Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bericht im Zuge des
Praktikum "Parallele Programmierung"

Direct Gravitational N-body Simulation

vorgelegt von

Nicholas Hickson-Brown, Michael Eidus

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

Abstract

Diese Arbeit beschäftigt sich mit der effektiven Parallelisierung eines N-body Codes zur Simulierung der Evolution von Kugelsternhaufen unter Einfluss von Gravitation, sogenannten "Direct Gravitational N-body Simulations". Im Kern dieser Arbeit steht die Frage, ob Algorithmen zur numerischen Integration mit Komplexitäten von $O(n^2)$ mit entsprechender Parallelisierung effektiver gestaltet werden können, ohne dabei an Genauigkeit zu verlieren. Dazu implementieren wir das dreidimensionale Plummer Dichteprofil zur Generierung initialer Konditionen eines Kugelsternhaufens und das Hermite Schema vierter Ordnung zur Lösung der numerischen Integralrechnung. Anschließend stellen wir ein Parallelisierungschema dar, mit dem die Arbeit auf mehrere Prozesse aufgeteilt werden kann, und implementieren dieses mittels des Message Passing Interface. Zum Abschluss analysieren wir Laufzeiten, Leistung und Skalierbarkeit unserer Anwendung.

Contents

1 Einführung

Unter dem Begriff N-body Simulation versteht man die Lösung numerischer Gleichungen, welche die Bewegung von N-Partikeln unter Einfluss der Gravitation oder anderer physikalischer Kräfte beschreiben. Zweck einer solchen Simulation ist es, die Interaktion der Partikel untereinander auf eine Weise zu beobachten, die in der Realität nicht möglich ist. N-body Simulationen können unser Verständnis des Universums erweitern, indem sie uns die Möglichkeit bieten, komplexe aber wohldefinierte Vorgänge mit Hilfe von Software darzustellen und zu analysieren, oftmals in einer vielfach höheren Geschwindigkeit als diese Vorgänge in der Wirklichkeit stattfinden.

N-body Simulationen finden Anwendungen in verschiedenen Bereichen der Astronomie und Astrophysik. Mit der Hilfe von solchen Computersimulationen lassen sich, relativ gesehen, einfache Vorgänge wie die Umlaufbahn unsere Mondes um die Erde simulieren, aber auch deutliche komplexere Abläufe, wie die Entstehung von Sternhaufen oder der Einfluss dunkler Materie. Des Weiteren finden sie nicht nur bei der Simulation großer Himmelskörper wie Planeten und Sternen Anwendung, sondern beispielsweise auch bei der Simulation einzelner Atome innerhalb einer Gaswolke.

Verschiedene Methoden existieren um solche Vorgänge zu simulieren, die allerdings alle eine Sache gemeinsam haben: Sie sind enorm rechenintensiv. Einige Methoden versuchen die Komplexität solcher Simulationen zu verringern, indem sie lediglich Schätzungen in bestimmten Fällen vornehmen und so weniger Ressourcen in Anspruch nehmen. Beispiele für solche Methoden sind der Barnes-Hut-Algorithmus, bei dem ein Sternhaufen in Quadranten unterteilt wird und diese Quadranten sich untereinander nur als Summe ihrer Masse betrachten, oder Particle-Mesh-Methoden.

Der Nachteil von Methoden, die Annahmen oder Annäherungen treffen, um Rechenzeit einzusparen, ist, dass sie zwar an Effizienz gewinnen aber an Genauigkeit verlieren. Dem Gegenüber stehen sogenannte "Direct Methods", welche die numerischen Newtonschen Bewegungsgleichungen direkt implementieren und damit zur Berechnung der Interaktionen innerhalb eines Systems jeden Partikel mit jedem anderen Partikel vergleichen müssen. Infolgedessen sind diese Methoden zwar sehr genau, besitzen aber eine wesentlich höhere Komplexität und erfordern demnach auch mehr Rechenzeit. Nennenswerte Vertreter der direkten Methoden sind beispielsweise die Runge-Kutta-Methode oder das Hermite-Schema.

2 Problemstellung

In dieser Arbeit wollen wir ergründen, ob es durch ausreichende Parallelisierung und Optimierung möglich ist, die Laufzeit von direkten Methoden dementsprechend zu verringern, dass diese eine effiziente und realistische Option gegenüber Tree-Codes (Barnes-Hut-Algorithmus) und Particle-Mesh-Methoden bieten. Dabei ist insbesondere der Faktor der Genauigkeit von direkten Methoden eine treibende Kraft hinter unserer Fragestellung. Sollte es möglich sein, mit entsprechender Parallelisierung einer direkten N-body Simulation deren Laufzeit zu verringern und Effizienz zu steigern, so wären diese gegenüber anderen Methoden zu präferieren.

