Speicher-Optimierungen der Linked-Cells-Datenstruktur in MarDyn

Yang Guo

25. Dezember 2010

Inhaltsverzeichnis

1 Einleitung

Im Fokus dieser Arbeit liegt die Molekulardynamik-Simulation MarDyn, die seit Sommer 2005 am Lehrstuhl für wissenschaftliches Rechnen an der Technischen Universität München entwickelt wird. Mit beteiligt sind das Institut für Technische Thermodynamik und Thermische Verfahrenstechnik an der Universität Stuttgart, der Lehrstuhl für Thermodynamik und Energietechnik an der Universität Paderborn und der Lehrstuhl für Thermodynamik an der Universität Kaiserslautern. Ziel von MarDyn ist, eine hochgradig skalierbare und performante Simulationsumgebung zu schaffen, die auf Hochleistungsrechnern in Mehrprozessorbetrieb ausgeführt werden kann.

Aufgrund der Fachlichkeit ist MarDyn ein interdisziplinäres Projekt, da nicht nur Anwendungswissen aus Physik, Verfahrenstechnik etc. eine wichtige Rolle spielen, um das Modell zu definieren, sondern auch Mathematik und Informatik, um eine korrekte und effiziente Umzusetzung zu ermöglichen.

In einer Simulation werden nicht selten mehrere Millionen Moleküle berechnet. Dementsprechend ist eine ausgefeilte Datenstruktur sehr wichtig. In dieser Arbeit wird die bisherige Version der Linked-Cells-Datenstruktur, die in MarDyn eingesetzt wird, untersucht, Verbesserungsmöglichkeiten erörtert, implementiert und sowohl theoretisch als auch experimentell evaluiert.

2 Ausgangsimplementierung

In diesem Abschnitt wird die ursprüngliche Version von MarDyn erläutert, um eine gewisse Grundlage für die später vorgestellten Optimierungsansätze zu haben. Dabei wird ein grobes Vorwissen über Molekulardynamik-Simulation vorausgesetzt, die mit den Werken [1] und [2] erworben werden können.

2.1 Simulationsablauf

Der Ablauf von MarDyn lässt sich grob in drei Phasen einteilen: Initialisierung, Simulation und Nachbereitung. Lediglich die Simulation interessiert uns. Sie besteht aus einer Schleife, um die gleichen Schritte in jedem Zeitschritt zu wiederholen. Die wichtigsten Bestandteile dabei sind:

- Die Domain-Dekomposition in einen gültigen Zustand überführen.
- Die Datenstruktur in einen gültigen Zustand überführen. Dies wird im Folgenden noch detailierter beschrieben.

- Mit Hilfe der Datenstruktur relevante Molekülpaarungen ermitteln und damit die wechselwirkenden Kräfte berechnen.
- Die Kräfte, die auf einzelne Komponenten eines Moleküls wirken, zusammenfassen.
- Aus dem ermittelten Kraftvektor die neue Position der Moleküle im nächsten Zeitschritt bestimmen. Dieser Teil ist irrelevant für die Themenstellung dieser Arbeit.

2.2 Domain

Die simulierte Domain, die alle Moleküle beinhaltet und in der die Simulation abläuft, wird für die parallele Ausführung in Subdomains partitioniert, so dass jeder Prozess genau für eine Subdomain verantwortlich ist. Bewegt sich im Laufe der Simulation ein Molekül aus der Subdomain, so wird es zur passenden benachbarten Subdomain überwiesen. Liegt eine torusförmige Topologie vor, so haben alle Subdomains sechs Nachbarn[1]. Die einzelnen Subdomains sind dabei keineswegs isoliert voneinander, Moleküle einer Subdomain interagieren mit den Molekülen in den Nachbar-Subdomains. Daher werden Moleküle in jedem Simulationsschritt zwischen den Subdomains ausgetauscht und im sogenannten Halo-Bereich gehalten, die räumlich eigentlich nicht zu den einzelnen Subdomains gehören.

Für den speziellen Fall der sequentiellen Ausführung besteht die Domain aus einer einzigen Subdomain, der wegen der torusförmigen Topologie mit sich selbst Nachbar ist. Dabei beinhaltet z.B. der untere Halo-Bereich Moleküle an der obere Grenze des Domains. Deshalb entsteht der Anschein, Moleküle würden mehrfach in der Simulation auftauchen, da sie für den Halo-Bereichen dupliziert werden.

Um die Halo-Bereiche aktuell zu halten, müssen in jedem Zeitschritt Moleküle nahe der Subdomain-Grenzen an die Nachbar-Prozesse übertragen werden.

2.3 Particle Container

Um die Moleküle in einer Subdomain strukturiert und für die Anforderungen gerecht abzuspeichern, hat jede Subdomain eine Instanz des Particle Containers[2], für den es in MarDyn momentan zwei Implementierungen exis-

[1] vorausgesetzt, die Simulation befindet sich im drei-dimensionalen Raum

[2] `datastructures/ParticleContainer.cpp`

tieren: Adaptive Subcells und Linked Cells. Diese Arbeit befasst sich mit letzterem.

Die wichtigsten Anforderungen, die der Particle Container erfüllen muss, sind:

- ein Iterator über alle der Subdomain zugehörigen Moleküle.
- eine Methode, um alle relevanten Molekülpaarungen zu ermitteln und diese an die Algorithmen zu übergeben, die sich um die Kräfteberechnung kümmern.
- eine Methode, um alle grenznahen Moleküle für die Halo-Bereiche der Nachbar-Subdomains für den Austausch zu ermitteln.
- Methoden, um Moleküle hinzuzufügen und zu entfernen.

Der Umstand, dass die zu berechnenden Kräfte zwischen den Molekülen sehr kurzreichend sind, hilft dabei, die Komplexität der Paarbildung einzuschränken. Der Kraftbetrag fällt sehr stark mit wachsendem Abstand, so dass ab einem sogenannten Cutoff-Radius der Kraftbetrag vernachlässigbar gering ist, so dass für ein bestimmtes Molekül M alle Moleküle außerhalb dieses Cutoff-Radius ausgehend von M für die Kraftberechung irrelevant werden.

Um das auszunutzen, wird die Subdomain in würfelförmige Zellen unterteilt. Wird für ein Molekül M die für seine Kraftberechnung relevanten Nachbarmoleküle benötigt, so müssen lediglich die Zelle, in der M befindet, und die Nachbarzellen innerhalb des Cutoff-Radius durchsucht werden.

Implementiert wird Linked Cells[3] folgendermaßen:

- Alle Moleküle in der Subdomain werden in einer STL-Liste[4] festgehalten. Der Iterator von Linked Cells ist der Iterator dieser verketteten Liste.
- Jede Zelle wird durch eine Instanz der Klasse `Cell` repräsentiert. Alle Zellen werden in einem STL-Vektor aufgereiht[5]. In jeder Zelle werden alle darin befindlichen Moleküle mit einer STL-Liste von Zeigern referenziert.

[3] `datastructures/LinkedCells.cpp`

[4] eine Implementierung der Standard Template Library für die verkettete Liste

[5] eine Implementierung der Standard Template Library für den dynamisch reallozierenden Array

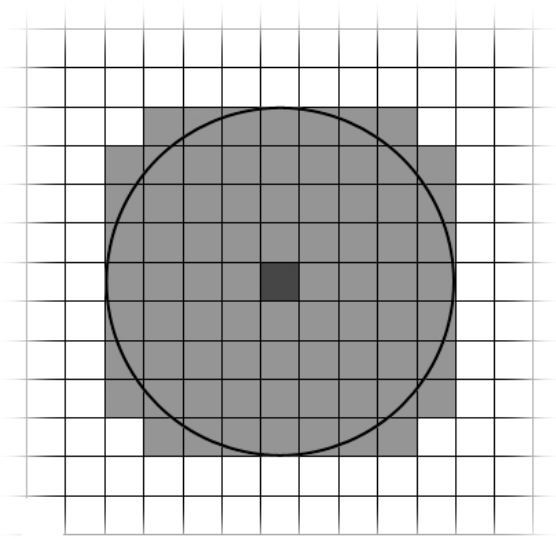

Abbildung 1: Eine Zelle (schwarz) sowie ihre Nachbarzellen (grau) bei einem Cutoff-Radius von 4 Zellenlängen in einer zweidimensionalen Domain

- Am Anfang von jedem Zeitschritt werden in der Methode `update` alle Referenzen in den Zellen neu erstellt, so dass Moleküle immer durch die korrekte Zelle referenziert wird.
- Zwei STL-Vektoren halten Offsets fest, um vom Index einer Zelle die relevanten Nachbarzellen-Indizes zu erhalten. Diese Vektoren werden beim Initialisieren erstellt.
- Weitere STL-Vektoren halten Indizes für Halo-, Grenz-Zellen und innere Zellen fest, so dass diese bei der Paarbildung unterschiedlich behandelt werden können. Redundant dazu wird der Zellentyp auch in den `Cell`-Objekten gespeichert.
- Die räumliche Position der Zelle leitet sich implizit aus dem Zellenindex ab. Die Methode `getCellIndexOfMolecule` berechnet aus der Position des Moleküls den zugehörigen Zellenindex.
- Bei der Paarbildung der Moleküle werden über alle Zellen iteriert und in jeder Zelle jedes Molekül mit jedem anderen Molekül in derselben Zelle und in den Nachbarzellen gepaart, falls der Abstand kleiner als der Cutoff-Radius ist.

2.4 Molekül-Objekt

Die eigentlichen Daten über ein Molekül werden jeweils in eine Instanz der Klasse `Molecule`[6] abgelegt (Abb. 2). Dazu gehören elementare Attribute wie

- die Position.
- der Geschwindigkeitsvektor.
- die räumliche Orientierung.
- die Drehgeschwindigkeit.
- die Molekül-ID zur Unterscheidung.
- die Component-ID, um den Typ des Moleküls festzuegen (z.B. ein Methan-Molekül). Daraus leiten sich auch die Eigenschaften des Moleküls ab.
- Referenzen auf gemeinsame Eigenschaften aller Moleküle des selben Typs, beispielsweise die relative Position und Orientierung von Lennard-Jones-Zentren.

