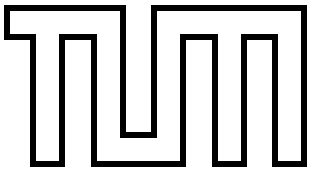


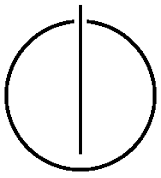
FAKULTÄT FÜR INFORMATIK

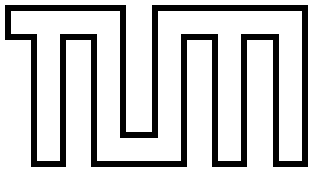
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Effiziente Vektorisierung von Simulationen für starre mehrzentrige Molekülmodelle

Johannes Heckl





FAKULTÄT FÜR INFORMATIK

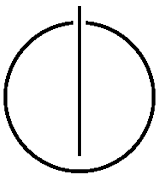
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Effiziente Vektorisierung von Simulationen für starre mehrzentrige Molekülmodelle

Efficient Vectorization of rigid-body multi-centered Molecular Dynamics Simulations

Bearbeiter:	Johannes Heckl
Aufgabensteller:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Betreuer:	Dipl.-Inf. Wolfgang Eckhardt
	Alexander Heinecke, M.Sc., M.Sc. with honors
Abgabedatum:	16. Juli 2012



Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 16. Juli 2012 Johannes Heckl

Zusammenfassung

Molekulardynamiksimulationen sind ein wichtiges Hilfsmittel der Forschung. Simulationen mit vielen komplexen, mehrzentrigen Molekülen erfordern sehr lange Laufzeiten, sodass stets schnellere Implementierungen gesucht werden. Moderne Supercomputer bestehen oft aus vielen Standardprozessoren, die Vektor-Befehlssätze wie SSE und AVX anbieten. Für Molekulardynamiksimulationen, die den Linked-Cell Algorithmus verwenden, ist die Optimierung mittels dieser Instruktionen jedoch schwierig.

Ein kürzlich entwickelter Algorithmus erlaubt die effiziente Vektorisierung der Kraftberechnung in Simulationen mit einzentrigen Molekülmodellen mit Hilfe von Vektorinstruktionen. Die Moleküldaten werden zuerst in eine vektorisierbare Form gebracht. Die so verbesserte Lokalität der Daten und die Verwendung von Vektorinstruktionen ermöglicht nun einen hohen Laufzeitvorteil. Die Erweiterung dieses Algorithmus auf mehrzentrige Moleküle und die Integration in das Simulationsprogramm MarDyn sind Thema dieser Arbeit.

Laufzeitanalysen für einzentrige Molekülmodelle zeigen einen Leistungsvorteil von bis zu 125% gegenüber der ursprünglichen Version von MarDyn auf. Für mehrzentrige Molekülmodelle hingegen kann nur eine geringe Leistungssteigerung erzielt werden. Da die Zentren wie einzentrige Moleküle behandelt werden, werden redundante Abstandsberechnungen durchgeführt, die zu einem Leistungsverlust führen.

Abstract

Molecular dynamics simulations are an important tool in research. Simulations with many complex, multi-centered molecules require very long runtimes. Thus there always is demand for faster implementations. Modern supercomputers frequently consist of many standard processors, which support vector instruction sets like SSE and AVX. However, optimizing a molecular dynamics simulation based on a linked cells algorithm using these instruction is difficult.

A recently developed algorithm describes the efficient vectorization of the force calculation in simulations of single-centered molecules using vector instructions. First, molecule data is rearranged into a vectorizable shape. The resulting improved locality of molecule data coupled with vectorized force calculation leads to greatly improved performance. Topic of this thesis is the extension of the algorithm to support multi-centered molecule models and the integration into the simulation program MarDyn.

Performance analysis for single-centered molecules shows a gain of up to 125% compared to the original version of MarDyn. For multi-centered molecules, only a small performance gain can be measured. Because each center is treated similarly to a single-centered molecule, redundant distance calculations are performed, which leads to a loss of performance.

Inhaltsverzeichnis

1. Einleitung

Bereits seit mehreren Jahrzehnten werden Molekulardynamiksimulationen verwendet, um wissenschaftliche Erkenntnisse zu erlangen. Molekulardynamiksimulationen haben das Potential, gefährliche, teure oder in der Realität nicht durchführbare Experimente zu simulieren. Reale Experimente können nachgestellt und im Detail analysiert werden. Mittlerweile werden solche Simulationen in großem Umfang in der Chemie, Biologie, Physik und anderen Bereichen durchgeführt.

Eine Molekulardynamiksimulation ist ein Programm, das eine Menge von Molekülen in einem gewissen Raumvolumen simuliert. Dazu wird zunächst die Zeit diskretisiert. In jedem Zeitschritt werden die Kräfte zwischen den Molekülen berechnet. Dann werden mit einem Integrationsverfahren die Molekülattribute für den nächsten Zeitschritt berechnet. Die Kraftberechnung ist der aufwändigste Teil einer Molekulardynamiksimulation. Szenarien können je nach Anwendung wenige hundert bis hin zu mehreren Millionen Molekülen enthalten. Für eine realistische Approximation realer Moleküle werden Zeitschritte im Bereich von 1 fs benötigt. Selbst mit modernsten Rechnern und Implementierungen können unter diesen Umständen viele reale Experimente nicht mit vertretbarem Zeitaufwand simuliert werden.

Aussagekräftige Molekulardynamiksimulationen in kurzer Zeit durchzuführen ist daher Ziel vieler Anstrengungen. Es existiert Hardware, die speziell für die Molekulardynamiksimulation entwickelt wurde. Hochleistungsrechner wie *Anton* [6] verwenden anwendungsspezifische integrierte Schaltkreise um die Kraftberechnung schnell und parallel durchführen zu können. Diese Rechner sind jedoch nicht flexibel und durch die spezielle Hardware teuer. Weiterhin wurden Ansätze zur Ausnutzung der hohen Parallelität, die auf modernen Grafikkarten verfügbar ist, unternommen [13]. Es existieren jedoch bisher nur wenige Hochleistungsrechner, die die benötigte Rechenleistung auf Grafikkarten zur Verfügung stellen. Für Vektorprozessoren wie dem Cray X1 existieren effiziente Implementierungen [15]. Diese können aber nicht mit modernen Rechnerarchitekturen konkurrieren.

Die meisten modernen Hochleistungsrechner bestehen aus vielen Standardprozessoren, die parallel arbeiten. Jeder dieser Prozessoren besteht aus mehreren Rechenkernen, die auf einen gemeinsamen Speicher zugreifen. Bei der Shared Memory Parallelisierung arbeitet ein Thread des selben Prozesses auf jedem der Kerne eines Prozessors und alle Threads greifen auf die selben Daten zu. Diese Threads werden dazu verwendet, die Kraftberechnung parallel auszuführen. Dabei muss jedoch darauf geachtet werden, dass die Daten nicht durch gleichzeitigen Zugriff mehrerer Threads fehlerhaft werden. Das Message Passing Interface (MPI) verfolgt einen anderen Ansatz der Parallelisierung. Hier wird der Si-

mulationsraum in disjunkte Teilbereiche zerlegt. Jeder Kern erhält einen eigenen Prozess, der die Simulation für einen dieser Teilbereiche durchführt. Nach jedem Zeitschritt werden die Prozesse synchronisiert. MPI- und Shared Memory Parallelisierung können natürlich auch kombiniert werden. Dann wird der Simulationsraum auf die Prozessoren aufgeteilt, während die Prozessoren ihre Arbeitsbelastung mittels Shared Memory Parallelisierung auf die Rechenkerne verteilen.

Eine weitere Optimierungsmöglichkeit auf Standardprozessoren bietet sich durch die seit einigen Jahren verfügbaren Vektorinstruktionen wie SSE und AVX. Diese erlauben die parallele Ausführung einzelner Rechenoperationen innerhalb eines Threads für eine geringe Anzahl von Daten. Ein anderer Ansatz ist es, die Kraftberechnung durch eine Tabelle zu ersetzen, aus der basierend auf dem Abstand zweier Moleküle die Kraft zwischen den Molekülen gelesen und interpoliert wird [14][17].

Ein Großteil der Simulationssoftware in der Molekulardynamik verwendet vorrangig kurzreichweitige Potentiale um die Kräfte zwischen den Molekülen zu simulieren. Man nutzt aus, dass diese Potentiale ab einem gewissen Abstand zwischen den Molekülen weitestgehend vernachlässigbar sind, indem man nur die Kräfte bis zu einem gewissen Abstand der Moleküle berechnet. Nun kann man beispielweise mit Hilfe des Linked-Cells Algorithmus [10] die Kraftberechnung so implementieren, dass nur noch wenige Molekülpaare für die Berechnung betrachtet werden müssen. Dazu wird der Raum in sogenannte Zellen partitioniert. Aufgrund der Zellstruktur können nun potentiell benachbarte Moleküle effizient gefunden werden. Dies verringert die Komplexität der Kraftberechnung von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n)$, wobei n die Anzahl der Moleküle in der Simulation ist.

In dieser Arbeit wird eine Optimierung für Molekulardynamiksimulationen auf Standardprozessoren mittels Vektorinstruktionen in das Simulationsprogramm MarDyn integriert und analysiert. Dazu werden vor der Kraftberechnung die Moleküldaten so in eine spezielle Datenstruktur umsortiert, dass eine effiziente Vektorisierung ermöglicht wird.

In Kapitel 2 werden die Grundlagen erörtert. Zunächst wird die Funktionsweise von Vektorinstruktionen erläutert. Es folgt eine Beschreibung des Algorithmus, mit dem die Kraftberechnung beschleunigt werden soll. Schließlich wird auf die relevanten Aspekte des Programms MarDyn eingegangen, in das der Algorithmus integriert werden soll.

In Kapitel 3 wird die Implementierung des Algorithmus erklärt. Dazu ist eine Veränderung am Softwaredesign von MarDyn notwendig. Nachdem diese im Detail dargelegt wurde, folgt die Beschreibung der Klasse *VectorizedLJCellProcessor*, welche die vektorisierte Kraftberechnung implementiert. Es wird noch die Implementierung einer weiteren Klasse *LJFlopCounter* erläutert, die genauere Laufzeitanalysen erlaubt.

In Kapitel 4 werden die Programme und Szenarien vorgestellt, die analysiert wurden. Die ursprüngliche Version von MarDyn wird mit verschiedenen neu erstellten Konfigurationen für Szenarien mit unterschiedlicher Komplexität verglichen. Die Ergebnisse für ein- und mehrzentrige Molekülmodelle werden präsentiert und interpretiert.

2. Grundlagen

In diesem Kapitel werden die für diese Arbeit verwendeten Grundlagen beschrieben. Es wird zunächst auf die Vektorisierungsmöglichkeiten moderner Standardprozessoren eingegangen. Dann wird ein Algorithmus vorgestellt, der die effiziente Vektorisierung der Kraftberechnung auf Standardprozessoren realisiert. Schließlich wird das Programm MarDyn beschrieben, in das dieser Algorithmus integriert werden soll.

2.1. Vektorinstruktionen

Moderne Hochleistungsrechner bestehen häufig aus vielen parallel arbeitenden Standardprozessoren wie dem Intel Nehalem Xeon Prozessor. In den letzten Jahren wurden die Befehlssätze dieser Prozessoren um Vektorinstruktionen erweitert, die das Single Instruction Multiple Data (SIMD) Konzept von Vektorrechnern aufgreifen. Vektorrechner können mittels paralleler Hardwareeinheiten eine Operation gleichzeitig auf sehr vielen Daten ausführen. Dies ist immer dann hilfreich, wenn die gleiche Berechnung auf vielen unabhängigen Datensätzen durchgeführt werden muss. Standardprozessoren erlauben nun auch die gleichzeitige Berechnung mehrerer Werte auf einem einzelnen Rechenkern. Dies ist möglich, da moderne Prozessoren über Register verfügen, die 128 oder 256 bit breit sind. In diesen Registern können jeweils meherere Daten abgelegt werden und mit Hife von speziellen Instruktionen, die auf dem gesamten Register arbeiten, gemeinsam bearbeitet werden. In einem 128 bit Register können beispielweise vier single precision oder zwei double precision floating point Werte gespeichert werden. Damit können im Optimalfall doppelt so viele double precision Berechnungen durchgeführt werden, als dies ohne Vektorinstruktionen der Fall wäre. Beschränkt man sich auf single precision, sind sogar viermal so viele Rechenoperationen möglich. In Abbildung 2.1 ist das Schema der Addition zweier Register mit je zwei double precision Werten dargestellt.