Zur Beantwortung dieser Frage konzentrieren wir uns auf sogenannte "Direct Gravitational N-body Simulations". Diese Art von N-body Simulationen verwendet direkte Methoden zur Lösung der numerischen Bewegungsgleichungen und beachtet dabei nur den Einfluss der Gravitation um die Interaktionen und Bewegungen der Partikel zu berechnen. Infolgedessen betrachten wir hier keine äußeren Einflüsse, wie etwa dunkle Materie oder schwarze Löcher, und auch keine Kollisionen, entweder der Partikel unter einander oder mit eventuellen Komponenten des Sternhaufens, die keine Sterne sind. Solche Betrachtungen sind unserer Meinung nach den Direct Gravitational N-body Simulations untergeordnet und wir wollen versuchen eine Aussage für den allgemeinen Fall treffen, nicht den speziellen.

Zudem betrachten wir in unserer Arbeit einen Kugelsternhaufen von variabler Anzahl an Sternen, sprich Partikeln, ab einem Zeitpunkt t_0, an dem die Sterne des Systems bereits geboren sind und sich im Zentrum der Masse des Sternhaufens konzentrieren. Ab diesem Zeitpunkt starten wir die Simulation und betrachten die Evolution des Sternhaufens. Der Endzeitpunkt ist dabei undefiniert, beziehungsweise die Partikel besitzen keine biologische Uhr und sterben demnach nicht, sondern die Simulation endet bei Erreichen eines Zeitpunkts t_x, der vor Beginn des Ablaufs vom Benutzer definiert wird.

Außerdem wollen wir den Ablauf der Simulation visualisieren. Die Visualisierung fließt weder in die Betrachtung der Laufzeiten, noch in die Parallelisierung unseres Programms mit ein. Sie soll lediglich als Veranschaulichung der Ergebnisse dienen und die Funktionalität unserer Implementierung für den Verwender belegen.

Nach erfolgreicher Implementation einer sequentiellen N-body Simulation wollen wir eine effiziente Methode zur Parallelisierung des Programms finden und diese umsetzen. Die parallele Umsetzung soll dabei Antwort auf unsere eingehend genannte Frage geben.

3 Design

Im Folgenden wollen wir den Aufbau und die theoretischen Grundlagen unseres sequentiellen Programms darlegen. Dabei unterscheiden wir zwischen der Simulation an sich, sprich dem sequentiellen Programm, welches die numerischen Gleichungen löst, und der Visualisierung, welche die entstehenden Datensätze ansprechend darstellt und interaktiv gestaltet. Wir wollen mit diesem Kapitel eine Übersicht über die der Implementation zugrundeliegende Theorie und unsere Überlegungen geben.

3.1 Simulation

Bei der Implementation der Simulation hatten wir drei grundlegende Entscheidungen zu treffen: Die Methode der Generierung der initialen Konditionen unseres Sternhaufens, die Art der direkten Berechnung der Bewegungen der Partikel, sowie die Form der Datenausgabe. Im folgenden Abschnitt wollen wir die Theorie hinter unseren Entscheidungen näher betrachten und damit die Grundlage für das Verständnis der Implementation schaffen.

Um unsere Simulation zu starten benötigen wir initiale Koordinaten, Geschwindigkeiten und Massen für unsere Partikel, die zudem auch noch relativ realistisch sein sollten. Außerdem war uns klar, dass wir einen Kugelsternhaufen simulieren wollten, da in einem solchen durch die Dichte an Sternen und den zentralen Punkt der Masse eine höhere Anzahl an Interaktionen von Partikeln wahrscheinlich ist, was mehr Rechenaufwand und damit auch mehr Laufzeit bedeutet, die es durch Parallelisierung zu verringern gilt. Die Prämisse unserer Entscheidung für ein bestimmtes Modell zur Generierung der initialen Konditionen war demnach vor allem der zu erwartende Rechenaufwand.

Unser erster Ansatz war ein einfacher Linear-Congruential-Generator (LCG), da dieser eine relativ gute statistische Verteilung besitzt und aufwandsarm zu implementieren ist. Nach einigen ersten Tests fiel uns allerdings auf, dass der LCG zwar die erwartete Verteilung besaß, dieser aber weder Geschwindigkeiten noch Massen für unsere Sterne generieren konnte, da das Intervall des LCG durch die Größe des Datentyps *long* begrenzt ist und somit eine hohe Anzahl an wiederkehrenden Werten zu erwarten war. Außerdem wurde die Verteilung der Partikel keinen astrophysischen Kennzahlen gerecht. Nach einer eingehenden Suche einigten wir uns dann auf das sogenannte Plummer Model.

