Programmieren einer Partikelsimulation für kurzreichweitige Interaktionen

Projektbericht

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Oliver Heidmann (1234567), Benjamin Warnke (6676867)
E-Mail-Adresse:	oliverheidmann@hotmail.de, 4bwarnke@informatik.uni-hamburg.de
Studiengang:	Bachelor Informatik
Betreuer:	Philipp Neumann

Hamburg, den 09.03.2017

Abstract

Ziel des dieses Projektes ist die Implementierung einer Partikel-Simulation für kurzreichweitige Partikel-Interaktionen mit Autotuning.

Contents

1 Einleitung

Bei Partikel Simulationen müssen normalerweise die Wechselwirkungen zwischen jedem möglichen Partikelpaar berechnet werden. Die Laufzeit des Programms steigt quadratisch zur Anzahl der Partikel. Dies ist besonders bei hohen Anzahlen von Partikeln kritisch für die Laufzeit. Die in diesem Projekt implementierte Partikel-Simulation ist für kurzreichweitige Interaktionen optimiert. Dadurch lassen sich die Interaktionen zwischen weit auseinanderliegenden Partikeln vernachlässigen, wodurch die Laufzeit kürzer werden kann. Es gibt verschiedene Möglichkeiten, die Interaktionen auf die kurzreichweitige Interaktion zu beschränken um das Programm zu beschleunigen. Das Problem bei diesen verschiedenen Möglichkeiten der Beschleunigung besteht darin, dass je nach Eingabe eine andere Art der Vereinfachung eine bessere Programm-Laufzeit ermöglicht. Die Besonderheit dieser Partikel Simulation liegt darin, dass das Programm zu beginn selbst entscheiden kann, welche Optimierungsstrategie für die gegebene Eingabe am sinnvollsten ist. Hieraus resultiert der Vorteil gegenüber anderen Programmen, dass die Laufzeit für jede beliebige Eingabe besonders schnell ist, und nicht nur wenn die Eingabe passend ist. Dies ist besonders dann spannend, wenn man selbst nicht sicher ist, welches Verfahren für die Eingabe am besten geeignet ist, ohne selbst vorher alle Möglichkeiten einmal auszuprobieren.