Daneben sind weitere Attribute, die Zwischenwerte speichern, in dem Molekül-Objekt enthalten:

- die auf dem Molekül einwirkende Kraft.
- eine Reihe von Zeiger auf die Eigenschaften des Molekültyps, die beschreibt, aus welchen Bestandteilen das Molekül besteht.
- ein Cache für aktuelle Werte der Bestandteile des Moleküls. Diese Werte werden anhand der Molekül-Position und -Orientierung in jedem Zeitschritt neu berechnet und im Cache als Zwischenergebnis für weitere Berechnungen festgehalten.
- eine Reihe von Zeigern, die als Abkürzung in unterschiedliche Abschnitte des Caches zeigen und so den Zugriff für bestimmte Daten im Cache erleichtern.
- ein Zeiger-Array für Tersoff-Nachbar-Molekülen.

Es ist offensichtlich, dass diese Zwischenwerte nicht zwingend notwendig sind, um ein Molekül ausreichend zu beschreiben. Wegen dem Cache

[6] `molecules/Molecule.cpp`

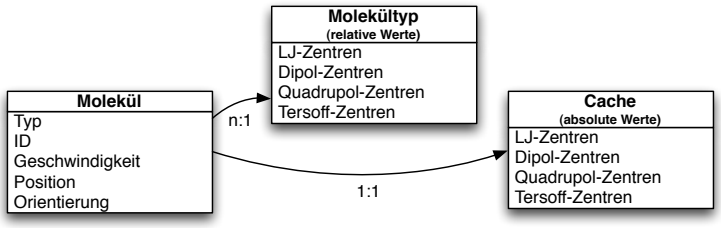

Abbildung 2: Ein Molekül verweist auf die gemeinsame Eigenschaften aller Moleküle des gleichen Typs und besitzt einen Cache

ist außerdem der Platz, den ein Molekül im Speicher einnimmt, weder zusammenhängend noch gleich groß. Aus Gründen der Einfachheit und Effizienz wird daher für den Austausch zwischen den Subdomains die Klasse `ParticleData`[7] verwendet, die nur die elementaren Attribute beinhaltet, so dass alle zu versendende Moleküle als ein Array versandt werden können. Aus den `ParticleData`-Objekten wird nach dem Empfangen wieder vollwertige `Molecule`-Objekte erzeugt und darin der Cache neu angelegt.

3 Optimierungsmöglichkeiten

Dieser Abschnitt beschreibt Alternativen, die den Speicherbedarf verringern oder die Simulationszeit verkürzen können. Dabei werden auch Details in der Umsetzung erläutert.

3.1 Optimierung 1: Datenlokalität

Praktisch alle moderne Rechenarchitekturen verwenden Speicherhierarchien, um die Vorteile von schnellem, aber teurem Speicher wie beispielsweise SRAM mit den der langsameren, aber dafür günstigen Speicher wie z.B. DRAM oder Festplatte zu verbinden: sind erforderliche Daten im Primärspeicher nicht verfügbar (ein sogenannter Cache-Miss), so wird ein Teil dessen Inhalts in den Sekundärspeicher ausgelagert und durch die benötigten Daten aus dem Sekundärspeicher ersetzt. So kann man bei oft verwendeten Daten die hohe Geschwindigkeit des Primärspeichers ausnutzen, muss aber dabei nicht auf die große Kapazität des Sekundärspeichers verzichten.

[7] `parallel/ParticleData.h`

Dabei werden die Daten zwischen den Schichten der Speicherhierarchie in Blöcken übertragen, da der Aufwand, einen ganzen Block zu übertragen, nur geringfügig größer ist als der Aufwand, einzelne Bytes zu übertragen. Für den Entwickler ist die Speicherhierarchie jedoch durch das Konzept des virtuellen Speichers transparent. Dabei hat das blockweise Übertragen die Folge, dass der Zugriff auf eng beieinander liegende Daten besonders effizient ist.

Datenlokalität hat auch weitere Vorteile. Möchte man z.B. die Simulation mit Hilfe von GPGPU[8] beschleunigen, so muss man in der Regel die Daten für die GPU in kleine Portionen zerlegen. Liegen die Daten verstreut im Speicher, so wiegt der Aufwand, diese Daten zusammenzustellen, den Performanzgewinn durch die GPU-Nutzung möglicherweise schon wieder auf.

3.1.1 Ausgangsimplementierung

Eine wünschenswerte Datenlokalität ist in der ursprünglichen Version von Linked Cells nicht gegeben. Die STL-Liste, in der alle Moleküle einer Subdomain enthält, wird als verkettete Liste nicht zusammenhängend im Speicher angelegt. Dadurch ist es möglich, dass die Molekül-Objekte an unterschiedlichsten Stellen im Speicher verstreut werden. So könnte das Iterieren über alle Moleküle und auch die Paarbildung viele Cache-Misses erzeugen, da die räumliche Nähe der Moleküle sich nicht im Speicher widerspiegelt.

Besonders deutlich wird das Problem, wenn es sich beim Sekundärspeicher um sehr langsame Medien wie etwa eine Festplatte handelt, so dass Cache-Misses besonders teuer sind.

3.1.2 Alternative

Um dieses Problem zu lösen, sollten Daten, auf die unmittelbar hintereinander zugegriffen werden, auch im Speicher dicht beieinander abgelegt werden.

Da bei der Paarbildung die Subdomain immer Zelle für Zelle abgesucht wird, ist es intuitiv, die Moleküle einer Zelle auf möglichst wenigen Speicherblöcken zu verteilen. Da durch die Abstraktion des virtuellen Speichers kein direkter Einfluss auf die Unterteilung in Blöcken möglich ist, kann man zumindest dafür sorgen, dass für die Moleküle einer Zelle ein zusammenhängender Speicherbereich reserviert wird.

Besonders gut dafür geeignet ist ein dynamischer Array als Datenstruktur, implementiert in der Klasse `stl::vector` des Standard Template Library. In einem Vektor liegen die Elemente zusammenhängend im Speicher.

[8] general purpose computing on graphics processing unit

Aufgrund der Beschaffenheit des dynamischen Arrays jedoch können Referenzen auf einzelne Elemente während der Laufzeit ihre Gültigkeit verlieren.

3.1.3 Umsetzung

Die betroffenen Klassen sind `LinkedCells` und `Cell`. Anstatt einer STL-Liste an Referenzen auf die Molekül-Objekte gibt es nun in jeder Zelle ein STL-Vektor, in dem die Molekül-Objekte abgelegt werden. Wegen dieser Änderung müssen selbstverständlich auch alle Codeteile, die ursprünglich über Referenzen auf die Moleküle zugreifen, angepasst werden.

Eine einzelne Liste für alle Moleküle im Subdomain gibt es nicht mehr: eine STL-Liste von Referenzen zu allen Molekülen ist nicht praktikabel, da Referenzen auf Vektor-Elementen nicht beständig sind. Eine solche Liste ist auch nicht notwendig. Es reicht, den von `LinkedCells` implementierten Iterator umzuschreiben, so dass dieser nun in einer verschachtelten Schleife über die Zellen und die darin befindlichen Moleküle iterieren. Gibt der Iterator ursprünglich immer die gleiche Sortierung der Moleküle wieder, so ist die Reihenfolge nun von der Zellenzugehörigkeit der Moleküle abhängig. Für die Simulation spielt die Reihenfolge jedoch keine Rolle.

Auch die Methode `LinkedCells::update` muss neu implementiert werden. Moleküle, deren Position von einer Zelle zu einer anderen Zelle wandert, migrieren nun auch im Speicher, um die Datenlokalität zu bewahren. Die `update`-Methode iteriert also Zelle für Zelle über alle Moleküle in der Subdomain und überprüft anhand der Position, ob ein Molekül verschoben werden muss.

3.2 Optimierung 2: Zugriffsmuster

Wie im Abschnitt 2.4 erwähnt hat jedes Molekül-Objekt ein Cache, um die aktuellen Daten über die einzelnen Molekülbestandteile (beispielsweise Lennard-Jones-Zentren) zwischenzuspeichern. Die Größe dieses Caches ist zu Compilierzeit unbekannt, da die Anzahl der Komponenten abhängig vom Molekültyp ist. Daher wird dieser Cache im Konstruktor des Molekül-Objektes beim Initialisieren im Speicher reserviert und der Speicherplatz erst beim Auflösen des Molekül-Objektes durch den Destruktor wieder freigegeben.

3.2.1 Ausgangsimplementierung

In der Originalversion ist dieser Cache als drei getrennte Arrays aus `double`-Werten realisiert:

- für die Position der einzelnen Potenzialzentren[9].
- für die Orientierung bei Potentialen, bei den die Orientierung relevant ist, beispielsweise Dipole[10].
- für die beim jeweiligen Potential resultierenden Kraftvektoren.[11].

Desweiteren gibt es eine Reihe von Zeigervariablen, die auf bestimmte Positionen im Cache referenzieren, um z.B. den ersten `double`-Wert zu adressieren, der die Position von Dipolen abspeichert.

Der Cache wird zwar im Lebenszyklus des Molekül-Objektes beim Konstruieren angelegt und beim Destruieren freigegeben, wird aber nur für die Kraftberechnung durch die Paarbildung benötigt. Danach werden die resultierende Kräfte zusammengefasst und der Cache wird erst für den nächsten Zeitschritt wieder benötigt. Dasselbe gilt für die Hilfszeiger.