3.1.1 Plummer Model

Das Plummer Model oder "Plummer density profile" besitzt einen hohen Bekanntheitsgrad, da es einerseits eine sehr gute Verteilung bietet und anderseits mit relativer Leichtigkeit zu implementieren ist. Es wird häufig als sogenanntes "Toy Model" zur Generierung initialer Konditionen in N-body Simulationen eingesetzt. Das Plummer Model beschreibt die observierte Dichte von Sternenhaufen besser als andere Modelle seiner Art, hat aber auch einige Schwachstellen.

So ist bei großen Radien die Dichte nicht mehr so akkurat, wie in der Realität zu beobachten ist, und andererseits ist es keine gute Beschreibung von elliptischen Galaxien. Diese Schwachstellen stellen aber kein Hindernis für uns da, da wir weder elliptische Galaxien noch große Radien betrachten. Außerdem hat bereits Sverre Aarseth et. al. in "A comparison of numerical methods for the study of star cluster dynamics" [AHW74](Appendix) das Plummer Model als Grundlage für ähnliche Betrachtungen vorgeschlagen.

Das dreidimensionale Dichte-Profil von Plummer lässt sich durch folgende Formel beschreiben:

$$\rho P(r) = (\frac{3M}{4\pi a^3}) \cdot (1 + \frac{r^2}{a^2})^{-\frac{5}{2}}$$

wobei M die totale Masse des Sternhaufens darstellt, und a den Radius.

Das Plummer Model soll uns 7 Werte für jeden Partikeln erzeugen: Die Koordinaten (x-, y- und z-Achse), die Geschwindigkeiten (x-, y- und z-Achse) und die jeweilige Masse. Die Erzeugung der Masse ist simpel, da in unserem System alle Sterne die selbe Masse besitzen sollen (sog. "mass equilibrium").

Dies ist natürlich nicht sonderlich realistisch, aber zum Vergleich von Laufzeiten und Implementation fundamental, da es keine Varianz in den Ergebnissen und Energiewerten erzeugt. Mit einem Gleichgewicht der Masse können wir sicher sein, dass zwei unterschiedliche Systeme mit den selben Rahmenbedingungen, sprich Anzahl Partikel, Zeitschritt und Endzeit, zumindest sehr ähnliche Laufzeiten erzeugen, womit ein Vergleich von sequentieller und paralleler Implementierung erst möglich wird.

Zusätzlich benötigen wir natürlich auch noch die Koordinaten und Geschwindigkeiten für unsere Partikel, die durch das Plummer Modell gegeben sind. Dazu greifen wir auf das eingehend genannte Paper von Aarseth et. al. [AHW74] zurück, in dessen Appendix eine ausführliche Anleitung zur Bestimmung initialer Konditionen mit dem Plummer Modell steht. Die dort genannten Formeln und Techniken werden wir als Grundlage für die Implementation eines Generators für initiale Konditionen eines Kugelsternhaufens verwenden.

3.1.2 Mersenne Twister

Für die Berechnung der initialen Konditionen benötigen wir zufällig gewählte Werte innerhalb eines bestimmten Intervalls, die zudem gute statistische Eigenschaften besitzen müssen. Aus diesem Grund haben wir uns gegen die Implementation eines eigenen Random Number Generators (RNG) entschieden und stattdessen für einen bereits implementierten. Der Mersenne Twister bot sich dafür an, da dieser alle von uns gewünschten Eigenschaften besitzt und dessen Entwickler Makoto Matsumoto und Takuji Nishimura auf ihrer Website Implementationen des Mersenne Twister in verschiedenen Programmiersprachen mit freigiebiger Lizenz anbieten [MN07].

3.1.3 Hermite Integrator

Abschließend benötigen wir noch eine Methode, um die Newtonschen Bewegungsgesetze umzusetzen und damit die Interaktion und Bewegung der durch das Plummer Model erstellten Partikel unseres Kugelsternhaufens zu simulieren.

Begonnen haben wir damit, eine sehr simple Variante der direkten Methoden zu implementieren, indem wir die Newtonschen Bewegungsgleichungen direkt in stark vereinfachter Form implementiert haben. Diese Implementation war zwar einfach zu verstehen, jedoch nicht sonderlich elegant und die erzeugten Ergebnisse waren außerdem nicht sehr akkurat. Wir wollten eine Methode implementieren, die auch in der anderen N-body Simulationen eingesetzt wird, um so nah wie möglich an den echten Use-Case der N-body Simulationen zu gelangen.