2 Realisierung

2.1 Design

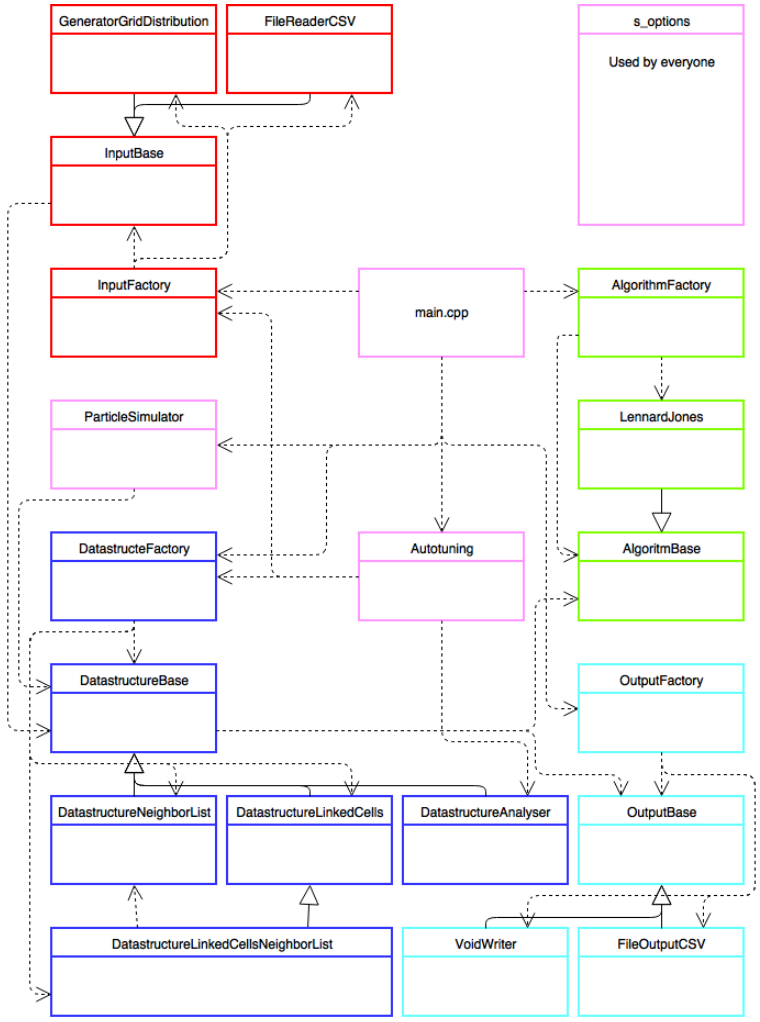

Figure 2.1: Klassendiagramm

Das Programm lässt sich grob in 5 Logische Teilbereiche gliedern.

1. **Input** (rot) Damit die Simulation starten kann, müssen Partikel vorhanden sein. Diese werden entweder generiert oder aus einer Datei geladen. Das laden aus einer Datei ist auch dann hilfreich, wenn aufgrund langer Programmlaufzeiten Checkpoints benutzt werden müssen, um später die Simulation fortzusetzen.

2. **Output** (cyan) Damit die Ergebnisse der Simulation später ausgewertet werden können, müssen diese als Datei vorhanden sein, damit andere Programme zur Visualisierung verwendet werden können. Wenn ausschließlich Analysen zur Laufzeit ausgeführt werden sollen, oder es darum geht Laufzeiten zu messen, dann kann der Output auch deaktiviert werden, um die reine Rechenzeit messen zu können.

3. **Datenstruktur** (blau) Es gibt verschiedene Möglichkeiten Partikelkombinationen auszuschließen, die nicht in einer Nachbarschaftsbeziehung sind. In diesem Projekt geht es besonders darum, je nach Eingabe eine andere Datenstruktur auszuwählen. Deshalb wurde bei dem Design darauf geachtet, das verschiedene Implementationen von Datenstrukturen leicht austauschbar sind. In der jeweils aktiven Datenstruktur sind alle Partikel gespeichert, die für die Simulation betrachtet werden sollen.

4. **Algorithmus** (grün) Es gibt verschiedene physikalische oder chemische zusammenhänge zwischen verschiedenen Partikeln oder Molekülen. Dieses Programm ermöglicht es relativ einfach weitere Arten der Interaktion zwischen Partikeln zu definieren, und diese dann zur Programmlaufzeit auszuwählen. Während dieses Projektes wurde ausschließlich das Lennard-Jones Potential zur Kräfteberechnung implementiert. Selbst wenn später andere Algorithmen zur Kraft Berechnung verwendet werden, sind die Datenstrukturen mit allen Optimierungen weiterhin verwendbar.

5. **Steuerung** (lila) Ein Teil des Programms ist für die Kontrolle der anderen Teilbereiche Zuständig. Zur den Kontrollstrukturen gehören in diesem Programm zum einen die Parameter, welche die Startbedingungen definieren, zum anderen gehört das Auto-tuning auch mit in diese Kategorie, da es beim Auto-tuning darum geht zu entscheiden, welche Voraussetzungen für das weitere Programm gelten sollen. Die Partikel Simulator Klasse steuert das verhalten der verschiedenen Iterationen. und führt Aktionen zwischen den Iterationen aus. Unter anderem können die Daten zwischen den Iterationen mithilfe der Output Komponente gespeichert werden.

2.2 Implementierung

Zu Beginn des Projektes haben wir uns eine grobe Struktur des Programms überlegt, und Schnittstellen definiert, um gleichzeitig in verschiedenen Komponenten an dem Projekt arbeiten zu können. Wir haben das Projekt in die folgenden Komponenten zerlegt:

- **Parameter** Schon zu beginn des Projektes war absehbar, dass das Programm mit verschiedenen Startparametern umgehen können muss. Zum einen ist dies sehr hilfreich, um zum Testen spezielles Verhalten zu provozieren, zum anderen ermöglicht ein parametrisierter Programmaufruf eine sehr flexible Einsatzmöglichkeit des Programms. Während des Programmierens wurden zunehmend mehr Parameter hinzugefügt. Sodass die Art wie die Parameter im Programm abgespeichert werden angepasst werden musste. Sobald die ersten Parameter übernommen

werden konnten, war das hinzufügen weiterer Parameter sehr einfach. Um es späteren Anwendern zu erleichtern wurde für jeden Programmparameter ein Hilfetext aufgeschrieben. Der Hilfetext für alle Parameter kann mithilfe des Parameters '--help' ausgegeben werden.