Der Befehlssatz SSE (Streaming SIMD Extensions) erlaubt es, Berechnungen auf 128 bit Registern durchzuführen. Der Befehlssatz AVX (Advanced Vector Extensions) dient der Benutzung von 256 bit Registern. Mit Hilfe sogenannter Intrinsics [11] können Vektorinstruktionen direkt in den Programmcode integriert werden. Sie werden von den meisten gängigen Compilern unterstützt. Ein Intrinsic wird wie eine normale Funktion verwendet und in der Regel von Compiler direkt in die entsprechende Assembler-Instruktion übersetzt. Diese Art der Programmierung ist sehr hardwarenah. Der wesentliche Unterschied zur Programmierung in Assembler ist, dass der Compiler und nicht der Programmierer die Belegung der Register verwaltet. Falls es dem Compiler erlaubt wird, kann er in manchen Fällen auch eine automatische Vektorisierung durchführen. Dazu erkennt er gewisse Strukturen im Code, die mittels Vektorinstruktionen effizienter durchgeführt werden können. Diese Methode ist jedoch nur in relativ einfachen Berechnungen erfolgreich.

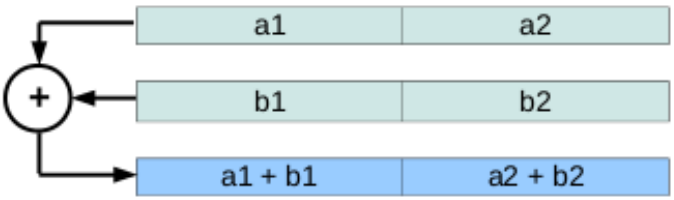


Abbildung 2.1.: Schema einer vektorisierten Addition zweier Register mit je zwei Werten.

Ein einfaches Beispiel für die Anwendung von Vektorinstruktionen ist die Addition der Einträge zweier Arrays gleicher Länge:

1: $ArrayA, B, C$
2: $i \leftarrow 0$
3: **while** $i < A.length$ **do**
4: $\quad C[i] \leftarrow A[i] + B[i]$
5: $\quad i \leftarrow i + 1$

Die einfache Schleife kann ersetzt werden durch eine Schleife, in der mittels Vektorinstruktionen in jedem Schritt mehrere Einträge bearbeitet werden:

1: $ArrayA, B, C$
2: $i \leftarrow 0$
3: **while** $i < A.length$ **do**
4: $\quad (C[i], C[i+1]) \leftarrow (A[i], A[i+1]) + (B[i], B[i+1])$
5: $\quad i \leftarrow i + 2$

Es muss allerdings beachtet werden, dass einige Elemente am Ende der Schleife übrig bleiben, wenn die Länge der Arrays kein Vielfaches der Anzahl Elemente ist, die pro Schritt bearbeitet werden. In diesem Fall kann beispielsweise noch einmal die nichtvektorisierte Schleife für die verbliebenen Elemente aufgerufen werden.

Vektorinstruktionen sind besonders effizient, wenn die benötigten Daten im Arbeitsspeicher aufeinanderfolgen, so dass sie gemeinsam mit einer einzelnen Operation in ein Register geladen werden können. Wenn eine Vektorinstruktion auf mehrere Datensätze angewendet werden soll, so ist es vorteilhaft, wenn auch diese Datensätze direkt aufeinanderfolgen. Dies erlaubt eine effiziente Verwendung der Caches im Prozessor.

2.2. Effiziente Vektorisierung in der Molekulardynamik

Eine intuitive Anwendung von Vektorinstruktionen in Molekulardynamiksimulationen ist bei der Berechnungen von mehrdimensionalen Werten wie der Position eines Moleküls möglich. Man kann zwei der drei Koordinaten gleichzeitig bearbeiten, da die meisten Rechenoperationen auf alle Koordinaten gleich angewendet werden. Leider ist dieser Ansatz nicht sehr effektiv, da jeweils drei Koordinaten existieren, Vektorinstruktionen aber auf zwei oder vier Werten arbeiten. Zudem fallen auch einige Berechnungen auf skalaren Werten an, die auf diese Weise nicht vektorisiert werden können. Solche Optimierungen

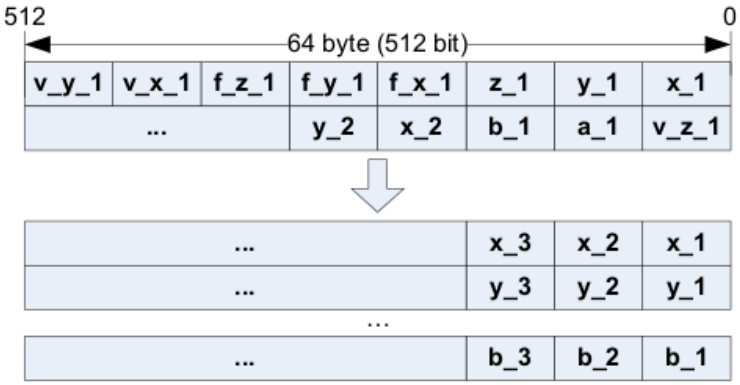


Abbildung 2.2.: Kopieren einiger Moleküldaten von einem *Array of Structures* in eine *Structure of Arrays*. Grafik aus [7].

können zumeist vom Compiler automatisch erkannt und umgesetzt werden, schöpfen das volle Potential der Vektorisierung jedoch nicht immer aus. Oft kann eine Berechnung aber vektorisiert werden, indem Datenparallelität ausgenutzt wird, die der Compiler nicht erkennt.

Aus diesem Grund haben Wolfgang Eckhardt und Alexander Heinecke einen anderen Ansatz erarbeitet [7]. In ihrer Arbeit beschreiben sie eine Implementierung, die zwar die Moleküle wie auch in MarDyn zunächst in einer Liste innerhalb einer Zelle speichert, die aber vor der Bearbeitung der Molekülpaare eine andere Datenstruktur erstellt, die eine effiziente Vektorisierung erlaubt.

Der Algorithmus basiert auf einer Molekulardynamiksimulation mit Linked Cells Algorithmus. Die Moleküle werden in den Zellen als Liste gespeichert, dies wird als *Array of Structures* bezeichnet. Vor der eigentlichen Kraftberechnung werden die Daten der Moleküle jedoch in eine *Structure of Arrays* kopiert, wie in Abbildung 2.2 dargestellt. Diese Datenstruktur enthält ein Array pro Datenelement der Molekülklasse, das für die Kraftberechnung relevant ist. Dabei entspricht das n-te Element eines Arrays dem korrespondierenden Datenelement des n-ten Moleküls in der Zelle. Während alle Daten eines Moleküls im *Array of Structures* mehr oder weniger gruppiert sind, sind sie in der *Structure of Arrays* über verschiedene Arrays verteilt. Umgekehrt sind aber alle Daten eines Typs (wie zum Beispiel eine x-Koordinate) im *Array of Structures* über den Arbeitsspeicher verteilt, wohingegen sie bei der *Structure of Arrays* alle in einem Array zu finden sind. Diese Eigenschaft ist für die Vektorisierung vorteilhaft, da auf alle Daten eines Typs die gleichen Operationen angewendet werden. Zudem sind die Daten nun so im Arbeitsspeicher plaziert, dass Daten, die nacheinander benötigt werden, auch nacheinander im Speicher liegen. Dadurch können Caches im Prozessor besser ausgenutzt werden.

Die Kraftberechnung wird mit SSE3 Intrinsics vektorisiert, indem jeweils die (double precision) Daten eines Moleküls in einem Register dupliziert werden und mit den Daten zweier anderer Moleküle gleichzeitig bearbeitet werden. Dadurch können die Kräfte für

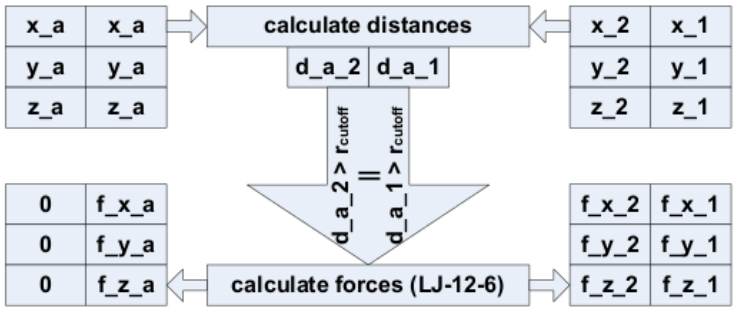


Abbildung 2.3.: Vektorisierung der Kraftberechnung. Grafik aus [7].

doppelt so viele Molekülpaare berechnet werden als ohne SSE3. Nach der Berechnung müssen die Ergebnisse wieder in das *Array of Structures* übertragen werden. Abbildung 2.3 stellt das Schema dieser Berechnung dar.

In jedem Durchlauf der Kraftberechnung werden zwei Molkülpaare bearbeitet. Es wird jedoch nicht garantiert, dass für beide tatsächlich eine Kraftberechnung durchgeführt werden muss. Wenn nur ein Paar nahe genug beisammen ist, um eine Berechnung erforderlich zu machen, wird die Berechnung dennoch für beide Paare durchgeführt. Das Ergebnis wird aber mit Hilfe einer Maske, die sich aus der Abstandsberechnung ergibt, für das Paar, dessen Moleküle zu weit voneinander entfernt sind, wieder verworfen.

Die Implementierung und Auswertung dieses Ansatzes erfolgte in einer relativ einfachen Molekulardynamiksimulation. Es wurden ausschließlich einzentrige Moleküle mit Interaktionen durch das Lennard-Jones-Potential betrachtet. Die *Structure of Arrays* Version mit SSE3 erreichte im Vergleich zur Version ohne SSE3 die doppelte Leistung. Im Vergleich zu einer Implementierung, bei der die Berechnung nichtvektorisiert auf dem *Array of Structures* stattfindet, konnte durch die verbesserte Lokalität der Daten sogar ein Faktor von drei erzielt werden. Eine Berechnung mit single precision anstatt double precision könnte die Leistung zu Lasten der Rechengenauigkeit nochmals verdoppeln.

2.3. MarDyn

2.3.1. Struktur

MarDyn [4] ist eine Software zur Simulation von Nanofluiden auf Vielprozessorsystemen. Sie unterstützt mehrzentrige, starre Molekülmodelle und kurzreichweitige Kräfte zwischen Molkülen. Ein Molekül besteht aus mehreren Zentren, die fest mit einer Art Schwerpunkt, der Molekülposition, verbunden sind. Die Positionen der Zentren werden relativ zur Molekülposition angegeben. Zur Laufzeit können die absoluten Positionen der Zentren aus der Molekülpostion und den relativen Positionen berechnet werden. Eine Molekülart kann durch mehrere Zentren verschiedener Typen modelliert sein. Lennard-Jones Zentren verschiedener Moleküle interagieren mittels dem Lennard-Jones Potential miteinander. Tersoff Zentren dienen beispielsweise zur Modellierung von Bindungen zwischen

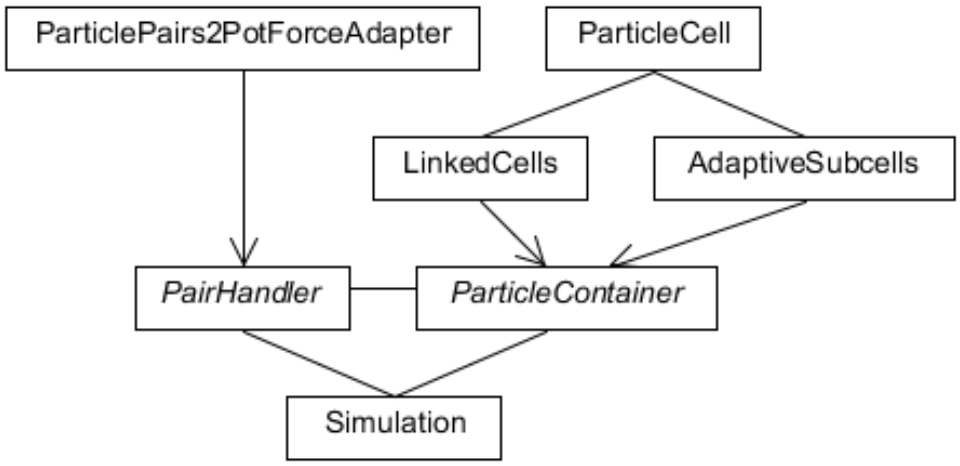


Abbildung 2.4.: Struktur von MarDyn. Grafik nach [4].