Über den Forward-Euler-Algorithmus, den Leapfrog-Algorithmus und die Runge-Kutta-Methode sind wir dann auf den Hermite Integrator oder das Hermite Schema gestoßen. Das Hermite Schema wird häufig in N-body Simulationen, die direkte Methoden verwenden, eingesetzt und war damit der perfekte Kandidat für unsere Leitfrage. Zusätzlich ist der Hermite Algorithmus der am einfachsten zu implementierende Algorithmus vierter Ordnung.

Der Hermite Algorithmus leitet die Newtonschen Gesetze der Bewegung nur einmal ab und gelangt so direkt an die entsprechende Beschleunigung und den sogenannten "Jerk", eine Art ruckartiger Bewegung, für jeden einzelnen Partikel. Die resultierenden Gleichungen können dann durch eine Taylor Reihe erweitert werden. Der Ablauf ist wie folgt darstellbar:

- Zuerst berechnen wir eine Vorhersage für die nächsten Positionen und Geschwindigkeiten unser Partikel:

$$r_{p,i+1} = r_i + v_i\Delta t + \frac{1}{2}a_i(\Delta t)^2 + \frac{1}{6}j_i(\Delta t)^3$$

$$v_{p,i+1} = v_i + a_i\Delta t + \frac{1}{2}j_i(\Delta t)^2$$

- Danach berechnen wir eine Vorhersage für die jeweiligen Beschleunigungen und Jerks.

- Anschließend verwenden wir die berechneten Beschleunigungen und Jerks um die Vorhersagen aus dem ersten Schritt zu korrigieren:

$$v_{c,i+1} = v_i + \frac{1}{2}(a_i + a_{p,i+1})\Delta t + \frac{1}{12}(j_i - j_{p,i+1})(\Delta)^2$$

$$r_{c,i+1} = r_i + \frac{1}{2}(v_i + v_{c,i+1})\Delta t + \frac{1}{12}(a_i - a_{p,i+1})(\Delta)^2$$

Die entsprechenden Gleichungen und Beweise für diese haben wir dem Buch "The Art of Computational Science: The Kali Code" [HM07](Kapitel 11) entnommen, welche als Grundlage für unsere Implementation des Hermite Algorithmus dienen sollen.

3.2 Visualisierung

Damit die erzeugten Daten auch greifbar werden und die Interaktion der Partikel beobachtet werden kann haben wir uns entschieden eine Visualisierung der Daten mit OpenGL zu realisieren.

Die Visualisierung der erzeugten Daten erfolgt mit OpenGL 3.3. OpenGL ist ein von der Khronos Group entwickelter offener Standard für die plattform- und programmiersprachenübergreifende 2D- und 3D-Grafik-Entwicklung. OpenGL stellt nur eine auf das Rendern begrenzte Funktionalität zur Verfügung und erlaubt somit nur eine direkte Zeichnung von wenigen primitiven geometrischen Figuren wie z. B. Punkten, Linien, Dreiecken, Vierecken und Polygonen. Komplexere geometrische Figuren wie z. B. Kreise oder Sphären müssen beim Programmieren aus den genannten Figuren zusammengestellt werden.

GLFW ist eine in C geschriebene Bibliothek die bestimmte plattformspezifische Aufgaben übernimmt. Dazu gehören die Erzeugung und Steuerung eines Fensters zur Darstellung der OpenGL Grafik, die Verarbeitung der Inputs wie Tastatur, Maus und Ähnliches sowie Zwischenspeicher- und Zeitfunktionen.

GLAD setzt treiberspezifische Pointer zu OpenGL-Funktionen und ermöglicht somit deren Ansteuerung.

OpenGL Mathematics (GLM) ist eine in C++ geschriebene, auf die Grafik-Entwicklung spezialisierte Bibliothek für mathematische Berechnungen. Sie ist nach den gleichen Namenskonventionen aufgebaut wie die OpenGL Shading Language (GLSL) (siehe Abschnitt 4.2.3) und erleichtert damit den Einstieg für Programmierer die bereits mit GLSL vertraut sind. Sie bietet für 3D-Berechnungen nützliche mathematische Funktionen wie z. B. Vektorberechnungen, Matrizentransformationen und viele weitere, die in diesem Projekt nicht weiter verwendet wurden.

Da OpenGL nur grundlegende Funktionalitäten zur Darstellung bietet gestaltet sich die Darstellung von Text als schwierig. Eine Möglichkeit wäre die einzelnen Buchstaben aus den oben genannten primitiven Formen zusammenzustellen was offensichtlich mit einem hohen Aufwand verbunden wäre. Eine andere Möglichkeit macht sich Texturen zunutze welche über die primitiven Formen gelegt werden.