- **Abstrakte-Klassen** Da unser Projekt zum Ziel hat, dass das Programm automatisch entscheiden kann, welche Optimierung für die gegebenen Daten am sinnvollsten ist, muss es einen Weg geben verschiedene Implementationen schnell zur Laufzeit austauschen zu können. Um später keine Probleme zu bekommen, war es sinnvoll möglichst früh zu erkennen, welche Arten von Methoden definiert sein müssen, um eine gute Austauschbarkeit zu erreichen. Da nicht nur die Datenstruktur sondern generell jede Komponente des Programms flexibel sein solle, wurde für jede Komponente eine andere Abstrakte Klasse definiert. Daraus folgte für die Implementierung, dass die Implementationen der abstrakten Klassen wiederum nur die Basisklassen der anderen Komponenten kennen dürfen, um die volle Flexibilität zu gewährleisten.

- **Logging** Beim Programmieren ist es manchmal sinnvoll, lokale Variablen auf die Konsole zu schreiben, um den aktuellen Status des Programms zur Laufzeit nachverfolgen zu können. Auch zur Fehlersuche ist dies manchmal sehr hilfreich, da Debugger die Ausführungszeit teilweise so stark verlangsamen, dass einige Fehler dadurch nicht mehr auftreten. Deshalb haben wir uns schon zu beginn des Programmierens überlegt, wie wir hilfreiche Informationen ausgeben können, während das Programm getestet wird. Gleichzeitig sollte die Release-Version des Programms nicht unnötig durch Textausgaben verlangsamt werden. Wir haben dies durch Makros realisiert, die in der Release-Version dafür sorgen, dass nicht nur keine Ausgabe generiert wird, sondern zusätzlich nicht mal eine leere Funktion aufgerufen wird. Der Grund hierfür ist, dass selbst der Aufruf einer leeren Funktion etwas Zeit kostet. Durch große Anzahlen an Funktionsaufrufen, würde selbst diese Zeit zu merkbare Laufzeitunterschieden führen.

- **Startdaten** Da es verschiedene Möglichkeiten geben soll, wie die Partikel zu beginn der Simulation angeordnet sind, ist auch hier eine sehr gute Austauschbarkeit von verschiedenen Datenquellen erforderlich. Zum einen können die Partikel zur Laufzeit unter Berücksichtigung von Start Parametern generiert werden, zum anderen können die Partikel auch aus einer Datei geladen werden. Im Rahmen dieses Projekts werden die Partikel ausschließlich aus '*.csv' Datein geladen.

- **Algorithmus zur Interaktion** Wir haben uns in diesem Projekt darauf beschränkt, zur Berechnung der Kräfte zwischen Partikeln das Lennard-Jones-Potential zu verwenden. Da es theoretisch möglich sein soll, dieses Verfahren auch durch Parameter zu ändern, wurde auch hier darauf geachtet, dass der Code so organisiert ist, dass eine einfache Austauschbarkeit erreicht wird. Der Algorithmus wurde hierzu aufgespalten in zwei Teile. Der erste Teil beschränkt sich auf die Bewegung, die aus der aktuellen Geschwindigkeit des einzelnen Partikels resultiert. Der zweite Teil

beschränkt sich auf die Kräfte, die durch Wechselwirkungen zwischen Partikeln entstehen.

- **Ausgabe** Damit es sinnvoll ist Partikel zu simulieren, müssen die Daten in irgendeiner Form ausgegeben werden, um eine spätere Analyse zu ermöglichen. Hierfür wurde in diesem Projekt ein Modul implementiert, welches alle Partikel in einer '*.csv' Datei-Serie abspeichern kann. Wenn eine Ausgabe unerwünscht ist, z.B. um Laufzeitmessungen durchzuführen, dann kann diese auch deaktiviert werden.