Kohlenstoffatomen. Weiterhin können Wechselwirkungen zwischen Dipolen und Quadrupolen simuliert werden. Unterschiedliche Molekültypen werden Komponenten genannt und können unterschiedliche Anzahlen und Kombinationen von Zentren sowie unterschiedliche Attribute für diese Zentren besitzen.

Um die kontinuierlichen physikalischen Interaktionen zwischen Molekülen zu simulieren, muss die Zeit diskretisiert werden. In jedem Rechenschritt werden zunächst die Kräfte, die auf die einzelnen Moleküle wirken, berechnet. Dann werden in einem Integrationsschritt die neuen Positionen, Orientierungen und Geschwindigkeiten der Moleküle berechnet. MarDyn implementiert dazu den Rotational Leapfrog Algorithmus [9]. MarDyn verwendet ein modulares Design in C++, so dass einzelne Programmteile modifiziert werden können, ohne den Rest des Programms anpassen zu müssen. Das Programm ist ein Gemeinschaftsprojekt der Technischen Universität München, dem Höchstleistungsrechenzentrum Stuttgart, der Technischen Universität Kaiserslautern und der Universität Paderborn.

Zentrales Element von MarDyn ist die Klasse *Simulation*, die den Programmablauf organisiert. Sie kennt unter anderem einen *ParticleContainer* und einen *PairHandler*. Ein *ParticleContainer* ist eine Klasse, die eine Liste von Molekülen verwaltet. Der Zugriff auf Molekülpaare mit einem maximalen Abstand r_{cutoff} zueinander erfolgt über die Methode *traversePairs*, die einen *ParticlePairsHandler* als Parameter erhält. Ein *ParticlePairsHandler* ist eine Klasse, welche die Methode *processPair* implementiert. Die Methode *processPair* erhält zwei benachbarte (also mit Abstand geringer als r_{cutoff}) Moleküle als Parameter und kann beispielsweise die Kräfte zwischen den Molekülen berechnen.

Eine naive Implementierung von *traversePairs* wird alle potentiellen Molekülpaare auf ihren Abstand testen um diejenigen zu finden, die benachbart sind. Dieser Ansatz führt zu

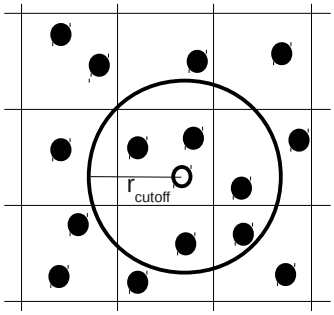


Abbildung 2.5.: Schema des Linked Cells Algorithmus. Grafik aus [7].

einer Laufzeit von $\mathcal{O}(n^2)$, wobei n die Anzahl der Moleküle im *ParticleContainer* ist. Da dies viel zu langsam ist, verwendet MarDyn *ParticleContainer*, die den Linked-Cell Algorithmus implementieren. Dabei wird der Simulationsraum in Zellen partitioniert und es werden nur Molekülpaare innerhalb einer Zelle oder in zwei benachbarten Zellen auf ihren Abstand verglichen. Bei konstanter Moleküldichte führt dieser Ansatz zu einer Laufzeit von $\mathcal{O}(n)$, da die Anzahl der Vergleiche pro Zelle im Wesentlichen konstant ist. MarDyn implementiert konkret die Klassen *LinkedCells* und *AdaptiveSubCells*, die beide auf dem Linked Cells Ansatz beruhen. Letztere versucht zusätzlich, die Größe der Zellen lokal an die Moleküldichte anzupassen. Die Klasse *ParticleCell* wird verwendet, um die Moleküle einer Zelle zu verwalten.

2.3.2. Kraftberechnung

Die Positionen der Zentren werden relativ zur Molekülposition gespeichert. Da MarDyn nur starre Molekülmodelle unterstützt, sind diese konstant. Zur Laufzeit können die absoluten Positionen der Zentren aus der Molekülpostion und den relativen Positionen berechnet werden. Die Entscheidung, ob eine Kraftberechnung zwischen zwei Molekülen stattfindet, wird bezüglich der Position der Moleküle getroffen. Falls eine Kraftberechnung stattfindet, werden alle Kräfte zwischen allen Zentren der beiden Moleküle berechnet, selbst wenn der Abstand zwischen zwei Zentren größer als r_{cutoff} ist. In MarDyn wird die Kraft zwischen zwei Molekülen nur einmal berechnet. Das Wechselwirkungsprinzip ausnutzend, wird die die Kraft einmal auf das erste und dann negativ auf das zweite Molekül angewendet. Alle Berechnungen werden mit double precision ausgeführt.

Die Klasse *ParticlePairs2PotForceAdapter* ist ein *ParticlePairsHandler* und implementiert die Berechnung aller derzeit von MarDyn unterstützten Kräfte. Dazu benötigt sie verschiedene Parameter, abhängig davon, welchen Typ die beiden gerade betrachteten Moleküle haben und welche Kraft gerade berechnet werden soll. Um diese Parameter für den *ParticlePairs2PotForceAdapter* effizient bereitzustellen, wurden die Klassen *Comp2Param* und *ParaStrm* implementiert [3]. *Comp2Param* erstellt eine zweidimensionale Tabelle von Listen von Parametern. Jede Liste enthält alle Parameter, die für die Kraftberechnung zwischen zwei Molekülen benötigt werden, wobei die Typen der beiden Moleküle die Position der

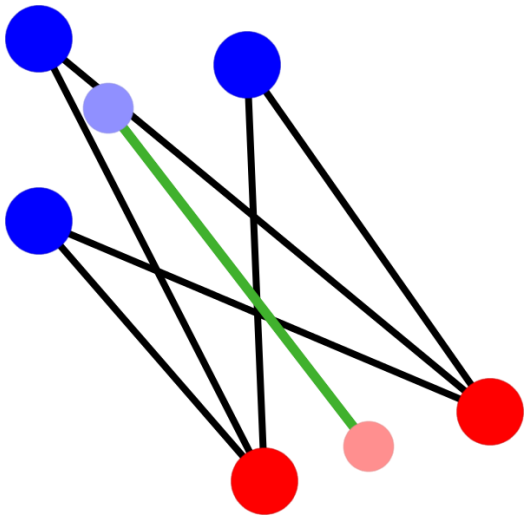


Abbildung 2.6.: Dreizentriges (blau) und zweizentriges (rot) Molekül, Molekülabstand (grün) und Interaktionen zwischen Zentren (schwarz). Kleine Kreise: Molekülposition.

Liste in der Tabelle bestimmen. Die Reihenfolge und Anzahl der Parameter ist exakt auf die Berechnung in *ParticlePairs2PotForceAdapter* abgestimmt.

2.3.3. MPI-Parallelisierung

MarDyn verwendet MPI-Parallelisierung um die Simulation auf mehrere Prozessoren zu verteilen. Dazu wird der Simulationsraum in Teilbereiche partitioniert. Jeder Bereich wird von einer Instanz von MarDyn berechnet. Nach jedem Zeitschritt werden globale Daten gesammelt und der Zustand der Simulation an den Bereichsgrenzen synchronisiert. Damit eine Instanz von MarDyn auch an den Grenzen korrekt rechnen kann, verwaltet jede Instanz eine zusätzliche Schicht Zellen, den sogenannten Halo, um den eigentlichen Teilbereich herum. Der Halo überlappt Bereiche, die zu anderen Instanzen gehören und enthält Kopien der Moleküle aus den entsprechenden Zellen der anderen Instanzen. Mit Hilfe der Molekülkopien können kurzreichweitige Interaktionen in jeder Instanz unabhängig von allen anderen Instanzen berechnet werden.

Bei der Synchronisation werden die Molekülkopien, die sich im Halo befinden, gelöscht und durch aktuelle Kopien der entsprechenden Bereiche ersetzt. In Abbildung 2.7 ist dargestellt, wie Moleküle aus Bereichen mehrerer Instanzen in die Halo-Bereiche anderer Instanzen kopiert werden. Falls sich ein Molekül aus dem Halo in den inneren Bereich bewegt hat, verbleibt es dort. In diesem Fall hat sich das entsprechende Molekül in allen anderen Instanzen in den Halo bewegt und wird damit gelöscht. Mit Hilfe des Halo können beispielsweise auch periodische Randbedingungen simuliert werden. Dazu werden im Synchronisationsschritt die Moleküle am Rand des Simulationsraums in den Halo des gegenüberliegenden Randes kopiert.

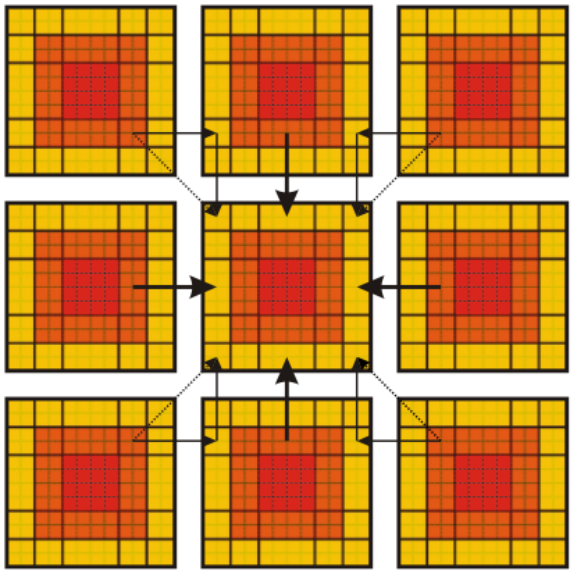


Abbildung 2.7.: Zerlegung des Simulationsraums in Teilbereiche mit umgebenden Halobereichen (gelb). Grafik aus [5].

3. Vektorisierung von MarDyn

3.1. Anpassung der Struktur von MarDyn

In diesem Kapitel wird beschrieben, wie die Struktur von MarDyn angepasst wurde, um die effiziente Vektorisierung der Kraftberechnung zu ermöglichen. Das Design und die Implementierung der neuen Struktur wurde von Wolgang Eckhardt vorgenommen.