Diesen Ansatz verfolgt auch die FreeType Bibliothek. Dabei werden skalierbare TrueType-Fonts wie in unserem Projekt die ”calibri.ttf” aus dem Ordner ”fonts” geladen und in Texturen umgewandelt. So kann jeder Buchstabe einzeln als Textur entnommen und über eine geometrische Figur, meist ein Viereck gelegt werden. Aus der Kombination solcher Vierecke kann schließlich ein Text zusammengesetzt werden.

4 Implementation

Dieses Kapitel soll dazu dienen, detaillierter auf die eigentliche Implementierung einzugehen und anhand von Quelltext-Ausschnitten zu zeigen, wir die von uns im vorangegangenen Kapitel benannten Modelle und Methoden konkret umgesetzt wurden. Hierbei unterscheiden wir wieder zwischen der Implementierung der Simulation und Visualisierung.

4.1 Simulation

Die Simulation wurde komplett in C geschrieben und modularisiert, sprich die einzelnen Schritte der Simulation wurden in eigene Dateien verpackt. Zur Speicherung der einzelnen Daten der Partikel (Masse, Position und Geschwindigkeit) verwenden wir eindimensionale Arrays, da diese nicht nur (theoretisch) einen Laufzeitvorteil gegenüber zweidimensionalen Arrays bieten, sondern auch wesentlich kompatibler mit MPI sind. Dabei werden die Daten zu den Massen der Partikel als *double* abgespeichert, während Daten zu den Positionen und Geschwindigkeiten als *double complex* abgespeichert werden, da diese ansonsten nicht mehr durch andere Datentypen darstellbar wären.

Die Arrays werden in der Datei "driver.c" erzeugt und bekommen dort auch der Anzahl an Partikeln und Dimensionen entsprechenden Speicherplatz zugewiesen. Diese Datei bietet die main-Funktion und verwaltet den Ablauf der Simulation. Hier werden User Input verarbeitet und die einzelnen Operationen der Simulationen in sinnvoller Reihenfolge aufgerufen. Als Input für unsere Simulation akzeptieren wir maximal 4 Argumente: einen Seed für den Mersenne-Twister, die Anzahl der Partikel, einen Zeitschritt und die Endzeit, wobei das Argument Seed optional ist und wenn nicht vorhanden, die momentane Unix-Clock als Seed verwendet wird.

Alle Daten, mit Ausnahme der Log-Datei, werden im *.csv-Format abgespeichert. Die Log-Datei wird als *.txt-Datei erstellt.

4.1.1 Erzeugung initialer Konditionen

Nachdem der Input verarbeitet und Speicherplatz zugewiesen wurde, startet der Ablauf des Plummer Models, wo die initialen Koordinaten erzeugt werden. Dazu werden die Pointer zu den Partikeldaten als Parameter an die Plummer-Operationen übergeben, wo diese in einer Schleife mit Daten befüllt werden, bis die angegebene Anzahl an Partikeln erreicht ist.

```
1  void plummer(int N, double *mass, double complex *pos,
     ↪ double complex *vel, int i, int mi, double M, double R)
2  {
3      /* mass equilibrium */
4      mass[mi] = M / N;
5
6      /* inverted cumulative mass distribution */
7      double complex radius = R /
         ↪ csqrt((cpow(genrand_real1(), (-2.0/3.0))) - 1.0);
8       /* Polar Angle */
9      double complex theta = cacos(rrand(-1.0, 1.0));
10      /* Azimuthal Angle */
11     double complex phi = rrand(0.0, (2 * 3.14159265359));
12
13     /* conversion from radial to cartesian coordinates */
14     pos[i] = (radius * csin(theta) * ccos(phi)) / scale;
15     pos[i + 1] = (radius * csin(theta) * csin(phi)) / scale;
16     pos[i + 2] = (radius * ccos(theta)) / scale;
17
18     double x = 0.0;
19     double y = 0.1;
20
21     /* Neumann's rejection technique */
22     while(y > (x * x * (pow((1.0 - x * x), 3.5))))
23     {
24         x = rrand(0.0, 1.0);
25         y = rrand(0.0, 0.1);
26     }
27
28     /* distribution function */
29     double complex velocity = x * csqrt(2.0) * cpow((1.0 +
         ↪ radius * radius), -0.25);
30     theta = cacos(rrand(-1.0, 1.0));
31     phi = rrand(0.0, (2 * 3.14159265359));
32
33     /* conversion */
34     vel[i] = (velocity * csin(theta) * ccos(phi)) *
         ↪ csqrt(scale);
35     vel[i + 1] = (velocity * csin(theta) * csin(phi)) *
         ↪ csqrt(scale);
36     vel[i + 2] = (velocity * ccos(theta)) * csqrt(scale);
37 }
```

