.222837	2.354439	1.219242	0.468178	0.324571	0.451377
9	14.212690	5.850242	1.269469	0.332344	3.628272	1.528189	0.482074	0.456336	0.665764
10	18.597208	7.750035	1.662948	0.467356	4.789291	1.983260	0.458435	0.656985	0.828898
11	27.118985	12.134028	2.284019	0.604502	7.021209	2.376036	0.475149	1.035057	1.188985
12	35.226373	15.424848	2.933667	0.791699	9.801303	2.922699	0.469418	1.347703	1.535036
13	44.262869	19.408324	3.844577	1.028940	12.403086	3.569328	0.475682	1.542547	1.990385
14	55.353778	23.451670	4.739363	1.265804	16.394835	4.246129	0.471733	2.253467	2.530777
15	71.243113	30.038328	6.444763	2.231363	20.698315	5.007978	0.480114	2.981932	3.360320
16	99.194552	38.103194	9.788124	4.546743	29.985514	5.944658	0.479689	4.480019	5.866611
17	131.808743	51.428433	12.778155	6.542905	39.693698	6.910884	0.490034	5.670690	8.293944
18	153.009175	59.058605	14.216465	7.271177	48.079670	8.075266	0.487685	6.515809	9.304498

Tabelle 4: Laufzeiten SPE +Verlet-Listen+Bidirektional+SIMD

Teilchen pro Achse	Gesamt-laufzeit	Temp anpassen	Positions-Update	Geschw.-Update 1	Positionen in Caches schreiben	Kräfte berechnen	Potentielle Energien empfangen	Kräfte aus Caches auslesen	Geschw.-Update 2 + Berechnung der kinetischen Energie
4	1.786422	0.003112	0.140868	0.024934	0.031664	1.009587	0.462815	0.051111	0.062331
5	3.538303	0.003149	0.271720	0.045877	0.039790	2.522358	0.465049	0.071080	0.119280
6	7.899763	0.003092	0.465276	0.077159	0.052302	6.516007	0.472409	0.108778	0.204740
7	17.058561	0.003077	0.741986	0.120860	0.070421	14.244811	1.394638	0.158080	0.324688
8	37.458506	0.003275	1.105051	0.180486	0.094333	33.255141	2.114965	0.218130	0.487125
9	69.833852	0.003171	1.556269	0.263020	0.134907	65.087910	1.755470	0.336663	0.696442
10	128.262546	0.003232	2.122016	0.379908	0.188758	122.573102	1.623913	0.406398	0.965219
11	223.241860	0.003284	2.857223	0.505493	0.250032	216.723548	1.082111	0.537856	1.282313
12	372.492170	0.003331	3.783099	0.662187	0.324691	364.753459	0.561947	0.686497	1.716959
13	598.945950	0.004062	4.839751	0.840519	0.398598	581.641280	8.127320	0.866512	2.227908

Tabelle 5: Laufzeiten SPE +Quadratisch+Bidirektional+SIMD

Anhang B - Abkürzungsverzeichnis

AoS - Array of Structures
BEI - Cell Broadband Engine Interface
BE - Broadband Engine
CBE - Cell Broadband Engine
CBEA - Cell Broadband Engine Architecture
DMA - Direct Memory Access
EIB - Element Interconnect Bus
GHz - Gigahertz
I/O - Input/Output
KB - Kilobyte
L1/L2 - Level 1/2
LS - Local Store
MB - Megabyte
MD - Molekulardynamik
MFC - Memory Flow Controller
MIC - Memory Interface Controller
PPE - Power Processor Element
PS3 - PlayStation 3
RAM - Random Access Memory
RISC - Reduced Instruction Set Computing
SIMD - Single Instruction Multiple Data
SoA - Structure of Arrays
SDK - Software Development Kit
SPE - Synergistic Processing Element
SPU - Synergistic Processing Unit
STI - Sony Toshiba IBM
YDL - Yellow Dog Linux

Anhang C - Quellenverzeichnis

[ALLEN89] Allen, Tildesley - „Computer Simulation of Liquids“
Oxford University Press - Neuauflage - 1989
a: S. 81 f.
b: S. 28 f.
c: S. 29

[FRENKEL02] Frenkel, Smit - „Understanding Molecular Simulation“ - „From Algorithms to Applications“
Academic Press - 2. Ausgabe - 2002
a: S. 75 ff.
b: S. 545 f.

[IBM06] IBM - „SPE Runtime Management Library“
Version 2.0 - 11.11.2006
Zu finden im CBE SDK unter http://www.ibm.com/developerworks/power/cell/, Kategorie: „Downloads“

[IBM07] IBM - „Cell Broadband Engine Programming Handbook“
Version 1.1 - 24.04.2007
S. 41 f.
Zu finden im CBE SDK unter http://www.ibm.com/developerworks/power/cell/, Kategorie: „Downloads“

[IBM08] IBM - „C/C++ Language Extensions for Cell Broadband Engine Architecture“
Version 2.5 - 27.02.2008
Zu finden im CBE SDK unter http://www.ibm.com/developerworks/power/cell/, Kategorie: „Downloads“