3.1.1. Schnittstelle auf Zellebene

Es stellte sich schnell heraus, dass die Vektorisierung von MarDyn mit Hilfe einer *Structure of Arrays* mit dem bisherigen Softwaredesign von MarDyn nicht machbar war. Die *Structure of Arrays* erfordert den Zugriff auf alle Moleküle einer Zelle, aber MarDyn bietet keine Schnittstelle auf Zellebene an. Daher wurde das Design von MarDyn überarbeitet und eine Klasse *CellProcessor* in MarDyn integriert. Anstatt wie bisher über alle benachbarten Molekülpaare zu iterieren, iteriert ein *ParticleContainer* nun über alle Zellen und Zellpaare, die potentiell ein Paar benachbarter Moleküle enthalten und ruft für diese Zellen die Schnittstellenfunktionen eines *CellProcessors* auf. Um die Kompatabilität zum alten Programmcode zu erhalten, wurde ein *LegacyCellProcessor* eingeführt. Dieser implementiert die Iteration über die Molekülpaare, die zuvor der *ParticleContainer* erledigte. Der *LegacyCellProcessor* ruft dann einen *ParticlePairsHandler* auf, so dass die Kombination aus neuem *ParticleContainer* und *LegacyCellProcessor* semantisch dem alten *ParticleContainer* entspricht. Der dazu benötigte Aufwand pro Zelle ist im Verhältnis zum Gesamtaufwand vernachlässigbar. Ein Nachteil des veränderten Designs ist, dass *CellContainers* nun immer eine Zell-Struktur implementieren müssen. Dies ist nach derzeitigem Erkenntnisstand jedoch keine Einschränkung.

Ein *CellProcessor* implementiert die Methoden *initTraversal*, *preprocessCell*, *processCellPair*, *processCell*, *postprocessCell* und *endTraversal*. Ein *ParticleContainer* muss sicherstellen, dass er diese Methoden in der korrekten Reihenfolge aufruft. Alle Aufrufe müssen von *initTraversal* und *endTraversal* umgeben sein. Dazwischen kann für jede Zelle einmal *processCell*, sowie für jedes unterschiedliche Zellpaar einmal *processCellPair* aufgerufen werden. Bevor jedoch eine Zelle in einem dieser Aufrufe verwendet wird, muss ein Aufruf von *preprocessCell* erfolgen. Zwischen der letzten Verwendung einer Zelle und *endTraversal* muss die Bearbeitung der Zelle noch mit *postprocessCell* abgeschlossen werden.

Das neue Design von Mardyn ist sehr flexibel. Es unterstützt nicht nur mit Hilfe des *LegacyCellProcessors* die ursprüngliche Implementierung der Kraftberechnung. Auch andere Ansätze zur Paartraversierung wie Verlet-Nachbarschaftslisten [2] oder Algorithmen zur Kraftberechnung mit verbesserter Speichereffizienz [8] sind kompatibel.

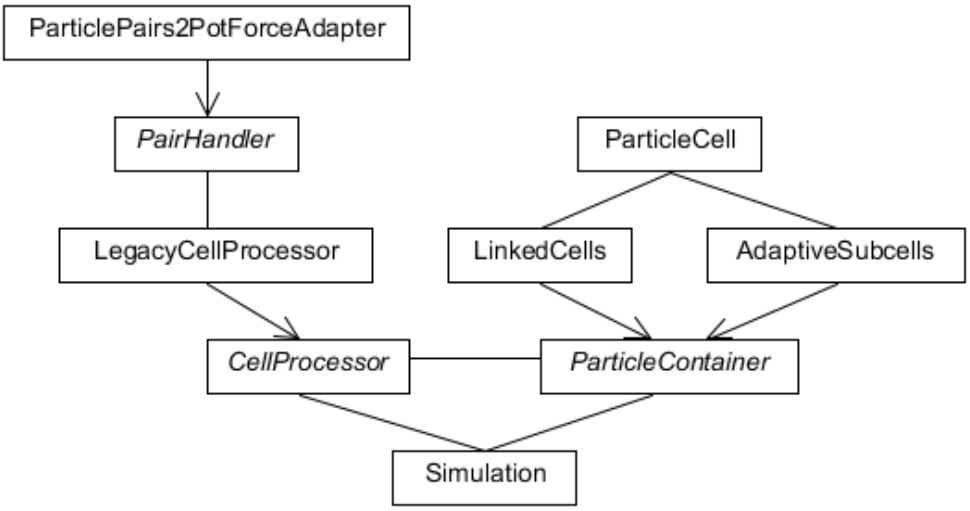


Abbildung 3.1.: Die neue Struktur von MarDyn.

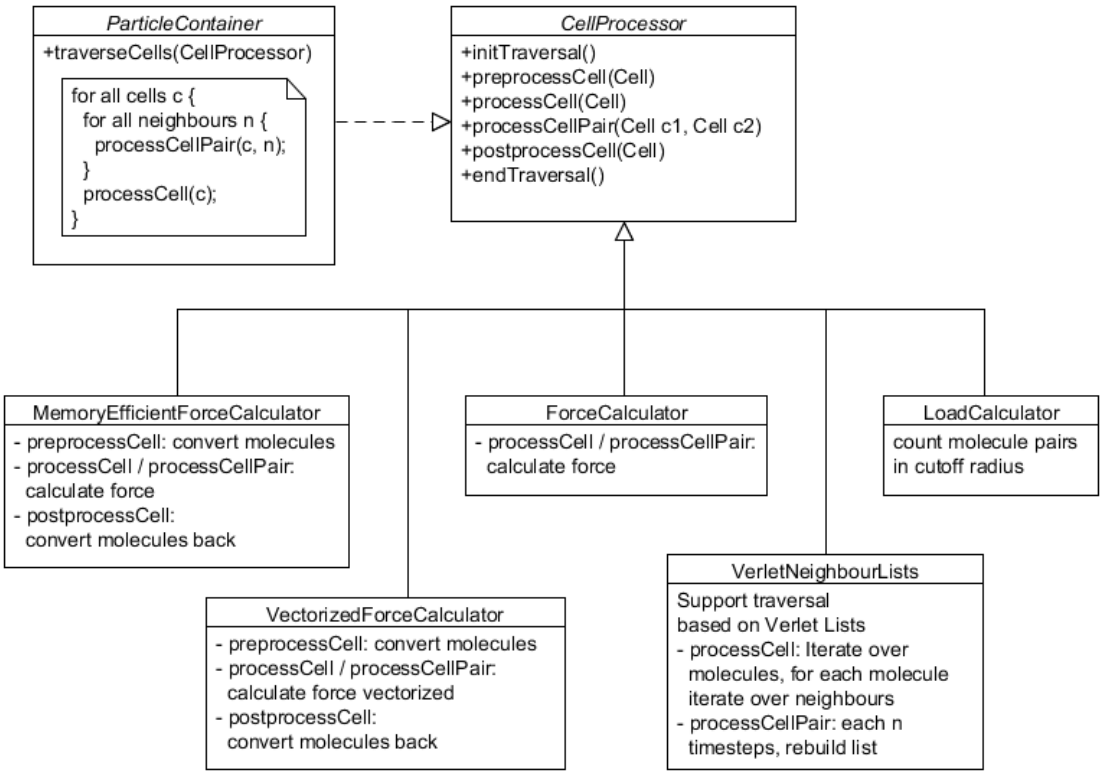


Abbildung 3.2.: Das *CellProcessor* Interface.

3.1.2. Neues Konzept für die Kraftberechnung

Insbesondere kann mit Hilfe des neuen Designs ein System implementiert werden, das es erlaubt, verschiedene Arten von Kraftberechnungen als *CellProcessors* zu implementieren, je nach Bedarf auszuwählen und beliebig zu kombinieren. Dazu wurde die Klasse *MacroCellProcessor* implementiert. Diese verwaltet eine Liste von *CellProcessors* und reicht alle Funktionsaufrufe an jeden der verwalteten *CellProcessors* weiter. So können mehrere *CellProcessors* gleichzeitig verwendet werden, ohne weitere Änderungen am Design von MarDyn notwendig zu machen. Die Klasse *Simulation* kennt nur einen einzigen *MacroCellProcessor*, diesem untergeordnete *CellProcessors* sind ihr unbekannt. Der *MacroCellProcessor* übernimmt zusätzlich die Initialisierung von Daten, die von mehreren *CellProcessors* gemeinsam genutzt werden, um Fehler durch Mehrfachinitialisierungen und Ähnliches zu vermeiden.

Der in Kapitel 3.2 vorgestellte *VectorizedLJCellProcessor* ist Teil dieses Konzepts und implementiert die Kraftberechnung durch das Lennard-Jones-Potential. Der *VectorizedLJCellProcessor* benötigt eine *Structure of Arrays* pro Zelle. Eine Tabelle aller *Structures of Arrays* und deren Zuordnungen zu den einzelnen Zellen zu verwalten ist jedoch aufwändig. Auch andere *CellProcessors* werden vermutlich ähnliche Daten pro Zelle verwalten müssen. Daher wurde die Klasse *ParticleCell* um eine Referenz auf eine *Structure of Arrays* erweitert, die der *VectorizedLJCellProcessor* benutzt, um Daten mit der Zelle zu verknüpfen. Andere *CellProcessors* können *ParticleCell* bei Bedarf ebenfalls erweitern.

3.2. Vektorisierung der Kraftberechnung

3.2.1. Formeln

Der *VectorizedLJCellProcessor* implementiert die Berechnung aller Daten, die auf der Interaktion von Lennard-Jones Zentren beruhen. Das Lennard-Jones-Potential [10] lässt sich nach folgender Formel berechnen:

$$U_{ij} = 4\epsilon((\frac{\sigma}{r_{ij}})^{12} - (\frac{\sigma}{r_{ij}})^{6}) \tag{3.1}$$

Hier ist r_{ij} der Abstand zwischen je zwei Lennard-Jones Zentren. Das Potential wird über alle Molekülpaare aufsummiert und zur Berechnung makroskopischer Werte wie dem Gesamtpotential der Simulation verwendet. Da das Lennard-Jones-Potential nur bis zu einem Abstand von r_{cutoff} berechnet wird, ist diese Summe nicht ganz korrekt. Ein zusätzlicher Parameter *shift* kann verwendet werden, um diesen Unterschied auszugleichen. In MarDyn wird dieser Parameter bei jeder Interaktion zwischen zwei Lennard-Jones Zentren einmal aufsummiert. Die Parameter ϵ, σ und *shift* sind abhängig von den Typen der Moleküle, die betrachtet werden, sowie von der Kombination der Zentren innerhalb dieser Molekükle. Die Kraft, die zwischen zwei Zentren wirkt, ergibt sich durch:

$$F_{ij} = -\nabla U_{ij} = \frac{24\epsilon}{r_{ij}}((\frac{\sigma}{r_{ij}})^{6} - 2(\frac{\sigma}{r_{ij}})^{12})\frac{\vec{r_{ij}}}{r_{ij}} \tag{3.2}$$

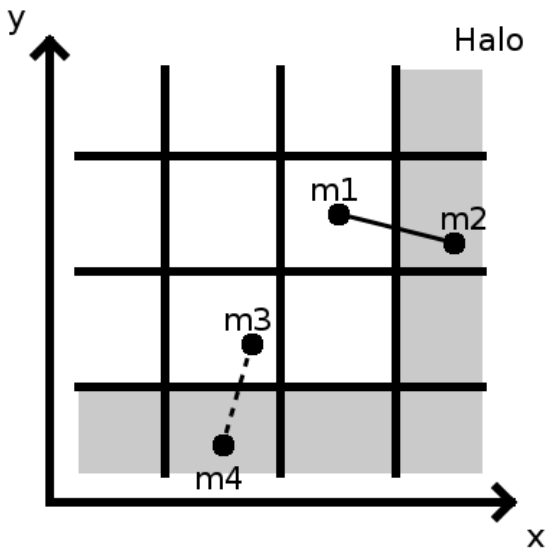


Abbildung 3.3.: Lexikographische Ordnung: $m1 < m2, m3 > m4$.

Wobei $\vec{r_{ij}}$ der Abstandsvektor zwischen den beiden Zentren ist. Außerdem wird während der Kraftberechnung das Virial mit Hilfe folgender Formel berechnet:

$$\sum_{i=1}^{N} \sum_{j<k} F_{ij} * r_{ij} \tag{3.3}$$

Es dient beispielsweise zur Berechnung des Drucks [16].