4.1.2 Integration

Wenn die initialen Konditionen erstellt worden sind, werden Operationen zum Drucken der initialen Konditionen und der Log-Datei aufgerufen. Danach werden die gerade erzeugten Daten weitergereicht an den Hermite Integrator, wo neben der Berechnung der nächsten Positionen und Geschwindigkeiten auch die Beschleunigung und die Jerks berechnet werden. Zusätzlich werden auch noch Aufrufe an die Energie-Diagnostik gemacht und nach jeder Iteration die momentanen Daten gedruckt.

```
1  void hermite(int N, int DIM, double dt, double *mass,
       ↪ double complex *pos,
2  double complex *vel, double complex *acc, double complex
       ↪ *jerk)
3  {
4      /* storing positions, velocities, acceleration and jerk
           ↪ from last iteration */
5      double complex old_pos[N * DIM];
6      double complex old_vel[N * DIM];
7      double complex old_acc[N * DIM];
8      double complex old_jerk[N * DIM];
9
10     memcpy(old_pos, pos, sizeof(old_pos));
11     memcpy(old_vel, vel, sizeof(old_vel));
12     memcpy(old_acc, acc, sizeof(old_acc));
13     memcpy(old_jerk, jerk, sizeof(old_jerk));
14
15     /* prediction for all particles */
16     for(int i = 0; i < (N * DIM); ++i)
17     {
18     pos[i] += vel[i] * dt + acc[i] * ((dt * dt)/2) +
           ↪ jerk[i] * ((dt * dt * dt)/6);
19     vel[i] += acc[i] * dt + jerk[i] * ((dt * dt)/2);
20     }
21
22     acc_jerk(N, DIM, mass, pos, vel, acc, jerk);
23
24     /* correction in reversed order of computation */
25     {
26     vel[i] = old_vel[i] + (old_acc[i] + acc[i]) * (dt/2) +
           ↪ (old_jerk[i] - jerk[i]) * ((dt * dt)/12);
27     pos[i] = old_pos[i] + (old_vel[i] + vel[i]) * (dt/2) +
           ↪ (old_acc[i] - acc[i]) * ((dt * dt)/12);
28     }
29 }
```

4.2 Visualisierung

4.2.1 Vorbereitende Schritte

Zuerst werden die in Abschnitt 3.2 genannten Bibliotheken und einige Standardbibliotheken geladen. Danach werden diverse Variablen deklariert und initialisiert und damit auch grundlegende Einstellungen der Visualisierung festgelegt. Dabei wird der Standardordner für die Daten auf "run" gesetzt, die Anzahl der Partikel aus der "initial_conditions.csv" ausgelesen und die Anzahl der Iterationen gezählt.

4.2.2 Initialisierung

Zunächst wird GLFW initialisiert und festgelegt, dass OpenGL 3 und Multisampling (Antialiasing) benutzt wird. Danach wird ein 800 x 600 Pixel großes Fenster mit der Bezeichnung NBody_Visualization_2.0 und variabler Fenstergröße erzeugt und GLAD ausgeführt.

4.2.3 Kompilieren der Shader

Die eigentliche Grafikberechnung auf Grundlage der von der CPU vorbereiteten Daten erledigen Shader. Diese sind eigene Programme und werden bei OpenGL in der C-ähnlichen Programmiersprache OpenGL Shading Language (GLSL) geschrieben. Sie werden zur Laufzeit kompiliert und in den spezialisierten Shadereinheiten der Grafikkarte ausgeführt. Es gibt unterschiedliche Shader-Arten, denen unterschiedliche Aufgaben zugeordnet werden. In unserem Fall nutzen wir den für die 3D-Transformation einer Szene zuständigen Vertexshader sowie den für die Farbberechnung von Pixeln zuständigen Fragmentshader. Für die Darstellung der Partikel, des Textes und der Kreise haben wir jeweils eigene, an die Aufgabe angepasste, Shader-Programme im Ordner „shaders“.

4.2.4 Render-Loop

Nachdem mit FreeType die Schrift "calibri.ttf" aus dem Ordner "fonts" geladen und die Energiedaten für jede Iteration aus der "energy_diagnostics.csv" ausgelesen und in einem Array gespeichert wurden, wird der Render-Loop gestartet. Dieser wird durchlaufen bis die Visualisierung beendet wurde. Ein Durchlauf des Render-Loops führt zur Darstellung eines Bildes, dabei wird genau eine Iteration an Daten ausgelesen, verarbeitet und dargestellt. Zur Ausgabe der frames per second (fps) wird die Zeitdifferenz zwischen Beginn und Ende des Render-Loops festgehalten und daraus die fps berechnet. Außerdem werden hier die Benutzereingaben verarbeitet.