- **Datenstrukturen** Zu beginn des Projektes wurden zwei verschiedene Optimierungsstrategien vorgeschlagen. In der Linked-Cells Variante wird ein Raster über das zu simulierende Volumen gelegt, und die Partikel werden in dieses Raster eingefügt. Interaktionen zwischen Partikeln können nur dann stattfinden, wenn sich 2 Partikel entweder in der gleichen Zelle befinden oder wenn die Zellen der Partikel benachbart sind. Bei einer großen Anzahl Zellen, sind nur noch relativ wenig Partikel in den einzelnen Zellen. Dadurch reduziert sich der benötigte Rechenaufwand enorm. Die andere Variante benutzt Nachbarschafts-Listen, um während der Iterationen nur die direkten Nachbarn zur Interaktion zu berücksichtigen. Der Nachteil bei dieser Variante besteht darin, dass das Aufbauen der Listen, welche Partikel in der Nachbarschaft sind eine hohe Laufzeit verursacht. Dafür ist allerdings sie benötigte Zeit pro Iteration niedriger als in der Linked-Cells Variante. Nachdem beide Varianten einzeln implementiert worden waren, war es relativ einfach möglich, eine kombinierte Variante zu erstellen, die innerhalb der Zellen Nachbarschafts Listen verwendet. Die kombinierte Variante kombiniert Vorteile und Nachteile von beiden Verfahren, und erzielt somit bei manchen Eingaben eine bessere Laufzeit, als die Verfahren jeweils einzeln betrachtet.

- **Parallelisierung** Zur Parallelisierung wurde in der Linked-Cells Variante OpenMP verwendet. Die Variante mit den Nachbarschafts Listen konnte nicht ohne weiteres Parallelisiert werden.

2.3 Analyse

2.3.1 Korrektheit

Um sicherzustellen, dass das Programm korrekte Ausgaben liefert, wurden verschiedene Arten von Test durchgeführt. Zum einen wurden einzelne Komponenten getestet, zum anderen wurde auch das Gesamtverhalten des Programms überprüft.

- **Unit-Tests** Um die Korrektheit von den einzelnen Komponenten des Programms zu gewährleisten wurden Unit-Tests eingesetzt. Schon beim schreiben der Unit-tests wurden viele Fehler gefunden und behoben. Bei späteren Refactoring-Maßnahmen wurden durch diese Testfälle viele neue Fehler verhindert. Zudem konnten diese Testfälle helfen, wenn neue Implementationen von Abstrakten Klassen eingefügt

wurden. Hier konnte man die neue Implementierung so lange verbessern, bis die schon existierenden Unit-Tests nicht mehr fehlschlagen.

- **Energieerhaltung** Aus dem Physikalischen Gesetz 'Aktion gleich Reaktion' ergibt sich, dass die Energie in einem geschlossenen System immer erhalten belieben muss. Eingebaute Funktionen im Programm ermöglichen die Ausgabe der aktuellen Energie im System, sodass leicht überprüft werden kann, ob die Energie in einem Akzeptablen Rahmen beleibt. Durch Rechenungenauigkeiten ist es sehr unwahrscheinlich, dass die Energie exakt gleich beleibt.

- **Visualisierung** Durch Visualisieren der Ausgabedaten wurde auch das fertige Ergebnis stichprobenartig überprüft. Bei den getesteten Parametern verhielt sich das Programm wie erwartet.