Eine weitere Beobachtung ist, dass bei der Paarbildung die Zellen dem Index nach abgearbeitet werden und für jede Zelle nur die Nachbarzellen im Cutoff-Radius relevant sind. Es wird also auf eine vorhersehbare Weise auf die Molekül-Objekte und somit dem Cache zugegriffen und dabei immer nur ein kleiner Teil gleichzeitig gebraucht.

Die Idee ist also, einen möglichst schlanken Molekül-Objekt zu haben und erst bei Bedarf dynamisch Speicherplatz dazu zu reservieren und diesen Speicherplatz auch möglichst früh wieder freizugeben. Das Zugriffsmuster auf die Moleküle und insbesondere dem Cache eignet sich sehr gut dafür.

3.2.2 Umsetzung

Die wesentlichen Änderungen am Code finden in der Klasse `Molecule` und der Methode `LinkedCells::traversePairs` statt.

Die Codeteile zum Reservieren bzw Freigeben des Caches werden aus dem Konstruktor und Destruktor entfernt und in Methoden verpackt, die bei Bedarf aufgerufen werden können. Um Speicherplatz zu sparen, werden die drei getrennte Arrays zusammengelegt und als ein Array reserviert, so dass zwei Zeiger-Variablen wegfallen.

Der Platz für die Hilfszeiger wird ebenfalls erst bei Bedarf reserviert und beim Freigeben des Caches auch wieder geräumt. Dazu werden die knapp ein Duzend Zeiger durch einen einzigen Zeiger zweiter Ordnung[12] ersetzt.

[9] `m_sites_d`
[10] `m_osites_e`
[11] `m_sites_F`
[12] also ein Zeiger auf ein Array von Zeigern

Beim Anlegen des Caches wird damit ein Array von Zeiger reserviert, die so doppelt referenzierend auf die entsprechenden Speicheradressen im Cache zeigen. Natürlich müssen auch alle Codeteile, die diese Hilfszeiger verwenden, entsprechend auf die doppelte Referenzierung hin angepasst werden.

Der Teil des Zeitschrittes, in dem auf den Inhalt des Caches zugegriffen wird, lässt sich in etwa so auflisten:

1. In jedem Molekülen werden die Werte im Cache aktualisiert und die Liste der Tersoff-Nachbarn wird geleert.

2. Es wird über alle innere Zellen, iteriert, um die Paarbildung und damit die Kräfteberechnung dortiger Moleküle vorzunehmen. Innere Zellen sind solche, die weiter als der Cutoff-Radius von der Grenze der Subdomain entfernt sind. Daneben werden gefundene Tersoff-Nachbarn in die Liste eingetragen.

3. In den Halo-Zellen werden ebenfalls nach Tersoff-Nachbarn gesucht.

4. Es wird über alle Grenzzellen zwecks Paarbildung, Kräfteberechnung und Suche nach Tersoff-Nachbarn iteriert. Bis auf einige Details ist das Verfahren genauso wie bei den inneren Zellen.

5. Es wird noch einmal über alle innere Zellen iteriert, um die gefundenen Tersoff-Nachbarn zu verarbeiten.

6. Die errechneten Kräfte, die in dem Molekül-Cache abgelegt sind, werden zusammengefasst.

Wir beobachten, dass in jedem dieser Schritte über alle oder zumindest eine Teilmenge der Zellen iteriert werden muss und der Cache zwischen diesen Schritten nicht geräumt werden darf. Es fällt jedoch auf, dass diese 6 Schleifen durch eine einzige Schleife ersetzt werden kann, die über alle innere Zellen und Grenz-Zellen iterieren und dabei in jeder Iteration die Aufgaben all dieser Schritte für eine Zelle ausführt. Ein Konflikt wegen Datenabhängigkeit besteht nicht.

Dabei ist die Zahl der Moleküle, auf die in einer Iteration zugegriffen werden, begrenzt: ist die Iteration bei der Zelle mit dem Index i, so sind alle Nachbarzellen im Cutoff-Radius durch die Offset-Liste der Nachbarn gegeben. Sei o_{min} und o_{max} das kleinste bzw. größte Offset, so sind die benötigten Nachbar-Moleküle auf jeden Fall in den Zellen mit den Indizes von $i + o_{min}$ bis $i + o_{max}$ zu finden. Lediglich für Moleküle in diesem Fenster von Zellen müssen die Molekül-Caches angelegt sein.

In der darauffolgenden Iteration, in der die Zelle mit dem Index $i + 1$ behandelt wird, verschiebt sich das Fenster um eine Zelle. Die Zelle $i + o_{min}$ wird nicht länger gebraucht, die darin befindlichen Moleküle können ihren Cache freigeben. Dagegen muss auf Moleküle in der Zelle $i + o_{max} + 1$ zugegriffen werden, so dass am Anfang dieser neuen Iteration die Caches der Moleküle in dieser Zelle angelegt und mit den aktuellen Werten initialisiert werden müssen.

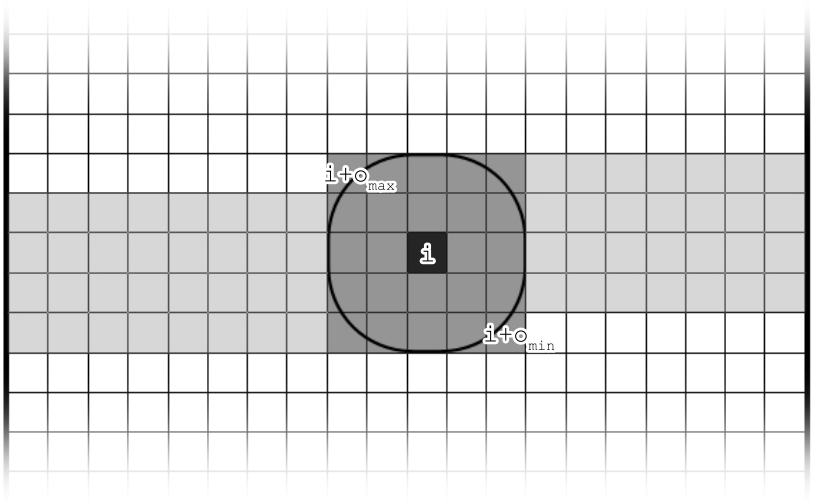

Abbildung 3: Fenster mit den Zellen, in denen die Caches Molekül-Objekte reserviert sind, wenn die Iteration die i-te (schwarze) Zelle behandelt. Der Cutoff-Radius ist 2.

4 Komplexitätsanalyse

In diesem Abschnitt werden wir die verbesserte Version einer asymptotische Komplexitätsanalyse sowohl bezüglich Performanz als auch Speicherverbrauch unterziehen. Da MarDyn als Ganzes zu komplex ist, werden wir stattdessen die Bestandteile einzeln betrachten.

Klar ist, dass ein konstanter Faktor bei der asymptotischen Komplexität nicht abgeschätzt wird und auch abhängig vom Szenario schwanken kann. Daher sind auch Performanz-Messungen an realen Szenarien notwendig.

Wir bezeichnen die Ausgangsimplementierung als A und die optimierte Version als B.

4.1 Performanz

Neben der üblichen Betrachtungsweise im Rahmen des RAM-Modells[13], in dem alle Speicherzugriffe gleich kosten, werden wir darüberhinaus auch das Sekundärspeichermodell[14] hinzuziehen, um die Vorteile der Datenlokalität besser verdeutlichen zu können. Zur besseren Unterscheidung verwenden wir ausschließlich für das RAM-Modell die Groß-O-Schreibweise.

4.1.1 Sekundärspeichermodel

Die Idee hinter dem Sekundärspeichermodell ist, dass der eigentliche Flaschenhals nicht die Rechenleistung des CPU ist, sondern die Speicherbandbreite und -latenz zwischen den einzelnen Schichten der Speicherhierarchie. Vereinfacht werden dabei mit einem Primärspeicher und einem Sekundärspeicher nur zwei Schichten abgebildet.

Die kleinste Einheit sei dabei der Speicherverbrauch eines Datensatzes, in unserem Fall die eines Moleküls. Als vereinfachtes Modell betrachten wir die Moleküle als Datensätze fester Größe, die einen zusammenhängenden Speicherbereich belegen. Ein einfaches Molekül-Objekt belegt in der Regel rund 500 Bytes im Speicher.

Weiter nehmen wir an, dass die N zu bearbeitenden Datensätze (entsprechen N Moleküle) im Sekundärspeicher liegen und in Blöcken gleicher Größe B partitioniert sind. Zur Bearbeitung müssen die Datensätze blockweise in den Primärspeicher mit der Kapazität M kopiert werden. Da der Primärspeicher in der Regel zu klein ist, um alle N Datensätze aufzunehmen, muss zwischen dem Primär- und dem Sekundärspeicher ein- und ausgelagert werden. Die Kosten entsprechen der Anzahl der Kopieroperationen.

Zusätzlich zu N, M und B definieren wir C als Anzahl der Zellen.

Um die Größen in Relation zu setzen, betrachten wir den L2-Cache und Arbeitsspeicher eines handelsüblichen Desktop-Rechner[15]. Ersteres entspricht dem Primärspeicher und hat eine Größe von 4 MB; letzteres entspricht dem Sekundärspeicher und hat eine Größe von 4GB. Die übliche Seitengröße für den virtuellen Speicher ist dabei 4KB. Demnach ist $B \approx 8$ und $M \approx 8000$.

Im Folgenden betrachten wir immer den *worst case*, da im *best case* die Daten optimal im Speicher abgelegt würden und der *average case* immer vom Szenario abhängt.

[13] random access machine
[14] external memory model
[15] Stand 2010

Details wie z.B. feste Größe der Cache Lines oder Assoziativität des Caches wollen wir ignorieren, da diese Eigenheiten nicht für alle Ebenen einer Speicherhierarchie zutreffen. Das Sekundärspeichermodell soll nicht nur das Verhalten zwischen dem L2-Cache und dem Arbeitsspeicher abbilden, sondern auch zwischen dem Arbeitsspeicher und der Auslagerungsdatei auf der Festplatte.