Danach wird der Hintergrund schwarz gestellt und das letzte Bild gelöscht. Im Fall des „Experimental painting mode“ wird das Löschen übersprungen und somit das neue Bild auf dem alten Bild gezeichnet.

Wenn die Datendarstellung eingeschaltet ist, werden mit Hilfe von FreeType folgende Daten in der genannten Reihenfolge von oben nach unten dargestellt:

- Frames per second (fps)
- Anzahl der markierten Partikel
- Gesamtanzahl der Iterationen
- Gesamtanzahl der Partikel
- Anzahl der Partikel mit einem bestimmten Abstand zum Mittelpunkt des Koordinatensystems
- Anzahl der Partikel außerhalb eines bestimmten Abstands zum Mittelpunkt des Koordinatensystems
- Energiewerte der aktuellen Iteration
- Aktuelle Iteration

Außerdem werden der eingeschaltete Slowmotion-Modus und das Erreichen der letzten Iteration signalisiert.

Weiterhin werden ein roter und ein grüner Kreis gezeichnet. Der grüne Kreis ist ein Orientierungsmaß für die Anzahl an Partikeln innerhalb einer Sphäre mit variablem Radius, während der rote Kreis ein Orientierungsmaß für die Anzahl an Partikeln außerhalb einer Sphäre mit variablem Radius ist. Da OpenGL keine direkte Implementierung für die Darstellung der Kreise liefert, mussten die Kreise aus vielen verbundenen Linien zusammengestellt werden. Die ausgelesenen Positionsdaten der jeweiligen Iteration aus der ”iteration_X.csv” werden in einem Array gespeichert und in den Grafikkartenspeicher geladen. Hier sollte festgehalten werden, dass so viele Daten wie möglich am Stück in den Grafikkartenspeicher übertragen werden sollten. Das mehrfache Senden kleiner Datenpakete nimmt vergleichsweise mehr Zeit in Anspruch und bremst damit die Darstellung unnötig aus was wir in einer frühen Version der Visualisierung deutlich gespürt haben.

Im Grafikkartenspeicher werden die Positionsdaten zuerst mit dem Vertexshader verarbeitet. Dieser erhält außerdem diverse weitere Daten wie z. B. die Punktgröße zur Darstellung der Partikel, die Anzahl der markierten Partikel und die Gesamtanzahl der Partikel zur weiteren Verarbeitung.

Im Vertexshader werden vereinfacht gesagt folgende Berechnungen durchgeführt:

- Für jedes Partikel wird entweder eine realistische oder bunte Farbe festgelegt
- Die Größe einzelner Partikel wird verändert um das Blinken der Sterne zu simulieren
- 70% der Partikel werden klein, 20% mittel und 10% groß dargestellt

- Markierte Partikel werden sehr groß und in roter Farbe dargestellt

Abhängig von der Position des Betrachters werden die Daten mit einer außerhalb des Shaders berechneten Matrix und einer Projektions-Matrix multipliziert um eine realitätsnahe 3D-Darstellung der Szene zu erreichen. Danach werden die Resultate des Vertexshaders an den Fragmentshader weitergeleitet, wo Farbveränderungen für das Blinken berechnet und die endgültige Farbe der Pixel festgelegt werden. Im Anschluss werden die so berechneten Partikel im Fenster gezeichnet und der Durchlauf des Render-Loops beginnt von vorne.

4.2.5 Features

Im Folgenden werden die Features der Visualisierung stichpunktartig beschrieben.
Hotkeys:

- Die Größe des Darstellungsfensters ist variabel, außerdem lässt sich mit Alt+Enter der Vollbildmodus aktivieren und deaktivieren
- Mit den Pfeiltasten nach oben und unten lässt sich zum Mittelpunkt der Welt rein- bzw. rauszoomen. Die Zoomgeschwindigkeit nimmt mit dem Abstand zum Mittelpunkt zu
- Mit den Pfeiltasten nach links und rechts dreht sich der Betrachter um den Mittelpunkt der Welt
- Mit „M“ lässt sich der Mouselook aktivieren/deaktivieren, Bewegungen werden dabei mit den Pfeiltasten ausgeführt, auch hier nimmt die Geschwindigkeit mit dem Abstand zum Mittelpunkt zu
- „R“ führt zu einem Reset, die Daten werden wieder ab der 1. Iteration ausgelesen
- „1“, „2“, „3“ verändern die Größe aller Partikel um die Sichtbarkeit zu verbessern
- „Leertaste“ startet und stoppt das Auslesen der Daten
- „S“ aktiviert/deaktiviert den Slowmotion Modus, die Partikeldaten werden mit 25% Geschwindigkeit ausgelesen, die Bewegungsgeschwindigkeit beim Mouselook wird reduziert, damit einzelne Partikel besser beobachtet werden können
- „PageUp“ und „PageDown“ markiert bzw. löscht die Markierung eines Partikels, hiermit kann das Verhalten einzelner Partikel besser beobachtet werden
- „D“ zeigt bzw. verbirgt die o. g. Daten
- „F“ zeigt bzw. verbirgt die o. g. Orientierungsmaße
- „NUM_1“ , „NUM_4“ , „NUM_2“ , „NUM_5“ Verändern die o. g. Orientierungsmaße, die Veränderungssprünge werden nahe des Mittelpunkts feiner