2.3.2 Parallelisierung

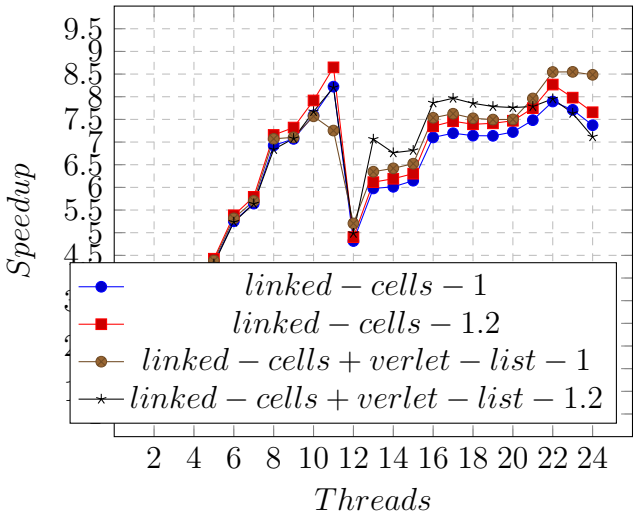

Figure 2.2: OpenMP Speedup Diagramm

2.4 Auto-tuning

Um herauszufinden, nach welchen Kriterien das Auto-tuning entscheiden kann, wurden viele Messungen mit verschiedenen Parametern durchgeführt. Natürlich ist klar, dass die Laufzeit des Programmes mindestens Linear von der Anzahl der zu simulierenden Partikel abhängt. Bei kurzreichweitigen Interaktionen kann man die Laufzeitabschätzung mit $p \cdot \frac{r^3}{V} \sim O(1)$ vereinfachen (siehe Tabelle 2.1).

	Basic	Nachbar-Listen	Linked-Cells	Kombination
Aufbau (Linked-Cells)	-	-	$\Theta(p)$	$\Theta(p)$
Aufbau (Nachbar-Listen)	-	$\Theta(p^2)$	-	$O\left(p^2 \cdot \frac{27 \cdot r^3}{V}\right)$ $\sim O(p \cdot 27)$
Iteration	$\Theta(p^2)$	$O\left(p^2 \cdot \frac{\frac{4}{3}\pi \cdot r^3}{V}\right)$ $\sim O\left(p \cdot \frac{4}{3}\pi\right)$	$O\left(p^2 \cdot \frac{27 \cdot r^3}{V}\right)$ $\sim O(p \cdot 27)$	$O\left(p^2 \cdot \frac{\frac{4}{3}\pi \cdot r^3}{V}\right)$ $\sim O\left(p \cdot \frac{4}{3}\pi\right)$

Table 2.1: Laufzeitvergleich der Datenstrukturen

Die Gesamtlaufzeit hängt von zwei Faktoren ab.

- **Wie oft** muss die Datenstruktur neu gebaut werden? Dies hängt ausschließlich von Eingabeparametern ab.
  - **cut-off Zusatzbereich** Je mehr Spielraum auf den cut-off Radius hinzugefügt wird, desto seltener müssen die Datenstrukturen neu aufgebaut werden. Dies geht allerdings auch sehr zulasten der Laufzeit, die in den Iterationen gebraucht wird.
  - **Startgeschwindigkeit** Je schneller sich die Partikel sich bewegen, desto schneller wird der dem cut-off hinzugefügte Bereich verlassen. Hieraus folgt, dass die Datenstruktur häufiger neu aufgebaut werden muss.
- **Wie lange** dauert es die Datenstruktur neu zu bauen? Wie viel schneller wird durch den Mehraufwand des Neubauens die einzelne Iteration? Je nachdem, welches Verfahren gewählt wird, ist die Laufzeit unterschiedlich. Bei der Variante bei der nur Nachbar-Listen verwendet werden ist der Aufwand diese Listen aufzubauen Quadratisch zur Anzahl der Partikel. Dies ist so ungünstig, dass es nicht empfehlenswert ist, diese Variante zu Benutzen. Die Linked-Cells-Variante benötigt wenig Zeit um die Datenstruktur aufzubauen, dafür aber wird pro Iteration viel Zeit benötigt. Die Kombinierte Variante benötigt mittelmäßig viel Zeit für den Aufbau der Datenstruktur kosten, und nur Minimale Zeit pro Iteration. Auch die Laufzeit für das Neubauen hängt von verschiedenen Parametern ab.
  - **cut-off Radius** Je größer der cut-off Radius gewählt wird, desto mehr Partikel befinden sich in der Nachbarschaft. Die Zellen werden hierdurch größer. Wenn

der cut-off Radius relativ groß ist, dann ist die Laufzeit pro Iteration in der nur Zellen basierten Version länger, in der Kombinierten Variante hingegen steigt die benötigte Laufzeit für den Neuaufbau der Datenstruktur. Allgemein gilt, je mehr Partikel in einer Zelle sind, desto günstiger wird die Verwendung der Kombinierten Variante, solange diese nicht allzu häufig neugebaut werden muss. Da dieses Programm sich auf kurzreichweitige Interaktionen fokussiert, werden keine sehr großen cut-off Radien auftreten.