4.1.2 STL-Vektor

Eine der wichtigsten Änderungen in der `LinkedCells`-Klasse ist das Ersetzen einer verketteten Liste, die alle Moleküle einer Subdomain enthält, durch Vektoren in den einzelnen Zellen.

Ähnlich zu einem Array wird der Speicher des Vektors zusammenhängend reserviert. Allerdings kann man zur Laufzeit weitere Elemente zu einem Vektor hinzufügen, für die ursprünglich kein Platz reserviert wurde. Reicht der bisher reservierte Speicher nicht mehr, so wird ein größerer Speicherbereich reserviert und die bisherigen Elemente dorthin verschoben. Die Größe des dabei reservierten Speicherplatzes wächst in Zweierpotenzen. Die Anzahl der Elemente, die in den reservierten Bereich passen, heißt Kapazität des Vektors. Die Anzahl der Elemente, die sich tatsächlich im Vektor befinden, ist die Größe.

Sind beispielsweise 5 Elemente im Vektor abgelegt, so ist die Kapazität 8, so dass 3 weitere Elemente ohne weiteres an das Ende des Vektors angefügt[16] werden können. Fügen wir nun ein neuntes Element hinzu, so wird ein neuer Speicherbereich für 16 Elemente reserviert, die 8 bisherigen Elemente dorthin kopiert und der alte Speicherplatz aufgegeben.

4.1.3 Zugriff auf Moleküle

Das Iterieren über alle Moleküle einer Subdomain kostet in B wie in A $O(N)$. Denn die Adresse eines Elements lässt sich aus dem Index direkt berechnen, so dass der Zugriff auf jedes Molekül $O(1)$ kostet.

Wird über alle Moleküle im Subdomain iteriert, so muss in A im *worst case* mit N Cache-Misses gerechnet werden, in B jedoch nur $\frac{N}{B} + C$. Denn jede Zelle mit n_c Molekülen belegt einen zusammenhängenden Speicherplatz, der sich über maximal $\frac{n_c}{B} + 1$ Blöcken ersteckt. Bei ausreichend großem B und im Vergleich zu N kleinem C erzeugt die optimierte Version weniger Cache-Misses.

[16] Methode `stl::vector::push_back`

Das Hinzufügen von Molekülen in eine Subdomain bleibt $O(1)$. Das ist auch dann gegeben, wenn die Kapazität vergrößert werden muss. Zwar müssen die 2^n bisherigen Elemente kopiert werden, wenn die bisherige Kapazität von 2^n auf 2^{n+1} erweitert wird. Diese Kosten amortisieren sich jedoch über die nächsten 2^n Elemente, für die bereits Kapazität im Voraus geschaffen wurde.

4.1.4 Aktualisieren der Zellenzugehörigkeit

In A kostet die `update`-Methode $O(C) + O(N)$, da zunächst in allen Zellen die Referenzen gelöscht und danach über alle N Moleküle iteriert wird, um diese den Zellen neu zuzuordnen.

Auch in B wird über alle Moleküle iteriert und die korrekte Zellenzugehörigkeit überprüft. Falls ein Molekül von Zelle C_1 in Zelle C_2 migriert, wird das Molekül in C_1 gelöscht und in C_2 hinzugefügt. Die `erase`-Methode des Vektors würde allerdings $O(n)$ (n ist hier die Größe des Vektors) kosten, weil alle nachfolgende Elemente aufrücken. Da die Reihenfolge der Elemente egal ist, wird stattdessen das letzte Element an die Stelle des gelöschten Elements kopiert und das letzte Element gelöscht, so dass das Löschen $O(1)$ kostet.

Um allerdings zu verhindern, dass Moleküle nach dem Verschieben erneut überprüft werden, wird zu aller Anfang das letzte Molekül im Vektor jeder Zelle markiert. Alle neu hinzukommenden Moleküle in der Zelle werden hinter der Markierung in den Vektor hinzugefügt und müssen nicht mehr überprüft werden.

Das Markieren und Überprüfen kosten zusammen $O(C) + O(N)$. Nicht vergessen sollte man allerdings, dass das Verschieben von Molekülen im Speicher bei B mit einem viel größeren Aufwand verbunden ist als das Erstellen von Referenzen bei A. Jedoch muss dieser Aufwand nur bei migrierenden Molekülen erbracht werden. Bei A ist das erneute Referenzieren unabhängig von den Molkülbewegungen. Bei Simulationen mit starker Molekülbewegung kann die Performanz also bei B schlechter sein. Umgekehrt kann A langsamer sein, wenn Moleküle zumeist an ihre Positionen verharren.

4.1.5 Paarbildung von Molekülen

Im RAM-Modell unterscheiden sich A und B nur unwesentlich. In beiden Versionen werden über alle Zellen iteriert und darin über alle Moleküle in der Zelle und in den Nachbarzellen iteriert.

Im Sekundärspeichermodell jedoch ist die Sache etwas komplizierter. Es

ist vorstellbar, dass $\frac{M}{B}$ kleiner ist als die Anzahl der Moleküle einer Zelle samt Nachbarschaft und diese Moleküle auch noch über möglichst vielen Blöcken verteilt sind, so dass im *worst case* laufend Cache-Misses auftreten.

Wir rechnen damit, dass die Moleküldichte begrenzt ist, so dass die Anzahl der Moleküle einer Zelle samt Nachbarzellen durch eine Konstante k begrenzt ist. Darüberhinaus wird in A für die Kräfteberechnung 6 mal über alle oder ein Teil der Zellen iteriert. Eine grobe Obergrenze für die Zahl der Cache-Misses ist also $6Nk$. Je nach Zellengröße, Moleküldichte und Cutoff-Radius ist ein k in der Größenordnung 10^5 durchaus vorstellbar.

In B jedoch wird ein Fenster mit den Zellen, die zuletzt gebraucht werden, im Primärspeicher behalten. Dieses Fenster entspricht genau dem Fenster der Zellen, in den die Molekül-Caches reserviert sind (Abb. 3), vorausgesetzt natürlich, die Blöcke dieses Fensters passen ins Primärspeicher. Wir erinnern uns, dass dank der Datenlokalität die Moleküle nicht verstreut, sondern in relativ wenigen Blöcken versammelt sind. In diesem Fall treten maximal $\frac{N}{B} + C$ Cache-Misses auf, da jedes Molekül nur einmal in den Primärspeicher geladen werden muss.

Passt das Fenster nicht ins Primärspeicher, so werden $N\left(\frac{k}{B} + m + 1\right)$ Cache-Misses maximal erwartet. Dabei ist m die Anzahl der Nachbar-Zellen. Offensichtlich wird über N Moleküle iteriert und bei jedem Molekül die gesamte Nachbarschaft von maximal k Molekülen neu in den Primärspeicher geladen. Im ungünstigsten Fall erstrecken sich alle Molekül-Vektoren in den Zellen über einen Block mehr als sie eigentlich an Speicher verbrauchen.

Mit realistischen Werten besetzt sind in beiden Fällen Version B günstiger. Dazu ein einfaches Beispiel, bei dem wir den L2-Cache als Primärspeicher und den Arbeitsspeicher als Sekundärspeicher betrachten:

- Die Domain bestehe aus $20 \times 20 \times 20$ Zellen zu je 100 Molekülen so dass $N = 800\,000$.
- Mit einem Cutoff-Radius gleich der Zellengröße ist $m = 8$ und $k = 900$.
- B sei 8.
- Vor den Änderungen würde das $6Nk \approx 4 \times 10^9$ Cache Misses im *worst case* erzeugen.
- Das Fenster der Zellen bestünde aus 42 Zellen mit insgesamt 4200 Molekülen und würde 2,1MB Speicher belegen.
- Ist der L2-Cache dafür groß genug, so vermelden wir $\frac{N}{B} + C \approx 10^5$ Cache-Misses.

- Ist der L2-Cache zu klein, so vermelden wir $N\left(\frac{k}{B}+m+1\right) \approx 10^8$ Cache-Misses.

4.1.6 Leeren der Halo-Zellen

Die Halo-Zellen werden zwischen zwei Zeitschritten immer geleert und im Zuge der Domain-Dekomposition neu befüllt. Bei A müssen dabei die Referenzen in den Zellen gelöscht werden und darüber hinaus in der Liste aller Moleküle die Zugehörigkeit überprüfen und bei Bedarf aus der Liste entfernen. Dieser Teil fällt bei B weg, da diese Liste nicht mehr existiert. Lediglich die Vektoren in den Halo-Zellen müssen geleert werden.

4.2 Speicherverbrauch

Letztendlich ist die Konsequenz eines möglichst geringen Speicherverbrauchs eine kürzere Laufzeit: können alle benötigten Daten in höhere Schichten der Speicherhierarchie abgelegt werden, so vermeidet man dadurch das teure Auslagern, beispielsweise vom Arbeitsspeicher auf die Festplatte. Durch die Änderungen werden zum Teil Einsparungen erzielt, an anderen Stellen jedoch wiederum mehr Speicherplatz beansprucht.

4.2.1 Molekül-Cache

Die wichtigste Änderung, um Speicherplatz zu verringern, besteht darin, für den Molekül-Cache nur bei Bedarf Speicherplatz bereitzustellen.

Im schlimmsten Fall befinden sich alle Moleküle der Subdomain in einer einzigen Zelle, so dass sich der Spitzen-Speicherverbrauch sich nicht ändert, denn in diesem Fall müssen alle Moleküle den Cache gleichzeitig bereit halten, wenn die Iteration diese Zelle erreicht. Wir wollen jedoch davon ausgehen, dass die Moleküle in der Subdomain in gleichverteilt sind und die Anzahl der Moleküle pro Zelle in etwa gleich ist.