Da MarDyn zur MPI-Parallelisierung den gesamten Simulationsraum in mehrere Bereiche partitioniert und jeder *ParticleContainer* zusätzlich einen Halo verwaltet, muss bei den makroskopischen Werten Potential und Virial eine Besonderheit beachtet werden. Molekülpaare, bei denen ein Molekül in einer Halo-Zelle und das andere Molekül in einer Zelle im Inneren liegt, können von mehreren Instanzen betrachtet werden. In diesem Fall würden das Potential und das Virial für das Paar mehrfach aufsummiert. Aus diesem Grund werden für solche Paare die makroskopischen Werte nur berechnet, falls das erste Molekül des Paares kleiner als das zweite ist, bezogen auf die lexikographische Ordnung der Molekülpositionen. Da die Paare so bearbeitet werden, dass das erste Molekül nie im Halo liegt, werden das Potential und Virial für kein Paar mehrfach aufsummiert. In Abbildung 3.3 sind daher die makroskopischen Werte für das Paar $(m1, m2)$ zu berechnen, für $(m3, m4)$ hingegen nicht.

3.2.2. Implementierung der Kraftberechnung

Der *VectorizedLJCellProcessor* implementiert die Berechnung der obigen Formeln mit Hilfe von Vektorinstruktionen um eine Geschwindigkeitssteigerung zu erzielen. Dabei wird der in 2.2 beschriebene *Structure of Arrays* Ansatz verwendet. Teil der Implementierung

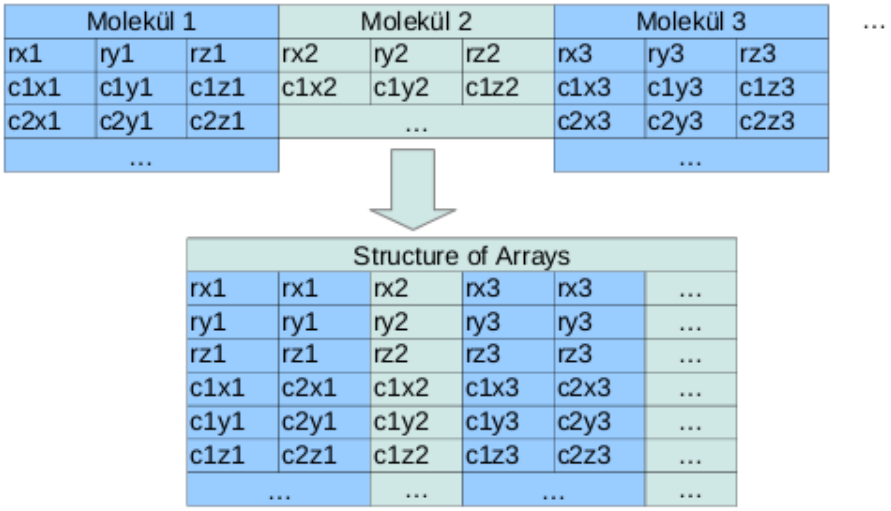


Abbildung 3.4.: Aufbau der *Structure of Arrays* aus einer Liste von Molekülen.

ist eine *Structure of Arrays* namens *LennardJonesSoA*. Sie verwaltet alle für die Berechnung benötigten Moleküldaten in vektorisierbarer Form. Die Klasse *ParticleCell* wurde um einen Zeiger auf eine *LennardJonesSoA* erweitert, so dass der *VectorizedLJCellProcessor* eine dieser Strukturen pro Zelle verwenden kann. Die *LennardJonesSoA* besitzt ein Array pro relevantem Molekülattribut. Jedes dieser Arrays enthält einen Eintrag für jedes Zentrum in jedem Molekül in der betrachteten Zelle. Mehrzentrige Moleküle erhalten also mehrere Einträge. Obwohl die Molekülposition für jedes Zentrum eines Moleküls identisch ist, wird auch diese mehrfach gespeichert. Dadurch wird vermieden, zwei Arrays unterschiedlicher Länge gleichzeitig bearbeiten zu müssen.

Die *LennardJonesSoA* umfasst folgende Daten:

Molekülposition: Die Entscheidung, ob für zwei Moleküle eine Berechnung erfolgen muss, wird durch den Vergleich des Abstands der Molekülpositionen mit r_{cutoff} getroffen. Außerdem wird der Abstandsvektor für die Berechnung des Virials benötigt.

Zentrumsposition: Die eigentliche Kraftberechnung betrachtet nur Paare von Lennard-Jones Zentren. Da die Kraft eine Funktion des Abstandes ist, werden die Positionen der Zentren benötigt.

Kraft: Dieser Wert wird auf 0 initialisiert und im Laufe der Berechnung werden die Kräfte, die auf das entsprechende Zentrum einwirken, hier aufsummiert.

Zentrumsindex: Dieser Index wird verwendet, um die Parameter ϵ, σ, und *shift* für die gerade betrachtete Kombination von Zentren zu ermitteln. Die beiden Indices des Zentrenpaars bestimmen dabei eine Position in den zweidimensionalen Parametertabellen.

Die Parametertabellen werden einmalig während der Initialisierung des *VectorizedLJCellProcessor* erstellt. Es ist nicht möglich, die bereits existierenden *ParamStreams* direkt zu verwenden, da diese eine eine andere Berechnungsreihenfolge vorgeben als der *VectorizedLJCellProcessor* verwendet. Derzeit werden dennoch die *ParamStreams* verwendet, um die Tabellen aufzubauen. Dazu muss zunächst eine Tabelle von Zentrumsindices erstellt werden. Für jede Komponente, die in der Simulation vorkommt, erhält jedes Zentrum einen eindeutigen Index. Die erste Komponente, mit n_1 Zentren, erhält die Indices $0 \dots n_1 - 1$, das zweite Zentrum, mit n_2 Zentren, die Indices $n_1 \dots n_1 + n_2 - 1$ und so weiter. Da für jede Kombination dieser Zentren unterschiedliche Parameter vorgegeben sein können, besitzen die Parametertabellen je eine Zeile und eine Spalte pro Zentrumsindex. Die Parametertabellen umfassen die Parameter ϵ, σ und *shift*.

Während der Zelltraversierung wird für jede Zelle zunächst einmal *preprocessCell* aufgerufen. Hier erstellt der *VectorizedLJCellProcessor* eine *LennardJonesSoA* für diese Zelle und speichert die Daten aller Moleküle in der Zelle passend in die *LennardJonesSoA*. Nun folgen für diese Zelle mehrere Aufrufe von *processCell* und *processCellPair*, in denen die Daten aus der *LennardJonesSoA* verwendet werden um die Kräfte zwischen den Molekülen zu berechnen. Die Resultate werden in der *LennardJonesSoA* gespeichert. Schließlich folgt ein Aufruf von *postprocessCell*, in dem die berechneten Kräfte aus der *LennardJonesSoA* auf die entsprechenden Moleküle aufsummiert werden. Danach kann die *LennardJonesSoA* wieder entfernt werden.

Bei jedem Aufruf von *processCell* und *processCellPair* muss für alle Molekülpaare in den entsprechenden Zellen im Wesentlichen die selbe Berechnung durchgeführt werden. Im Detail kann sich diese Berechnung jedoch unterscheiden. Ein wichtiger Unterschied ist bei der Berechnung der makroskopischen Werte Virial und Potential zu finden. Diese müssen für alle Molekülpaare berechnet werden, bei denen keines der Moleküle im Halo liegt. Falls beide Moleküle im Halo liegen, muss nichts berechnet werden. Nur wenn genau ein Molekül im Halo liegt, muss die lexikographische Ordnung der Moleküle herangezogen werden, um zu entscheiden, ob die makroskopischen Werte berechnet werden müssen. Da der *VectorizedLJCellProcessor* nicht Molekülpaare, sondern Paare von Zentren betrachtet, muss bei Paaren innerhalb einer einzelnen Zelle darauf geachtet werden, dass ein Paar von Zentren nicht zum selben Molekül gehört. Theoretisch hätte dies zwar keine Auswirkungen auf das Ergebnis, da die selbe Kraft einmal auf das Molekül addiert und einmal subtrahiert würde. Da zwei Zentren innerhalb eines Moleküls jedoch einen sehr geringen Abstand zueinander haben, wäre die Kraft zwischen den beiden Zentren im Vergleich zu den Kräften zwischen verschiedenen Molekülen sehr groß und könnte daher zu numerischen Auslöschungseffekten führen. Es kann angenommen werden, dass zwei Zentren genau dann Teil des selben Moleküls sind, wenn ihre Molekülpositionen identisch sind. Daher wird für Paare von Zentren mit identischen Molekülpositionen keine Kraftberechnung durchgeführt.

Insgesamt muss die Kraftberechnung in drei einander sehr ähnliche Funktionen aufgeteilt werden: Eine für Paare mit beiden Molekülen in der selben nicht-Halo Zelle, eine für Paare mit Molekülen in zwei verschiedenen nicht-Halo Zellen und eine für Paare mit

genau einem Molekül im Halo. Mit Hilfe von Templates konnte Codeduplikation vermieden werden. Hierzu wurde eine Variation des Policy Based Design Ansatzes verwendet.[1] Eine Funktion deckt den Hauptteil der Kraftberechnung ab und erhält als Templateparameter Instuktionen für die Codestellen, die sich in den drei Varianten der Kraftberechnung unterscheiden. Der Compiler erzeugt daraus dann für jede der Varianten eigenen Maschinencode. Laufzeitvergleiche mit semantisch identischem Code ohne die Verwendung von Templates lassen keine Unterschiede erkennen. Der Code ohne Templates ist durch die mehrfache Codeduplikation jedoch fehleranfällig, so dass der Template-basierten Variante der Vorzug gegeben wurde.

Die Kraftberechnung wurde sowohl nichtvektorisiert als auch vektorisiert mittels Intel SSE3 Intrinsics implementiert. Falls SSE3 nicht verfügbar ist, wird automatisch die nichtvektorisierte Version kompiliert. In beiden Varianten werden in *processCell* und *processCellPair* in einer Schleife alle Paare von Lennard-Jones-Zentren betrachtet und falls der Abstand gering genug ist, wird die Kraftberechnung durchgeführt. In der nichtvektorisierten Variante werden jeweils nur zwei Zentren betrachtet, wohingegen die vektorisierte Version je ein Zentrum gleichzeitig mit zwei anderen kombiniert. Dazu wird zunächst jedes Datenelement des ersten Zentrums in je ein SSE-Register dupliziert. Dann werden die Daten der beiden anderen Zentren ebenfalls in SSE-Register geladen, und zwar so, dass in einem Register das gleiche Datenelement beider Zentren liegt. Das Laden dieser Daten kann mit nur einer Instruktion pro Register erfolgen, da die Daten aller Zentren zuvor in der *LennardJonesSoA* entsprechend sortiert wurden und die beiden Zentren so gewählt sind, dass ihre Daten in der *LennardJonesSoA* nebeneinander liegen. Wenn beispielsweise die x-Koordinate geladen wurde, befindet sich die Koordinate des ersten Zentrums zweifach in einem Register. Die x-Koordinaten des zweiten und dritten Zentrums befinden sich in einem weiteren Register. Nun kann für die Abstandsberechnung die Differenz zwischen diesen Koordinaten mit einer Vektorinstruktion berechnet werden. Als Ergebnis erhält man ein Register, das den Abstand in x-Richtung zwischen dem ersten und dem zweiten Zentrum sowie zwischen dem ersten und dem dritten Zentrum enthält. Auf diese Weise kann die Interaktion zweier Paare von Zentren gleichzeitig berechnet werden.

Nachdem für zwei Paare von Zentren der Abstand der zugehörigen Moleküle berechnet wurde, muss entschieden werden, ob für diese Paare eine Kraftberechnung durchgeführt werden soll. Falls beide Paare weniger als r_{cutoff} voneinander entfernt sind, kann die Berechnung für beide gleichzeitig mit Hilfe von Vektorinstruktionen erfolgen. Wenn nur bei einem der beiden Paare der Abstand kleiner als r_{cutoff} ist, wird zwar die Berechnung wieder vektorisiert durchgeführt, aber das Ergebnis für das Paar, dessen Moleküle weit von einander entfernt sind, verworfen. Dazu wird das Ergebnis mit einer Maske versehen, die aus dem Vergleich der Abstände mit r_{cutoff} gebildet wird. Für die Berechnung der makroskopischen Größen Potential und Virial wird ähnlich vorgegangen. Die Maskierung basiert hier jedoch auf der lexikographischen Ordnung der Moleküle.