- **Dichte** Wenn die Partikel dichter aneinander liegen, führt das dazu, dass sich Potentiell mehr Partikel innerhalb des cut-off Radius befinden.
- **Partikel Anzahl** Wenn die Anzahl der Partikel sehr gering ist, dann ist die Laufzeit des naiven Algorithmus ohne Optimierung kürzer, als eine der Optimierten Varianten, da der Naive Algorithmus nur sehr kleine Konstanten hat. Sobald mehrere hundert Partikel verwendet werden lohnt es sich optimierte Datenstrukturen zu verwenden, um unnötige Interaktionen schnell herausfiltern zu können. Wenn die absolute Anzahl der Partikel sehr groß ist, dann lohnt sich jedes einzelne Partikelpaar zwischen dem keine Interaktion berechnet werden muss. Die Listen basierenden Verfahren reduzieren die Laufzeit pro Iteration auf ein Minimum, haben aber einen dementsprechend größeren Zeitaufwand beim Neubauen der Datenstruktur.
- **Startanordnung** Je nachdem wie die Partikel bei Programmstart angeordnet sind ergeben sich andere Effekte. Zum Beispiel wäre es möglich, dass sich zu beginn alle Partikel in einem kleinem Bereich des zu simulierenden Raum aufhalten. Hieraus folgt, dass einige Zellen sehr viel mehr Partikel enthalten als andere. In der aktuellen Version des Programms wird für das Auto-tuning angenommen, das die Partikel einigermaßen gleichmäßig auf das Volumen verteilt sind.

2.4.1 Formel

Wenn die Geschwindigkeit des schnellsten Partikels bekannt ist, sowie die Strecke, die Maximal zurückgelegt werden darf, dann lässt sich die Zeit berechnen, bis die Datenstruktur Reorganisiert werden muss. Da die verschiedenen Datenstrukturen verschiedene Laufzeit Eigenschaften bezüglich Reorganisieren und Iterationen haben, ergibt sich die folgende Formel um zu entscheiden, welche Datenstruktur besser geeignet ist.

- $\mathbf{c} \rightarrow$ cut-off-radius
- $\mathbf{f} \rightarrow$ cut-off-radius-factor
- $\mathbf{s} \rightarrow$ start-speed

$$2 \cdot s < c \cdot (f-1) - 1$$
$$\cdot(f-1)$$
$$\frac{}{s}$$

- **true** → linked-cells+verlet-list
- **false** → linked-cells

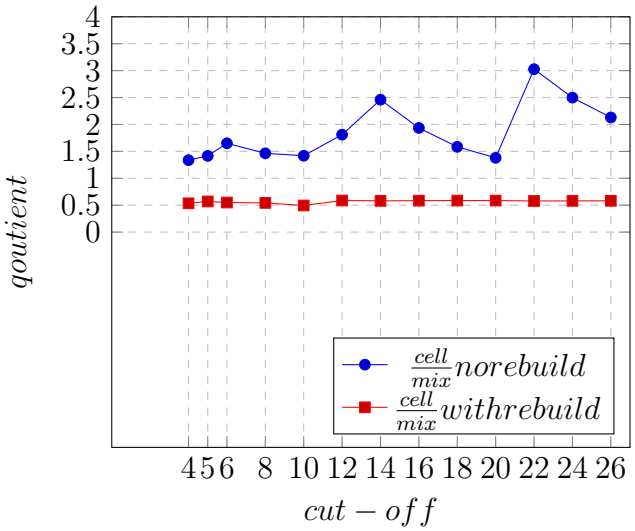

Figure 2.3: Laufzeit-Quotient der verschiedenen Verfahren

3 Zusammenfassung

4 Literatur

- M-Griebel, S. Knapek, G. Zumbuschm, A. Caglar: Numerische Simulation in der Moleküldynamic. Springer, 2003
- D.C Rapaport: The Art of Molecular Dynamics Simulation - 2nd edition, Cambridge University Press, 2004

5 Anhang

Verwendete Bibliotheken und Programme zum ausführen des Programms

- Boost
  - unit-tests
- CMake
- Make
- clang-format
- paraview
  - Visualisierung der Ausgabe
- lcov
  - Testabdeckung ermitteln und visualisieren
- slurm
  - Messtabellen berechnen
- doxygen
  - Dokumentation des Quelltextes
- latex
  - dieses Dokument
  - Präsentationen