Der Speicherverbrauch für den Cache zu jedem Zeitpunkt hängt demnach von der Anzahl der Zellen in dem Fenster ab. Diese wiederum ergibt sich aus dem Cutoff-Radius und der Größe der Subdomain ab. Sei der Cutoff-Radius r und die Größe der kubischen Subdomain d^3, so stellt das Fenster in etwa eine Scheibe aus der Subdomain mit der Dicke $2r$ dar. Skalieren wir d bei konstantem r, so haben wir den Speicherverbrauch von $O(d^3)$ bei A auf $O(d^2)$ bei B verringert.

Betrachten wir nun das Beispiel aus 4.1.5. Dazu nehmen wir vereinfacht an, dass ein Molekül 500 Bytes Speicher belegt und davon 100 Bytes für den Molekül-Cache verwendet wird.

- In der Subdomain sind 800 000 Moleküle enthalten. Diese nehmen insgesamt ca. 400 MB Speicher ein.
- Das Fenster besteht aus ca. 2×20^2 Zellen mit 80 000 Molekülen.
- Durch die Änderung sparen wir Speicherplatz für den Cache von 720 000 Molekülen ein. Das entspricht 72 MB und ca. 18%.

Natürlich hängt die Einsparung neben der Größe der Subdomain, dem Cutoff-Radius und Molekülverteilung auch von den Molekültypen ab. Komplexere Moleküle haben mehrere Potential-Zentren und benötigen daher auch einen größeren Cache.

Ein Nachteil hat das Reservieren nach Bedarf: das Belegen und Freigeben in jedem Zeitschritt bei B verursacht mehr Aufwand als das einmalige Initialisieren bei A. Das gilt jedoch nicht für die Moleküle in den Halo-Zellen, die sowieso in jedem Zeitschritt neu erstellt werden.

4.2.2 Vektor

Die Entscheidung, Molekül-Objekte in Vektoren unmittelbar in den Zellen zu speichern, hat zum einen die gewünschte Datenlokalität zur Folge, zum anderen aber auch die Konsequenz, dass in der Regel mehr Speicherplatz reserviert wird als eigentlich benötigt, da die Kapazität des Vektors immer eine Zweierpotenz ist. Das ist notwendig, damit die Komplexität für das Hinzufügen eines Elements sich zu $O(1)$ amortisiert.

Im ungünstigsten Fall besitzt jede Zellen genau $2^n + 1$ Moleküle, so dass die Kapazität mit 2^{n+1} annähernd doppelt so groß ist. Der Brutto-Speicherverbrauch ist fast doppelt so groß wie der Netto-Speicherverbrauch.

Ein weiteres Problem liegt in den Implementationseigenheiten des STL-Vektors: die Kapazität wird nicht angepasst, wenn aus dem Vektor Moleküle entfernt werden. Füllt man einen Vektor mit 65 Elemente, so steigt die Kapazität auf 128 an. Entnimmt man alle Elemente eins nach dem anderen, bleibt die Kapazität jedoch immer noch bei 128. Die Überlegung dahinter ist, dass die maximale Größe eines Vektors ein gutes Indiz für die künftigte Größe ist. Das ist z.B. in Halo-Zellen der Fall, die immer wieder geleert und neu befüllt werden.

Für Szenarien mit viel Molekülbewegungen kann das jedoch ein Problem werden, da Zellen, die anfänglich viele Moleküle beinhalten, im Laufe der Simulation leer werden können. Abhilfe kann man dadurch schaffen, indem man eine kleinere Kapazität erzwingt. Immer wenn die Größe nach dem Entfernen eines Elements kleiner oder gleich der Hälfte der Kapazität ist,

halbiert man die Kapazität durch erneutes Reservieren. Das Entfernen hat nach wie vor einen konstanten Kosten, da das Neureservieren sich amortisiert: sind sowohl Größe und Kapazität 2^{n+1}, so findet das Verschieben von 2^n Elemente erst statt, nachdem bereits 2^n Elemente entfernt worden sind.

Damit ist gewährleistet, dass die Kapazität nicht mehr als das Zweifache der Größe des Vektors ist. Der Brutto-Speicherverbrauch N' ist maximal doppelt so groß wie der Netto-Speicherverbrauch N.

Um die Beziehung zwischen N' und N genauer zu betrachten, beschreiben wir den Füllzustand einer Zelle probabilistisch. Die N Moleküle werden eins nach dem anderen zufällig auf C Zellen verteilt, so dass für jede Zelle ein Bernoulli-Experiment stattfindet: in jedem Versuch bekommt die Zelle mit der Wahrscheinlichkeit C^{-1} ein zusätzliches Molekül. Am Schluss beinhaltet die Zelle X Moleküle. Es ist unschwer zu erkennen, dass der Erwartungswert $E(X) = NC^{-1}$ ist.

Für die realistische Annahme von großen N und kleinem C^{-1} haben wir somit eine Poisson-Verteilung:

$$P(X = k) = \frac{\lambda^k}{k!} e^{-\lambda}$$

Da für Poisson-Verteilungen $E(X) = \lambda$ gilt, erhalten wir

$$\lambda = E(X) = NC^{-1}$$

Ist die Größe einer Zelle X, so ist dessen Kapazität X' die nächst größere Zweierpotenz

$$X' = 2^{\lceil \log_2 X \rceil}$$

Erwartungswert für die Kapazität ist demnach

$$E(X') = \sum_{k=0}^{N} 2^{\lceil \log_2 k \rceil} \cdot \frac{\lambda^k}{k!} e^{-\lambda}$$

Das numerische Ergebnis (Abb. 4) legt nahe, dass - zumindest bei einer Gleichverteilung der Moleküle auf die Zellen - die Größe der Zellen und somit die durchschnittliche Vektorgröße $E(X)$ ausschlaggebend dafür ist, wie groß der Overhead des Brutto- gegenüber dem Netto-Speicherverbrauchs ist. Ist $E(X)$ knapp kleiner als ein Zweierpotenz, so ist der Overhead $E(X') - E(X)$ klein. Ist $E(X)$ hingegen ein wenig größer als ein Zweierpotenz, ist der Overhead besonders groß. Im Allgemeinen schwankt $E(X')$ um etwa 40% über $E(X)$.

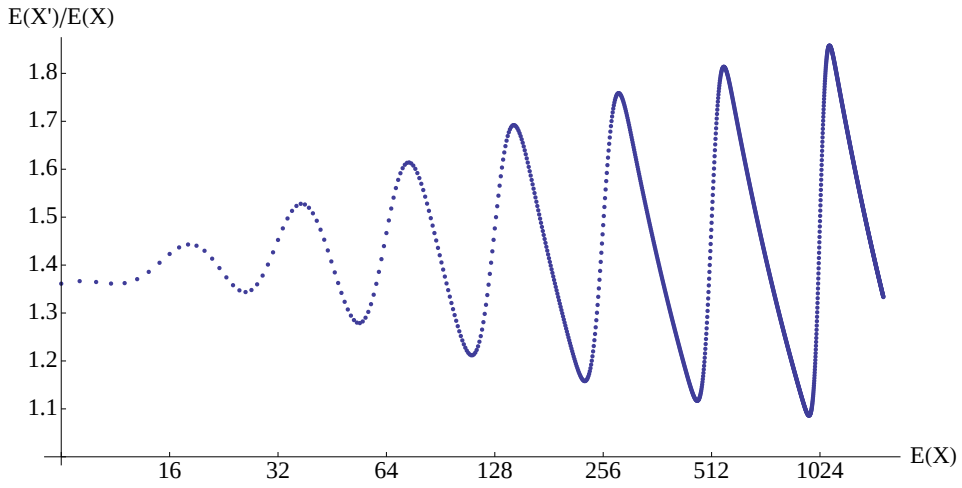

Abbildung 4: Overhead bei verschiedenen Durchschnitts-Zellengrößen

Für praktische Simulationen sind diese Aussagen allerdings nur mit Vorsicht zu genießen, da physikalische Gegebenheiten wie beispielsweise Clustering in diesem stochastischen Modell völlig ignoriert werden. Experimentelle Beobachtungen liefern genauere Aufschlüsse.

5 Empirische Untersuchung

Die theoretischen Überlegungen haben gezeigt, an welchen Stellen Einsparungen und Leistungssteigerungen erzielt worden und welche Einbußen zu verzeichnen sind. Dennoch kann eine eindeutige Aussage, ob zu Laufzeit weniger Speicher verbraucht wird und ob die Simulation schneller läuft, nicht gemacht werden. Das liegt zum einen daran, dass unklar ist, inwiefern einzelne Verbesserungen ins Gewicht fallen, zum anderen auch daran, dass viele Ergebnisse der theoretischen Überlegungen sehr von den Parametern des Szenarios abhängt.

5.1 Eingabedaten

Um unterschiedliche Anwendungsfälle zu betrachten, verwenden wir vier unterschiedliche Eingabedatensätze für die Messungen. Diese Szenarien unterscheiden sich in vielen Aspekten wie beispielsweise Cutoff-Radius, Anzahl der Moleküle, Anfangsverteilung der Moleküle, involvierte Kräfte etc. und

eignen sich daher gut für einen Vergleich.

Für die Messungen beschreiben wir die Versuche als Kombination aus Version und Szenario. A1 etwa bezeichnet Szenario 1 ausgeführt auf der Originalversion von MarDyn.

Da Parallelisierung kaum von den Änderungen betroffen ist und nicht das Thema dieser Arbeit ist, vergleichen wir nur die sequentielle Ausführung mit einem Prozess.

5.1.1 Szenario 1

Der erste Datensatz stellt eine relativ kleine Simulation mit einfach aufgebauten Argon-Molekülen dar. Dabei entsprechen die Lage und Geschwindigkeiten der Moleküle einem bereits weitestgehend eingependelten Zustand.