3.3. Hilfsmittel zur Laufzeitanalyse

Um weitere Laufzeitanalysen zu ermöglichen, wurde eine Klasse *LJFlopCounter* erstellt. Sie implementiert das *CellProcessor* Interface und dient dazu, die effektiven floating point operations (FLOPs) bei der Lennard-Jones Kraftberechnung zu zählen und die Laufzeit der tatsächlichen Berechnung zu messen. Die Gewichtung der einzelnen Bestandteile der Kraftberechnung basiert auf der nichtvektorisierten Implementierung der Berechnung. Bei mehrzentrigen Molekülen wird die Berechnung des Molekülabstands nur einmal gezählt, auch wenn diese im *VectorizedLJCellProcessor* mehrfach stattfinden kann. Dies begründet sich dadurch, dass die mehrfache Berechnung eigentlich redundant ist. Eine detaillierte Auflistung der effektiven FLOPs in der Kraftberechung ist in Anhang A zu finden.

Der *LJFlopCounter* analysiert bei einem Aufruf von *processCell* oder *processCellPair* die Moleküle in den Zellen und entscheidet, wie viele Rechenoperationen für diese Zellen insgesamt zur Berechnung der Interaktionen zwischen Lennard-Jones Zentren notwendig sind. Dann ruft er die entsprechende Funktion eines anderen *CellProcessors* auf und misst die benötigte Zeit für diesen Aufruf. So können die FLOPs pro Sekunde für einen beliebigen *CellProcessor*, der die Lennard-Jones Kraftberechnung implementiert, gemessen werden. Als Nebeneffekt kann der Ausgabe des *LJFlopCounters* unter Anderem entnommen werden, welcher Anteil der insgesamt überprüften Molekülpaare tatsächlich benachbart sind.

4. Experimente und Ergebnisse

In diesem Kapitel wird die Testumgebung beschrieben. Die Ergebnisse der Laufzeitanalysen für ein- und mehrzentrige Szenarien sowie verschiedene Konfigurationen von MarDyn werden vorgestellt und analysiert. Die Szenarien wurden von Wolfgang Eckhardt bereitgestellt.

4.1. Konfiguration und Szenarien

Die Experimente zur Laufzeitanalyse wurden auf dem SGI Ice Cluster des Leibniz Rechenzentrums in München durchgeführt. MarDyn wurde auf einem Intel Nehalem Xeon Prozessor (Intel Xeon E5540 at 2.53GHz) ausgeführt. Da MarDyn grundsätzlich gut mit der Anzahl der Rechenkerne skaliert [4], konnte auf einen detaillierten Vergleich der sequentiellen mit der MPI-parallelisierten Version von MarDyn verzichtet werden. Folgende Konfigurationen von MarDyn wurden untersucht:

Original: Keinerlei Änderungen am ursprünglichen Code. Der *ParticleContainer* ruft direkt den *ParticlePairs2PotForceAdapter* auf.

Legacy: Die Kombination von *MacroCellProcessor*, *LegacyCellProcessor* und *ParticlePairs2PotForceAdapter* entspricht dem ursprünglichen Code nach Anwendung des Refactorings aus Kapitel 3.1.

Novec: Der *MacroCellProcessor* und *VectorizedLJCellProcessor* führen die Berechnung der Lennard-Jones-Kraft aus. Diese Version verwendet keine Vektorinstruktionen.

Vectorized: Wie *Novec*, aber unter Verwendung von SSE3 Vektorinstruktionen.

Die Konfigurationen *Novec* und *Vectorized* unterstützen ausschließlich die Berechnung der Interaktionen von Lennard-Jones Zentren. Da für das Tersoff-Potential und die Dipol-Quadrupol-Wechselwirkungen noch keine Implementierungen nach dem *CellProcessor* Design existieren, kann eine vollständige Berechnung nur unter Verwendung des *LegacyCellProcessors* stattfinden. Man kann zwar die Berechnung der Lennard-Jones Kraft im *LegacyCellProcessor* deaktivieren und diesen mit dem *VectorizedLJCellProcessor* kombinieren, aber da dabei dennoch im *LegacyCellProcessor* über alle Paare von Lennard-Jones Zentren iteriert wird, ist dies keine effiziente Implementierung. Daher wurden für alle Konfigurationen nur Simulationen von Lennard-Jones Interaktionen durchgeführt.

Für die Laufzeitanalyse wurden homogene Szenarien mit Molekülen mit ein bis drei Lennard-Jones Zentren verwendet. Es wurden jeweils ein Szenario mit 160.000 und eines mit 1.300.000 Molekülen verwendet. Zusätzlich wurde r_{cutoff} so variiert, dass jedes Szenario mit 20, 50 und 100 Moleküle pro Zelle analysiert wurde. Für einzentrige Moleküle entspricht dies in etwa den Szenarien, die in [7] verwendet wurden.

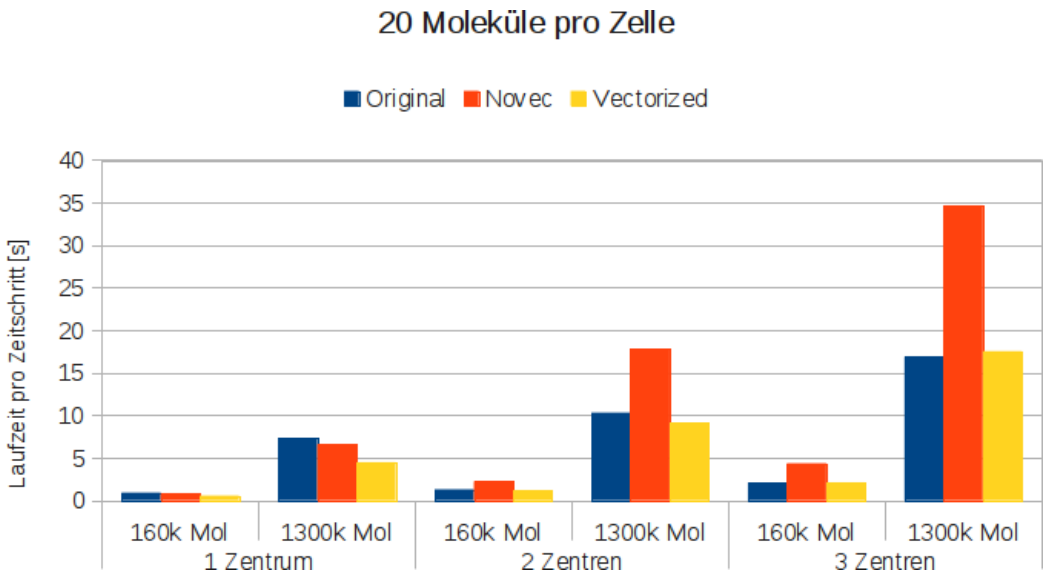


Abbildung 4.1.: Laufzeit in Sekunden für verschiedene Konfigurationen und Szenarien bei 20 Molekülen pro Zelle.

Zunächst stellt sich heraus, dass kein signifikanter Unterschied zwischen den Konfigurationen *Original* und *Legacy* besteht. Dies entspricht den Erwartungen, da lediglich ein geringer, konstanter Aufwand pro Zelle notwendig ist, um das *CellProcessor* Design zu implementieren. Dieser Aufwand ist im Vergleich zur Kraftberechnung vernachlässigbar.

4.2. Einzentrige Moleküle

Beim Vergleich der drei Konfigurationen *Legacy*, *Novec* und *Vectorized* für die Szenarien mit nur einem Lennard-Jones Zentrum pro Molekül fällt auf, dass *Novec* ein wenig schneller arbeitet als *Legacy*. Insbesondere bedeuted dies, dass sich die Umsortierung der Moleküldaten in *preprocessCell* nicht negativ auf die Geschwindigkeit von *Novec* auswirkt. *Novec* kann Vorteile aus der verbesserten Lokalität der Daten ziehen, die durch die *Structure of Arrays* entsteht. Die Methoden *preprocessCell* und *postprocessCell* beanspruchen im Verhältnis zu *processCell* und *processCellPair* nur wenig Laufzeit. Dies erklärt sich dadurch, dass *preprocessCell* und *postprocessCell* nur einmal pro Zelle aufgerufen werden und ihre Laufzeit linear zur Anzahl der Moleküle in einer Zelle ist. Für die Kraftberechnung hingegen wird jede Zelle mehrfach benötigt und da alle Paare von Molekülen betrachtet werden, ist die Laufzeit quadratisch in Abhängigkeit von der Anzahl der Moleküle in einer Zelle.

In MarDyn erfolgen alle Berechnungen in double precision. Mit Hilfe von SSE3 können zwei double precision Werte gleichzeitig in einem Register bearbeitet werden. Daher kann mit SSE3 theoretisch eine Leistungsverbesserung von 100% erreicht werden. Bei 20 einzentrigen Molekülen pro Zelle verbessert sich die Leistung von *Novec* nach *Vectorized* um etwa 45%, bei 100 Molekülen dagegen um fast 90%. Je mehr Moleküle sich in einer Zelle befinden, desto weniger fällt der zusätzliche Aufwand für die Vektorisierung wie das

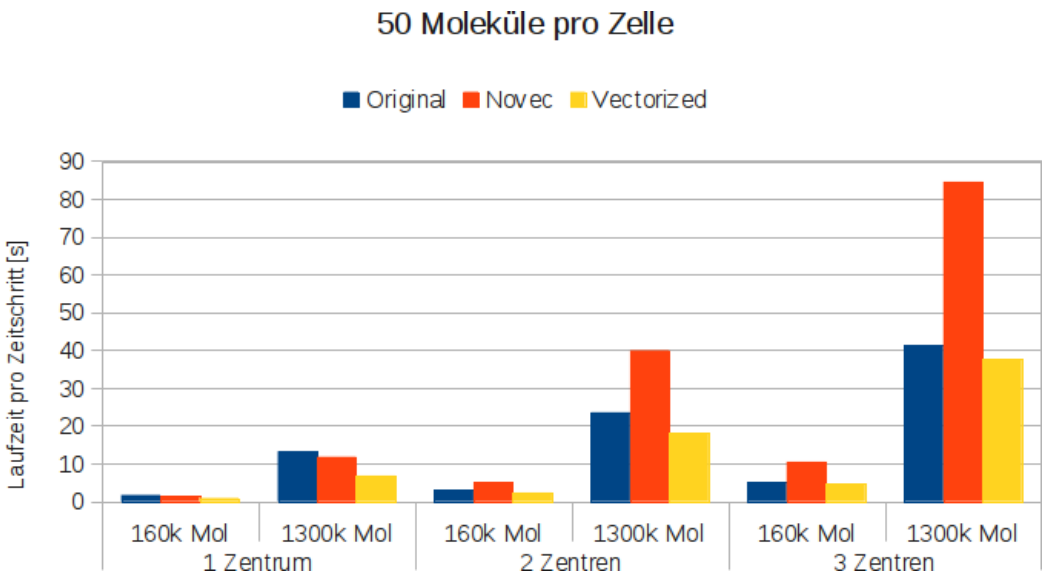


Abbildung 4.2.: Laufzeit in Sekunden für verschiedene Konfigurationen und Szenarien bei 50 Molekülen pro Zelle.