- Die Simulation umfasst 40 000 Molekülen mit jeweils nur einem Lennard-Jones-Zentrum.
- Der Cutoff-Radius entspricht einer Zellenlänge. Der Halo- und der Grenz-Bereich haben daher jeweils eine Dicke von einer Zellenlänge.
- Die Domain wird in $21 \times 21 \times 21$ Zellen inklusive der Halo-Zellen unterteilt.
- Damit ergeben sich insgesamt 9 261 Zellen, davon 4 913 innere Zellen, 1 946 Grenzzellen und 2 402 Halo-Zellen.
- Das Fenster der Zellen, in den der Molekül-Cache reserviert ist, hat eine Größe von 927 Zellen.
- Im ersten Zeitschritt wird der Halo-Bereich mit 27 421 Molekülen befüllt, so dass sich inklusive der Duplikate insgesamt 67 421 Moleküle in der Domain befinden.
- Die meisten Zellen sind leer, die Moleküldichte ist nicht homogen, wie Abb. 5 zeigt. Der Brutto-Speicherverbrauch im ersten Zeitschritt für Version B ist 87 411.
- Die Moleküle bewegen sich relativ wenig, so dass in jedem Zeitschritt nur wenige Moleküle ihre Zellenzugehörigkeit ändern. So sind es lediglich jeweils 0 bis 4 Moleküle in den ersten 10 Zeitschritten.

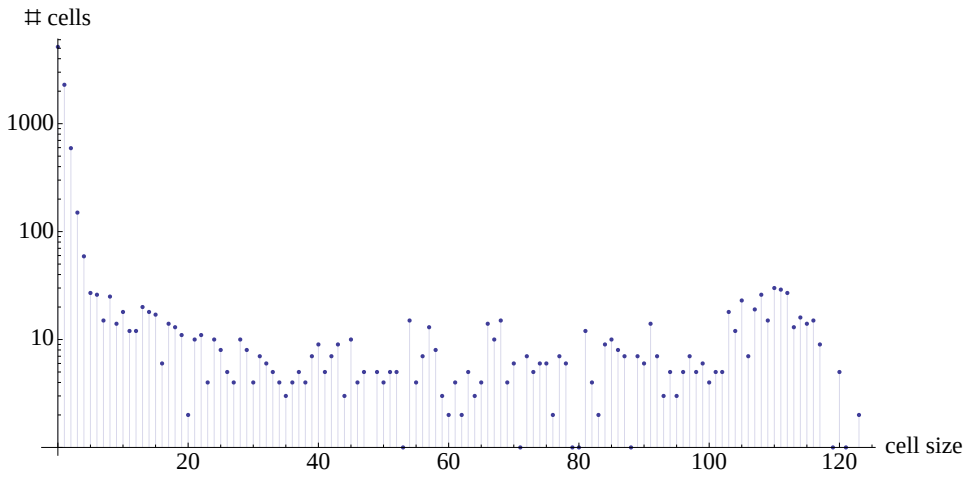

Abbildung 5: Verteilung der Anzahl der Moleküle in den Zellen im Szenario 1

5.1.2 Szenario 2

Szenario 2 unterscheidet sich vom Szenario 1 nur darin, dass die Argon-Moleküle durch Ethylenoxid-Moleküle ersetzt werden und die Moleküle so jeweils drei Lennard-Jones-Zentren und ein Dipol-Zentrum besitzen. Das führt z.B. dazu, dass die Molekül-Caches weitaus mehr Speicherplatz beanspruchen und die Kräfteberechnung aufwändiger ist.

5.1.3 Szenario 3

Der dritte Fall stellt eine etwas umfangreichere Simulation dar. Dabei müssen die Moleküle noch aus ihrem Anfangszustand einpendeln.

- Die Simulation umfasst 1 000 000 Molekülen mit jeweils nur einem Lennard-Jones-Zentrum.
- Der Cutoff-Radius entspricht einer Zellenlänge.
- Die Domain wird in $48 \times 48 \times 48$ Zellen inklusive der Halo-Zellen unterteilt.
- Von insgesamt 110 592 Zellen sind 85 184 innere Zellen, 12 152 Grenzzellen und 13 256 Halo-Zellen.
- Das Fenster hat eine Größe von 4707 Zellen.

- Im ersten Zeitschritt wird der Halo-Bereich mit 123 566 Molekülen befüllt, so dass inklusive der Duplikate insgesamt 1 123 566 Moleküle sich in der Domain befinden.
- Die Zellen sind vergleichsweise gleichmäßig und dünn mit Molekülen besetzt (s. Abb. 6). Der Brutto-Speicherverbrauch im ersten Zeitschritt für Version B ist 1 556 612.
- Die Moleküle bewegen sich relativ viel. In den ersten 10 Zeitschritten etwa migrieren im Schnitt rund 16 000 Moleküle pro Zeitschritt.

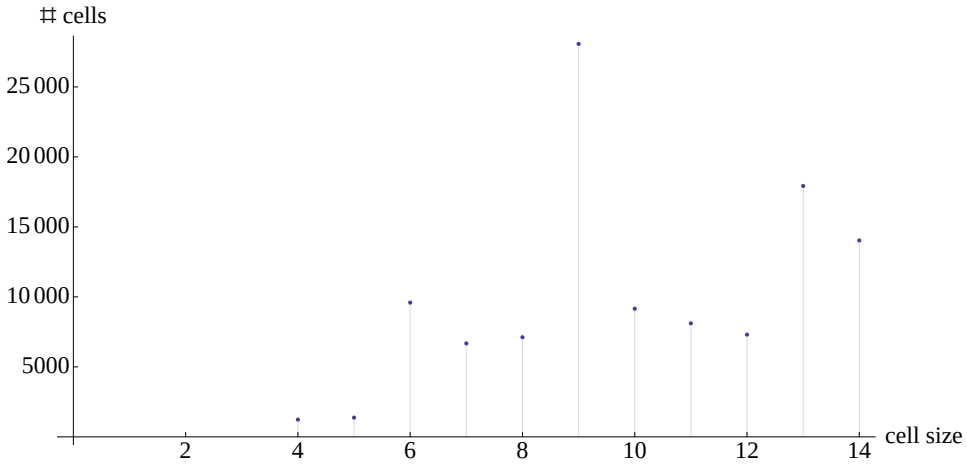

Abbildung 6: Verteilung der Anzahl der Moleküle in den Zellen im Szenario 3

5.1.4 Szenario 4

Der letzte Fall ist sehr klein, beinhaltet jedoch eine Mischung von unterschiedlichen Molekülen, die anfangs sich auf engsten Raum konzentrieren.

- Die Simulation umfasst 10 000 Molekülen zweier verschiedener Arten. Die erste besitzt nur ein Lennard-Jones-Zentrum, die zweite besitzt sowohl ein Lennard-Jones-Zentrum als auch ein Tersoff-Zentrum.
- Der Cutoff-Radius entspricht zwei Zellenlängen.
- Die Domain wird in $10 \times 22 \times 17$ Zellen inklusive der Halo-Zellen unterteilt.

- Insgesamt 3 740 Zellen, davon 252 innere Zellen, 1 152 Grenzzellen und 2 336 Halo-Zellen.
- Das Fenster hat eine Größe von 925 Zellen.
- Im ersten Zeitschritt wird der Halo-Bereich mit 19 527 Molekülen befüllt, so dass inklusive der Duplikate insgesamt 29 527 Moleküle sich in der Domain befinden.
- Die meisten Zellen sind zu Anfang leer. Die Moleküle konzentrieren sich auf wenige Zellen (s. Abb. 7). Der Brutto-Speicherverbrauch für Version B beträgt im ersten Zeitschritt 39 552.
- Die Moleküle bewegen sich relativ viel. In den ersten Zeitschritten etwa migrieren bis zu ca. 1 700 Moleküle pro Zeitschritt.

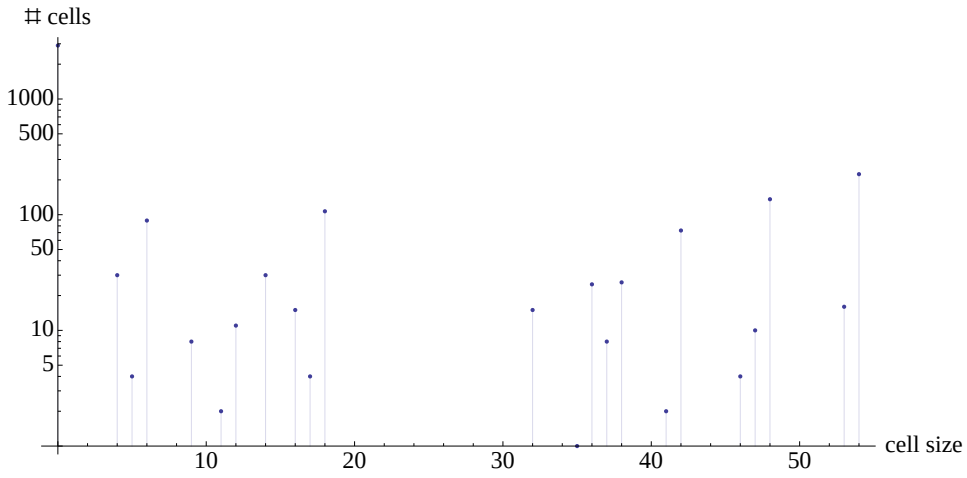

Abbildung 7: Verteilung der Anzahl der Moleküle in den Zellen im Szenario 4

5.2 Testumgebung

Zum Messen der Performanz werden die Versionen A und B mit den oben vorgestellten Szenarien ausgeführt und die Laufzeit gemessen. Dabei ist natürlich der Vergleich wichtiger als die absoluten Werte, die abhängig von der verwendeten Hardware ist. Zusätzlich werden die Versuche an einer mit GProf instrumentalisierten Version ausgeführt, um die Anteile einzelner

Schritte zu untersuchen. Um den Speicherverbrauch zu messen, verwenden wir Valgrind mit Massif.

Die Performanzmessung führen wir auf dem Linux-Cluster am Leibniz-Rechenzentrum durch, da dort kaum Beeinträchtigungen durch nebenläufige Prozesse zu erwarten sind. Die Messung des Speicherverbrauch führen wir auf einer virtuellen Maschine durch, auf dem Ubuntu-Linux läuft, da auf dem Linux-Cluster Valgrind nicht verfügbar ist.

5.3 Performanz

Um die Performanz der beiden Versionen A und B zu vergleichen, messen wir einfach die Laufzeit bei den vier Szenarien. Dabei führen wir die Simulation immer 20 Zeitschritte aus und vergleichen die Zeit für die reine Berechnung und zum anderen die Gesamtdauer, die auch Ausgaben zwischen jedem Zeitschritt und zum Schluss beinhaltet. Um den statistischen Fehler zu verringern, führen wir jede Messung 10 mal durch und ermitteln den Durchschnitt.

	1	2	3	4
A - Rechenzeit	29,7	95,2	193,4	14,1
A - Gesamtzeit	34,0	103,0	244,3	16,8
B - Rechenzeit	26,1	90,7	182,8	12,9
B - Gesamtzeit	30,1	98,1	233,0	15,5
Einsparung - R	12,0%	4,8%	5,5%	8,8%
Einsparung - G	11,3%	4,7%	4,6%	7,8%

Abbildung 8: Laufzeit bei 20 Zeitschritten in Sekunden

Wie wir in Tabelle 8 sehen, lassen sich bei allen vier Szenarien eine kürzere Laufzeit beobachten. Am deutlichsten ist die Verbesserung bei Szenario 1, bei dem 12% der Rechenzeit einsparen lässt.

Mit Hilfe von GProf lässt sich die benötigte Rechenzeit besser analysieren. Wir recompilieren A und B mit der Compiler-Option `-p` für das Profiling mit GProf und führen A1 und B1 jeweils 20 Zeitschritte lang aus. Wenn auch durch das Profiling die Laufzeit verzerrt wird, so spiegelt das Ergebnis mit 258 Sekunden bei A und 185 Sekunden bei B für die reine Rechenzeit die Verbesserung wider. Anhand der Messwerte lassen sich also zumindest Aussagen machen, welche Stellen für die Verbesserung verantwortlich sind.

Die Tabelle 9 zeigt Messungen einiger Methoden inklusive Unteraufrufe.

Daraus lassen sich einige Schlüsse ableiten:

Methodenname	Laufzeit in Sek.		Anteil in %	
	A	B	A	B
`BlockTraverse::traversePairs`	247,0	177,4	95,7	95,8
→ `PotForce`	153,6	99,1	59,5	53,5
→ `Molecule::numCharges`	28,9	17,7	11,2	9,6
→ `Molecule::numLJcenters`	27,2	16,3	10,5	8,8
→ `Molecule::numQuadrupoles`	26,9	16,9	10,4	9,1
→ `Molecule::numDipoles`	26,8	17,0	10,4	9,2
→ `Molecule::numTersoff`	14,9	10,1	5,8	5,5
→ bzgl. `std::list<Molecule*>`	28,6	-	11,1	-
→ bzgl. `std::vector<Molecule>`	-	26,9	-	14,5
`LinkedCells::update`	0,92	0,38	0,01	0,02
`LinkedCells::addParticle`	1,24	1,11	0,5	0,6
`Cell::prepareCache`	-	1,69	-	0,9
`Cell::dismissCache`	-	0,14	-	0,02

Abbildung 9: Ausschnitt aus dem GProf-Ergebnis

- Die Kräfteberechnung durch die Paarbildung nimmt die meiste Rechenzeit ein.
- Einzelne einfache Methoden des Molekül-Objekts sind deutlich schneller geworden.
- Das Neuzuordnen der Moleküle in die korrekte Zelle ist bei B günstiger als bei A, zumindest bei relativ wenigen Molekülbewegungen.
- Die Datenstrukturen `std::list` und `std::vector` haben etwa den gleichen Aufwand.
- Das Reservieren und Freigeben des Molekül-Caches in jedem Zeitschritt ist vernachlässigbar.

Der beobachtete Performancegewinn lässt sich also vor allem damit erklären, dass Zugriffe auf die Molekül-Objekte günstiger geworden sind. Da die Methoden in der Klasse `Molecule` sich nicht geändert haben, ist eine kürzere Zugriffszeit dank Datenlokalität vermutlich die beste Erklärung.

5.4 Speicherverbrauch

Wir betrachten die Szenarien getrennt und messen die Veränderungen beim Speicherverbrauch während der Laufzeit. Die Zeit wird dabei in Anzahl der

Programm-Instruktionen gezählt, entspricht also nicht unbedingt der realen Laufzeit.

5.4.1 Szenario 1

Führt man Szenario 1 auf A und B jeweils drei Zeitschritte lang aus und zeichnet den Speicherverbrauch auf, so erhält man die Graphen in Abb. 10 und 11.

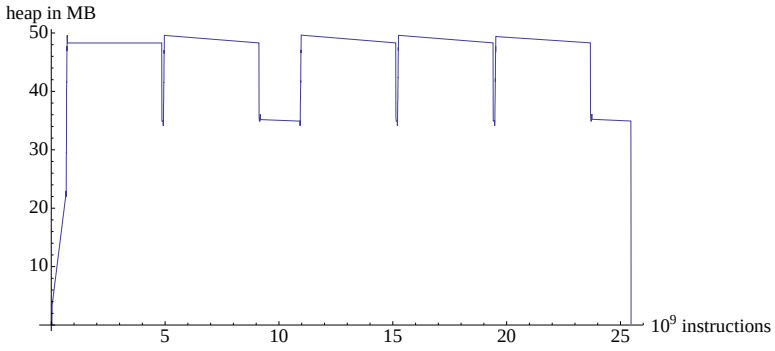

Abbildung 10: A1, 3 Zeitschritte. Spitzenverbrauch 49,7 MB.

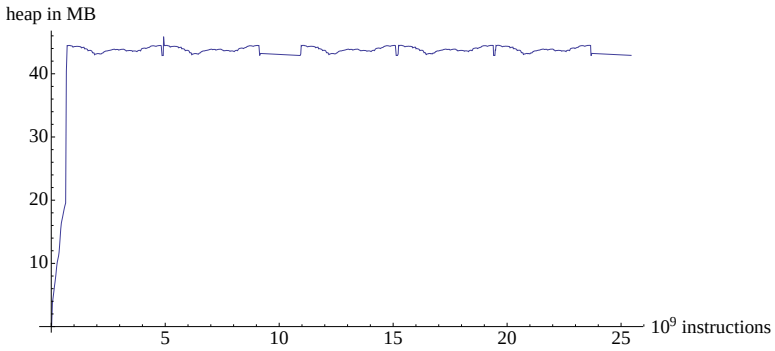

Abbildung 11: B1, 3 Zeitschritte. Spitzenverbrauch 45,9 MB.

Zunächst fällt auf, dass B wie gewünscht etwas weniger Speicher verbraucht als A.

Desweiteren fällt auf, dass der Speicherverbrauch von A phasenweise stark schwankt. Wenn man die Aufschlüsselung in Massif betrachtet, stellt man fest, dass die Methode `DomainDecomposition::exchangeMolecules` die Differenz verursacht. Dies findet bei 3 Zeitschritten 5 mal statt: beim Initialisieren, vor jedem Zeitschritt und zum Schluss.

Um Moleküle für den Halo-Bereich über MPI auszutauschen, werden die Moleküle bekanntlich in ein Array von `ParticleData`-Objekten kopiert, die nach dem Emfangen in Molekül-Objekte umgewandelt werden. Werden die Halo-Zellen nach der Paarbildung nicht mehr gebraucht, so werden sie geleert und der Speicherplatz wird frei.

Bei B tritt diese starke Schwankungen nicht auf. Der Grund dafür ist, dass die Vektoren in den Halo-Zellen zunächst ihre Kapazität beibehalten und so den Speicherplatz beim Leeren der Zelle nicht freigeben.

Die geringfügige Flukuation bei B lässt sich dadurch erklären, dass durch die inhomogene Moleküldichte das wandernde Fenster der Zellen keine konstante Anzahl an Molekülen beinhaltet.

5.4.2 Szenario 2

Für Szenario 2 erhalten wir jeweils Abb. 12 und 13. Durch die komplexeren Moleküle steigt auch der Speicherbedarf des Molekül-Caches. Das belastet A weitaus mehr als B und ist deutlich zu erkennen (vgl. Abb. 10 und 11).

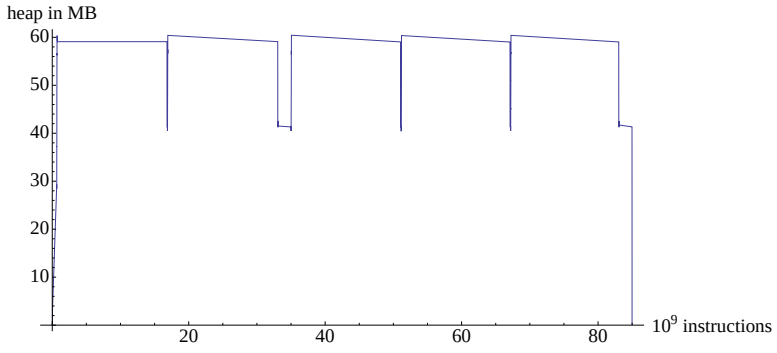

Abbildung 12: A2, 3 Zeitschritte. Spitzenverbrauch 60,4 MB.

Für Details betrachten wir die Massif-Snapshots (Tabelle 14) von A2 und B2 gleich zu Anfang der Simulation, bei dem die Halo-Zellen gerade befüllt werden und der Speicherverbrauch einen lokalen Höchststand erreicht hat.