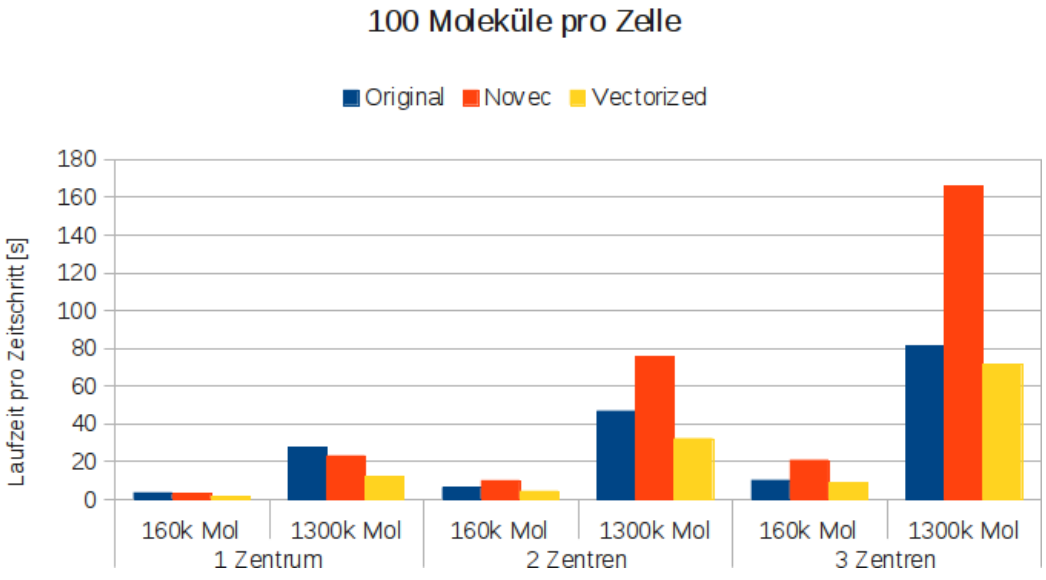


Abbildung 4.3.: Laufzeit in Sekunden für verschiedene Konfigurationen und Szenarien bei 100 Molekülen pro Zelle.

Erstellen der *Structure of Arrays* ins Gewicht, da der Aufwand für die Kraftberechnung quadratisch mit der Anzahl der Moleküle pro Zelle zunimmt. Dieses Ergebnis entspricht in etwa den Erwartungen, das Maximum von 100% wird nahezu erreicht. *Vectorized* kann die Leistung gegenüber *Original* um mehr als 60% für 20 Molelüle beziehungsweise 125% für 100 Moleküle pro Zelle verbessern.

4.3. Mehrzentrige Moleküle

In den Szenarien mit zweizentrigen Molekülen befinden sich in einer Zelle doppelt so viele Zentren wie in den einzentrigen Szenarien. Aus diesem Grund unterscheiden sich die Leistungsverbesserungen hier weniger als bei den einzentrigen Molekülen. Sie liegen zwischen etwa 95% für 20 und 135% für 100 Moleküle pro Zelle. Bei dreizentrigen Molekülen ergeben sich ähnliche Werte. Dieses Ergebnis ist überraschend, da eine Verbesserung um mehr als 100% eigentlich nicht zu erwarten ist. Der Grund für dieses Ergebnis konnte bisher jedoch nicht präzise bestimmt werden.

Besonders auffällig ist, dass bei mehrzentrigen Molekülen die Konfiguration *Vectorized* nur in etwa so schnell ist wie *Original*, und *Novec* sogar deutlich langsamer. Der Grund dafür ist, dass in *Original* der Vergleich mit r_{cutoff} nur einmal pro Molekülpaar durchgeführt wird. In *Novec* und *Vectorized* erfolgt dieser Vergleich einmal pro Lennard-Jones Zentrum. Für einzentrige Moleküle macht dies offensichtlich keinen Unterschied, aber bei zweizentrigen Molekülen benötigt *Novec* bereits vier Vergleiche pro Molekülpaar und bei drei Zentren sind es schon neun. Da im Durchschnitt nur etwa 20% der überprüften Molekülpaare weniger als r_{cutoff} voneinander entfernt sind, wird bei 80% der Paare nur die Berechnung des Abstandes und der Vergleich durchgeführt. Somit benötigt *Novec* bei 80% der Paare jeweils den vier- beziehungsweise neunfachen Aufwand. Zwar ist der Aufwand für die eigentliche Kraftberechnung etwa fünf Mal so hoch wie der für die Abstandsberechnung, sie wird aber nur relativ selten durchgeführt. So entfallen je nach Szenario und Konfiguration bis zu 52% der benötigten Rechenleistung auf die Abstandsberechnung. Die Konfiguration *Vectorized* geht nach dem selben Schema vor und kann daher die Leistung gegenüber *Original* kaum oder gar nicht steigern. Dieses Ergebnis kann nicht zufriedenstellen. Obwohl der Prozessor mit *Vectorized* stärker ausgelastet ist, kann kein signifikanter Laufzeitvorteil erreicht werden. Der Algorithmus muss daher für mehrzentrige Moleküle verbessert werden. Ziel sollte sein, die redundanten Abstandsberechnungen zu vermeiden. Vorschläge, wie dies umgesetzt werden könnte, werden in Kapitel 5.2 gemacht.

4.4. Prozessorauslastung

Der für die Performanceanalyse verwendete Intel Xeon E5540 Prozessor kann auf SSE-Registern eine Addition und eine Multiplikation gleichzeitig ausführen [12]. Jedes dieser Register kann zwei double precision Gleitkommazahlen beinhalten. Durch pipelining kann jede Recheneinheit eine Instruktion pro Taktzyklus bearbeiten, doch eine einzelne Instruktion kann mehrere Zyklen benötigen, um zu einem Ergebnis zu kommen. Da der Prozessor mit einer Taktfrequenz von 2.53GHz arbeitet, ergibt sich eine theoretische peak

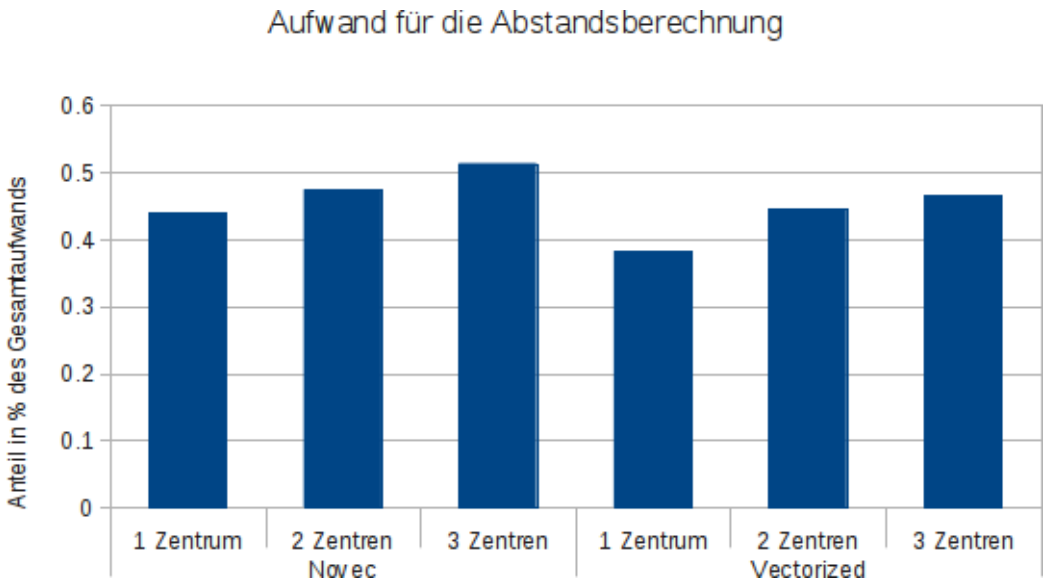


Abbildung 4.4.: Anteil der Abstandsberechnung am Gesamtaufwand der Kraftberechnung für verschidene Szenarien.

performance von 10.1 Giga FLOPS pro Sekunde und Rechenkern. Um die Auslastung des Prozessors bei Verwendung des *VectorizedLJCellProcessor* zu analysieren, wurden die Konfigurationen *Novec* und *Vectorized* mit dem *LJFlopCounter* kombiniert. Zusätzlich wurde eine Konfiguration *Nodiv* erstellt, die eine Variante von *Vectorized* ist. Die Kraftberechnung in *VectorizedLJCellProcessor* enthält genau eine Division bei der Invertierung des Abstandes der Zentren (siehe Anhang A, Codezeile 19). Divisionen sind sehr aufwändig und blockieren daher andere Berechnungen über viele Taktzyklen hinweg. In *Nodiv* wurde diese Division entfernt, doch die dadurch verfälschten Ergebnisse werden nicht an den nächsten Zeitschritt weitergegeben. Für die Analysen mit dem *LJFlopCounter* wurde ein Szenario mit 65.000 einzentrigen Molekülen verwendet.

Wie in Abbildung 4.5 zu sehen ist, steigt der Durchsatz bei 100 Molekülen pro Zelle um etwa 100% bei Verwendung von *Vectorized* anstelle von *Novec*. Die Konfiguration *Nodiv* erhöht die Auslastung um 165% gegenüber *Novec* und um 30% gegenüber *Vectorized*. Da die Division in *Vectorized* nur für etwa 20% der Paare durchgeführt wird, wird klar, dass die Division für einen Großteil der Rechenzeit in diesem inneren Teil der Kraftberechnung verantwortlich ist.

Es werden mit *Nodiv* bis zu 3,3 Giga FLOPs pro Sekunde erreicht. Dies entspricht etwa einem Drittel des maximalen Durchsatzes. Das Maximum wird nicht erreicht, da nicht alle Rechenoperationen in der Kraftberechnung unabhängig voneinander sind. Insbesondere ist dies bei der Abstandsberechnung und bei der Berechnung des Terms $(\frac{\sigma}{r_{ij}})^6$ der Fall. Hier muss die Additions- beziehungsweise Multiplikationseinheit erst auf ein Ergebnis warten, bevor sie die nächste Instruktion ausführen kann. Dies führt dazu, dass der Prozessor potentiell über mehrere Taktzyklen nicht ausgelastet ist. Die Auslastung der Addi-

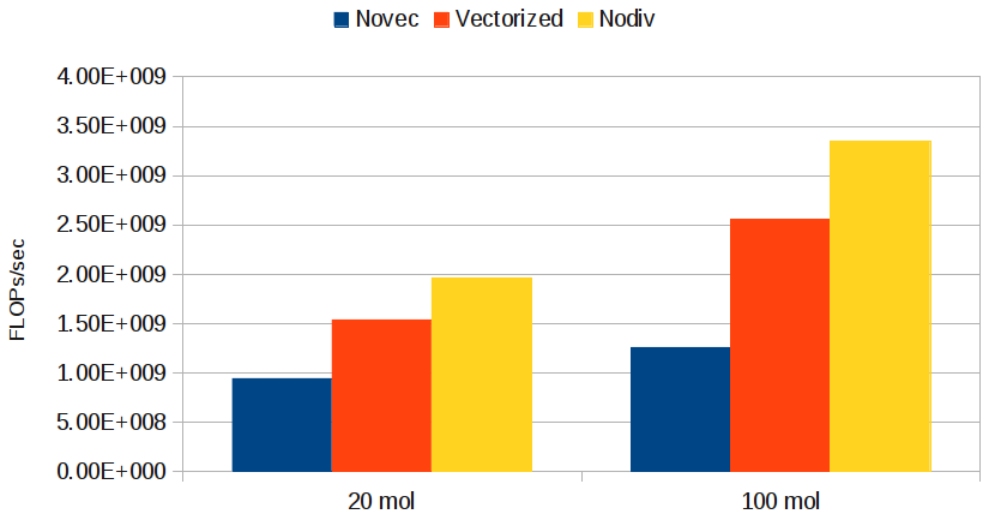


Abbildung 4.5.: FLOPs pro Sekunde für einzentrige Moleküle.

tionseinheit ist insgesamt eher gering, da die Kraftberechnung hauptsächlich Multiplikationen verwendet. Weitere Verluste ergeben sich durch fehlerhafte branch prediction. Bei der Entcheidung, ob für ein Molekülpaar die Kraftberechnung durchgeführt werden muss, sind in etwa 20% der Fälle fehlerhafte Vorhersagen zu erwarten. Dies führt dazu, dass der Prozessor einen falschen Codeteil ausführt und so Leistung verliert. Insgesamt sind die 3.3 Giga FLOPs pro Sekunde, die *Nodiv* erreicht, somit zufriedenstellend. Die Konfiguration *Vectorized* leistet 2,6 Giga FLOPs pro Sekunde, *Novec* 1,3. Der Unterschied von *Vectorized* zu *Nodiv* ist auf die Division zurückzuführen.

5. Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der vorherigen Kapitel zusammengefasst und ein Ausblick auf weitere Optimierungsmöglichkeiten gegeben.

5.1. Zusammenfassung

Das *CellProcessor* Interface erwies sich als sehr nützlich. Mit dem *MacroCellHandler* konnte ein System implementiert werden, das es erlaubt, die Kraftberechnung ohne signifikanten Aufwand flexibel an die Anforderungen des zu simulierenden Szenarios anzupassen. Der *LJFlopCounter* kann auch zur Performanceanalyse weiterer *CellProcessors*, die die Lennard-Jones Kraftberechnung durchführen, verwendet werden.

Da mit dem *VectorizedLJCellProcessor* die Kraftberechnung für Lennard-Jones Zentren vollständig in einer Klasse untergebracht wurde, beschränkt sich auch die Verwendung der SSE3 Intrinsics auf diese Klasse. Andere Programmteile wurden nicht beeinflusst. So konnten die maschinennahen Vektorinstruktionen gut mit objektorientiertem C++ Code vereint werden. Mit Hilfe von Templates konnte Codeduplikation vermieden werden, so dass die Kraftberechnung trotz Vektorisierung überschaubar und wartbar bleibt.

Die Ergebnisse für einzentrige Moleküle decken sich mit den Erwartungen. Hier konnte erfolgreich eine effiziente Vektorisierung von MarDyn implementiert werden. Bei mehrzentrigen Molekülen sind die Ergebnisse noch nicht zufriedenstellend. Es konnte nur eine geringe Leistungssteigerung erzielt werden, in manchen Fällen war auch die vektorisierte Version langsamer als das Originalprogramm. Bei komplexeren Molekülen würde das Ergebnis noch schlechter ausfallen. Daher muss in Betracht gezogen werden, die Abstandsberechnung auch im *VectorizedLJCellProcessor* nur einmal pro Molekülpaar anstatt einmal pro Zentrumspaar durchzuführen. Es ist jedoch nicht sinnvoll, für ein Molekülpaar zu bestimmen, ob die Kraftberechnung durchgeführt werden muss und dann alle Zentren dieser zwei Moleküle vektorisiert zu bearbeiten. In Kapitel 4 wurde gezeigt, dass sich die Vektorisierung erst bei vielen Zentren pro Zelle lohnt. Wenn nur alle Zentren eines Molekülpaares betrachtet werden, dann entspricht dies einer Berechnung mit dem *VectorizedLJCellProcessor* für zwei Zellen mit eben diesen Zentren. Selbst bei komplexen Molekülen ist diese Zahl zu gering um eine effiziente Vektorisierung zu erlauben.

5.2. Ausblick

Ein Erfolg versprechender Ansatz zur Elimination der redundanten Abstandsberechnungen im *VectorizedLJCellProcessor* ist, zunächst die Abstandsberechnung und den Vergleich mit r_{cutoff} für alle Molekülpaare durchzuführen und die Ergebnisse zu speichern.

Danach wird die eigentliche Kraftberechnung wie zuvor durchgeführt. Die Ergebnisse der Abstandsberechnung können pro Molekül oder pro Zentrum gespeichert werden. Ersteres erfordert weniger Speicher, macht es aber erforderlich, während der Kraftberechnung die Relation zwischen Molekülen und Zentren zu beachten. Zusätzlich muss auf die Berechnung der makroskopischen Werte Rücksicht genommen werden. Auch für diese ist zu entscheiden, ob sie berechnet werden müssen oder nicht. Da zur Berechnung des Virials in MarDyn der Abstandsvektor zwischen den Molekülen benötigt wird, muss eventuell auch dieser gespeichert werden. Dies erhöht den Speicherbedarf um einen signifikanten Faktor gegenüber einfachen bool-Werten, wie sie für das Ergebnis des Vergleichs mit r_{cutoff} notwendig sind, da die Darstellung des Abstandsvektors drei double precision Werte erfordert.

Anstelle der Berechnung des Virials mit Hilfe des Molekülabstandes ist es auch denkbar, den Abstand der Zentren zu verwenden. Dadurch kann der Abstandsvektor nach dem Vergleich mit r_{cutoff} verworfen werden. Dies spart Speicherplatz und könnte auch bei der aktuellen Version des *VectorizedLJCellProcessors* die Laufzeit verbessern, indem eventuell frei werdende Register anders genutzt werden.

Der *VectorizedLJCellProcessor* wurde bisher nur als nichtvektorisierte und mit SSE3 vektorisierte Version implementiert. Moderne Prozessoren unterstützen aber den Befehlssatz AVX, mit dem doppelt so viele Berechnungen gleichzeitig durchgeführt werden können als mit SSE3. In [7] wurde gezeigt, dass die Vektorisierung mittels des *Structure of Arrays* Ansatzes auf AVX erweiterbar ist, auch wenn ein Faktor von zwei im Vergleich zu SSE3 nicht erreicht wird. Da die Zentren im *VectorizedLJCellProcessor* im Wesentlichen wie einzentrige Moleküle behandelt werden können, ist anzunehmen, dass sich auch der *VectorizedLJCellProcessor* von AVX Instruktionen profitieren kann. Die Implementierung einer vektorisierten Kraftberechnung für Dipole und Quadrupole wird derzeit entwickelt. Auch hierzu soll der *Structure of Arrays* Ansatz verwendet werden.

Anhang

A. Effektive FLOPs in der Kraftberechnung

Im Folgenden findet sich eine Auflistung aller Rechenoperationen in der Lennard-Jones Kraftberechnung, die im *LJFlopCounter* als effektive FLOPs gezählt werden. Alle Operationen werden gleich gewichtet.

Insgesamt ergeben sich so acht FLOPs pro Molekül und zusätzlich bis zu 35 FLOPs pro Zentrum. Die Operationen ab Zeile elf werden nur gezählt, falls der Abstand zwischen den Molekülen geringer als r_{cutoff} ist. Die Operationen in den Zeilen 33 und 34 sind nur relevant, wenn die makroskopischen Werte für dieses Paar berechnet werden sollen.

```
1  // Diese Operationen werden nur einmal pro Molekül gezählt.
2  // 3 Subtraktionen, 3 Multiplikationen, 2 Additionen.
3  double m_dx = soa1.m_r_x[i] - soa2.m_r_x[j];
4  double m_dy = soa1.m_r_y[i] - soa2.m_r_y[j];
5  double m_dz = soa1.m_r_z[i] - soa2.m_r_z[j];
6  double m_r2 = m_dx * m_dx + m_dy * m_dy + m_dz * m_dz;
7
8  // Alle folgenden Operationen werden einmal pro Zentrum gezählt.
9  if (/* distance < cutoff */) {
10   // 3 Subtraktionen, 3 Multiplikationen, 2 Additionen.
11   double c_dx = soa1.c_r_x[i] - soa2.c_r_x[j];
12   double c_dy = soa1.c_r_y[i] - soa2.c_r_y[j];
13   double c_dz = soa1.c_r_z[i] - soa2.c_r_z[j];
14   double c_r2 = c_dx * c_dx + c_dy * c_dy + c_dz * c_dz;
15
16   double eps_24 = _eps_sig[soa1.c_id[i]][2 * soa2.c_id[j]];
17   double sig2 = _eps_sig[soa1.c_id[i]][2 * soa2.c_id[j] + 1];
18
19   double r2_inv = 1.0 / c_r2; // 1 Division
20   double lj2 = sig2 * r2_inv; // 1 Multiplikation
21   double lj6 = lj2 * lj2 * lj2; // 2 Multiplikationen
22   double lj12 = lj6 * lj6; // 1 Multiplikation
23   double lj12m6 = lj12 - lj6; // 1 Subtraktion
24
25   // 2 Multiplikationen, 1 Addition
26   double scale = eps_24 * r2_inv * (lj12 + lj12m6);
27   double fx = c_dx * scale; // 1 Multiplikation
28   double fy = c_dy * scale; // 1 Multiplikation
29   double fz = c_dz * scale; // 1 Multiplikation
```

```
30
31    if (/* molecule1 < molecule2, lexicographic */) {
32      // 4 Multiplikationen, 5 Additionen
33      _upot6lj += eps_24 * lj12m6 + _shift6[soa1.c_id[i]][soa2.c_id[j]];
34      _virial += m_dx * fx + m_dy * fy + m_dz * fz;
35    }
36
37    soa1.c_f_x[i] += fx; // 1 Addition
38    soa1.c_f_y[i] += fy; // 1 Addition
39    soa1.c_f_z[i] += fz; // 1 Addition
40    soa2.c_f_x[j] -= fx; // 1 Subtraktion
41    soa2.c_f_y[j] -= fy; // 1 Subtraktion
42    soa2.c_f_z[j] -= fz; // 1 Subtraktion
43  }
```

Literaturverzeichnis

[1] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] M. P. Allen and D. J. Tildesley. *Computer simulation of liquids*. Clarendon Press, New York, NY, USA, 1989.

[3] Martin Bernreuther, Martin Buchholz, and Hans-Joachim Bungartz. Aspects of a parallel molecular dynamics software for nano-fluidics. In G.R. Joubert, C. Bischof, F. Peters, T. Lippert, M. Bücker, P. Gibbon, and B. Mohr, editors, *Parallel Computing: Architectures, Algorithms and Applications*, volume 38 of *NIC Series*, pages 53–60, Jülich, September 2007. International Conference ParCo 2007, NIC.

[4] Martin Buchholz, Hans-Joachim Bungartz, and Jadran Vrabec. Software design for a highly parallel molecular dynamics simulation framework in chemical engineering. *Journal of Computational Science*, 2(2):124–129, May 2011.

[5] Hans-Joachim Bungartz. Vorlesungsskript: Algorithms of Scientific Computing II: Molecular Dynamics Simulation. `http://www5.in.tum.de/lehre/vorlesungen/asc2/WS1011/Lectures/1_MolecularDynamics.pdf`, 2010.

[6] Ron O. Dror, Cliff Young, and David E. Shaw. Anton, a special-purpose molecular simulation machine. In *Encyclopedia of Parallel Computing*, pages 60–71. 2011.

[7] Wolfgang Eckhardt and Alexander Heinecke. An efficient vectorization of linked-cell particle simulations. In *ACM International Conference on Computing Frontiers*, pages 241–243, Cagliari, May 2012.

[8] Wolfgang Eckhardt and Tobias Neckel. Memory-efficient implementation of a rigid-body molecular dynamics simulation. In *The 11th International Symposium on Parallel and Distributed Computing - ISPDC 2012*, June 2012. accepted for publication.

[9] David Fincham. Leapfrog rotational algorithms. *Molecular Simulation*, 8:165–178, 1992.

[10] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[11] Intel Corporation. Intel c++ compiler user and reference guides, 2009. Document Number 304968-023US.

[12] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 3A and 3B, 2011. Document Number 325462-039US.

[13] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, first edition, 2007.

[14] Lennart Nilsson. Efficient table lookup without inverse square roots for calculation of pair wise atomic interactions in classical simulations. *Journal of Computational Chemistry*, 30(9):1490–1498, 2009.

[15] D. C. Rapaport. Multibillion-atom molecular dynamics simulation: Design considerations for vector-parallel processing. *Computer Physics Communications*, pages 521–529, 2006.

[16] Jadran Vrabec. Vorlesungsskript: Molekulare Thermodynamik, 2008.

[17] D. Wolff and W.G. Rudd. Tabulated potentials in molecular dynamics simulations. *Computer Physics Communications, vol. 120, Issue 1*, pages 20–32, 1999.