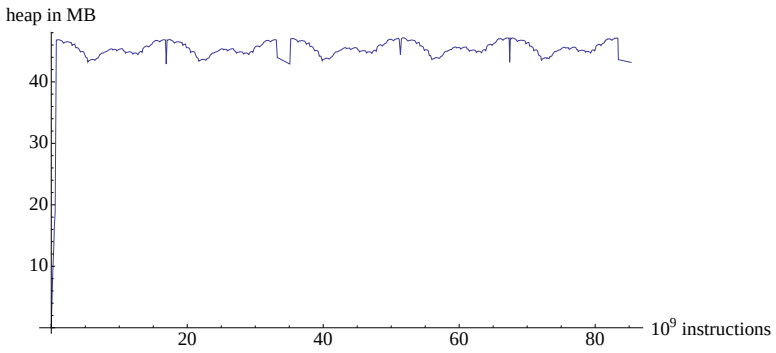

Abbildung 13: B2, 3 Zeitschritte. Spitzenverbrauch 47,2 MB.

	A2	B2
`Cell::addParticle`	27 507 768 B	28 135 836 B
→ `InputOldstyle::readPhaseSpace`	16 320 000 B	16 866 468 B
→ `DomainDecomposition::exchangeMolecules`	11 187 768 B	11 269 368 B
`ompi_free_list_grow`	14 203 884 B	14 203 884 B
`Molecule::Molecule`	12 944 832 B	-
`Molecule::reserveCache`	-	3 222 288 B
sonstige	4 076 175B	3 026 002 B

Abbildung 14: Massif-Snapshot von A und B beim befüllen der Halo-Zellen

Dabei weicht der Speicherverbrauch für den statischen Teil der Moleküle[17] bei beiden Versionen nur unwesentlich voneinander ab. Der Unterschied zwischen Netto- und Brutto-Speicherverbrauch bei B ist weitaus kleiner als erwartet.

Das lässt sich mit virtuellem Speicher erklären: für den Vektor werden Speicherseiten reserviert, die zunächst leer und rein virtuell sind und deshalb keinen Speicher belegen. Bietet eine Speicherseite beispielsweise Platz für 8 Moleküle und eine Zelle beinhaltet 70 Moleküle, so wird zwar Platz für 128 Moleküle und somit 16 Seiten reserviert, aber nur 9 Seiten existieren tatsächlich, die restlichen 7 sind nur virtuell.

Der große Unterschied findet sich beim dynamisch reservierten Teil des Moleküls. Während bei A der Molekül-Cache im Konstruktor[18] reserviert

[17] `Cell::addParticle`
[18] `Molecule::Molecule`

wird, übernimmt das bei B eine Methode[19], zu der es auch eine Methode zum Freigeben existiert[20]. Das Ergebnis ist, dass B für den Molekül-Cache nur etwa ein Viertel so viel Speicher belegt wie A.

Das Aktualisieren der Halo-Zellen[21] belegt deshalb so viel Speicherplatz, weil mit 27 421 aus 40 000 Molekülen ein Großteil im Zuge der DomainDekomposition über MPI versandt und in den Halo-Zellen dupliziert werden.

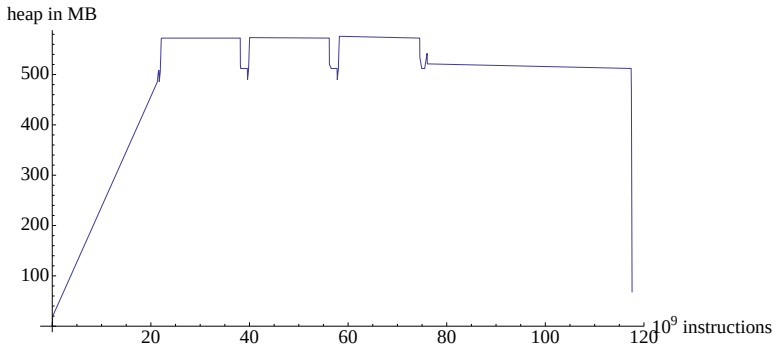

Abbildung 15: A3, 3 Zeitschritte. Spitzenverbrauch 576,0 MB.

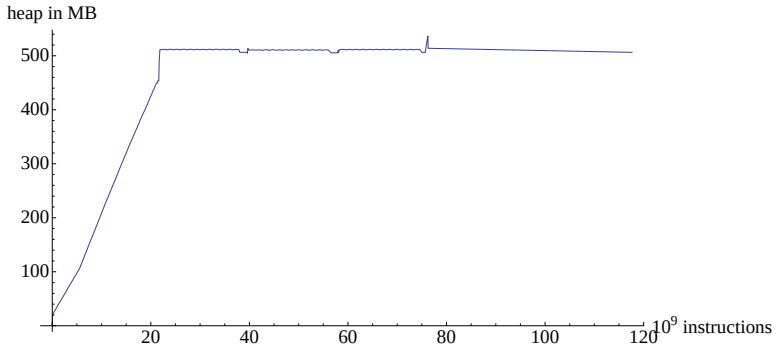

Abbildung 16: B3, 3 Zeitschritte. Spitzenverbrauch 536,9 MB.

[19] `Molecule::reserveCache`
[20] `Molecule::freeCache`
[21] `DomainDecomposition::exchangeMolecules`

5.4.3 Szenario 3

Beim umfangreicheren Szenario 3 (Abb. 15 und 16) ist der Speicherverbrauch selbstverständlich sowohl bei A als auch bei B größer. Die Einsparungen jedoch entsprechen relativ gesehen ungefähr dem erheblich kleineren Szenario 1. Interessant hierbei ist, dass der Speicherverbrauch von B über weite Strecken bei knapp über 510 MB bleibt und nur einmal gegen Schluss den Spitzenwert von 536,9 MB erreicht. Bei einer langen Simulation mit sehr vielen Zeitschritten wäre dieser Spitzenwert vernachlässigbar.

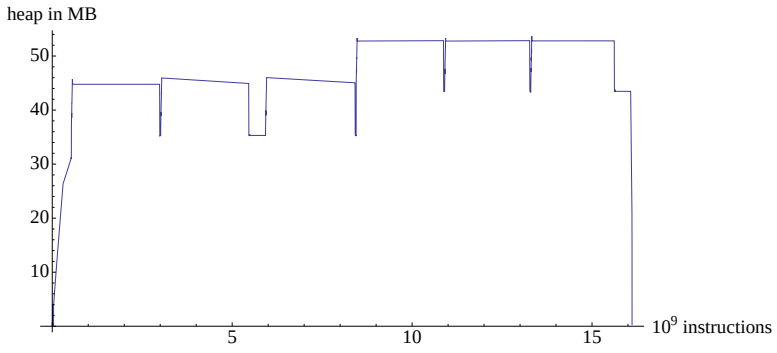

Abbildung 17: A4, 4 Zeitschritte. Spitzenverbrauch 53,6 MB.

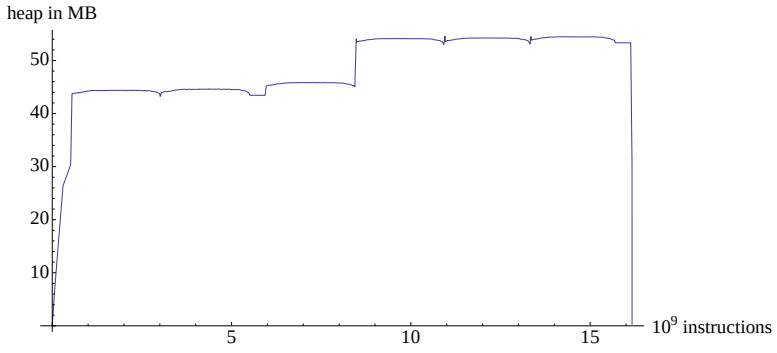

Abbildung 18: B4, 4 Zeitschritte. Spitzenverbrauch 54,6 MB.

5.4.4 Szenario 4

Szenario 4 (Abb. 17 und 18) hingegen ist sehr viel kleiner. Das gilt nicht nur für die Zahl von Molekülen, sondern auch für die Größe der Domain. Dadurch ist die Einsparung durch Cache-Reservieren nach Bedarf kleiner als der Mehrverbrauch durch Vektoren in den Zellen. Dementsprechend ist es auch keine Überraschung, dass B etwas mehr Speicher benötigt als A. Ein Sprung im Speicherbedarf lässt sich bei beiden beobachten. Der Grund dafür ist die explosionsartige Ausbreitung der zunächst konzentrierten Moleküle in die anfangs leeren Halo-Zellen.

6 Schluss

Die beiden vorgeschlagenen Optimierungen führen nicht nur theoretisch, sondern auch im praktischen Einsatz zu messbaren Verbesserungen sowohl in Hinsicht auf Laufzeit als auch auf Speicherverbrauch.

Zwar sind die besagten Verbesserungen in relativ kleinem Maße, könnten aber in extremen Szenarien signifikante Vorteile erzielen, denn je größer die Latenz zwischen den Schichten der Speicherhierarchie ist, umso deutlicher zeichnet sich der Vorteil von Datenlokalität und intelligentem Caching ab. Bei sehr großen Simulationen wäre es durchaus vorstellbar, dass die verwendete Speicherhierarchie bis zu Auslagerungsdateien auf der Festplatte reicht. Darüberhinaus kann die Datenlokalität als Grundlage für die Parallelisierung mit Hilfe von GPGPU dienen.

Weitere Verbesserungen sind durchaus möglich. So bietet das Molekül-Objekt noch Spielraum, weiter verschlankt zu werden. Auch feineres Tuning der neuen Vektor-Datenstruktur ist vorstellbar.

Andererseits können solche Optimierungen auch zu schlecht wartbarem Quellcode führen. Der Kompromiss zwischen Performanz und leichte Anpassbarkeit für neue Problemstellungen sollte daher berücksichtigt werden.

Literatur

[1] Dissertation: Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen, Martin Buchholz, 2010

[2] Numerische Simulation in der Molekldynamik, Michael Griebel, 2003