- „C“ wechselt zwischen der farblich realitätsnahen und einer bunten Darstellung der Partikel
- „P“ aktiviert/deaktiviert den o. g. „Experimental painting mode“, dieser ist als Spass Modus gedacht, es lassen sich jedoch auch die Orbitale der Partikel beobachten
- „Escape“ beendet die Visualisierung

Weitere Features:

- Sollten die Iterationsdaten in einem anderen als dem „run“ Ordner liegen, lässt sich dieser mit N Body Visualization 2.0.exe <Ordnername> nutzen
- Fehlermeldungen werden in den folgenden Fällen ausgegeben:
  - GLFW kann kein Fenster erzeugen
  - GLAD lässt sich nicht initialisieren
  - FreeType lässt sich nicht initialisieren
  - FreeType kann die Schrift „calibri.ttf“ aus dem Ordner „fonts“ nicht laden
  - FreeType kann ein bestimmtes Zeichen nicht laden
  - „initial_conditions.csv“ oder „iteration_X.csv“ kann nicht geöffnet werden
  - Fehler in einer Bestimmten Zeile von „initial_conditions.csv“, „iteration_X.csv“ oder „energy_diagnostics.csv“
  - Mehr als 1 Argument übergeben

4.2.6 Zusammenfassung

Zusammenfassend lässt sich festhalten, dass die Visualisierung der Daten mit OpenGL ohne Nutzung einer fertigen Engine die richtige Entscheidung war. Zwar sind der Lernaufwand und die Fehleranfälligkeit hoch, jedoch behält man im Endeffekt die volle Kontrolle über die Darstellung und lernt die Hintergründe sowie Grundlagen der 3D-Entwicklung. Außerdem war die Visualisierung nützlich um die Korrektheit der berechneten Daten nachvollziehen und Schlussfolgerungen aus dem Verhalten der Partikel ziehen zu können. Da für unser Ziel der Darstellung eines Sternencluster keine aufwändigen 3D-Effekte oder Ähnliches notwendig sind war mit OpenGL ein relativ schlanker Code mit niedrigen Hardwareanforderungen entstanden und ein großes Interesse an der 3D-Entwicklung geweckt worden.

5 Parallelisierungsschema

Durch Spurdatenanalysen haben wir herausgefunden, dass unser sequentielles Programm die meiste Zeit damit verbringt, die Beschleunigungen und Jerks zu berechnen, da hier alle Partikel gegen alle anderen Partikel verglichen werden müssen. Unsere Anwendung verbringt im Schnitt 83% der Laufzeit mit der Berechnung von Beschleunigungen und Jerks. Weitere 11% der Laufzeit werden mit der Berechnung der potentiellen Energie und 5% mit dem Drucken der Daten in Dateien nach jeder Iteration verbracht. Insgesamt stellen diese drei Operationen also 99% der Laufzeit unserer sequentiellen Implementierung. Dementsprechend haben wir uns zu erst auf die Parallelisierung dieser konzentriert.

Aus unserer Sicht macht es am meisten Sinn, zu aller erst die Berechnung der Beschleunigung und Jerks zu parallelisieren. Unser Parallelisierungsschema geht dabei davon aus, dass der Root-Prozess die meiste Arbeit erledigt. Zwar bekommen alle Prozesse die Pointer für die Partikeldaten und zugehörigen Speicherplatz zur Verfügung gestellt und erzeugen auch initiale Daten, doch die Print-Operationen werden alleine vom Root-Prozess übernommen, damit es hier zu keinen Konflikten kommt.

Sobald der Ablauf bei der Berechnung neuer Geschwindigkeiten und Jerks angelangt, wird die Arbeit verteilt: Jeder Prozess erhält einen Teil der globalen Arrays für Positionen, Geschwindigkeiten, Beschleunigungen und Jerks. Anschließend schickt Root die aktuellen Positionen und Geschwindigkeiten an alle Prozesse, da diese immer nur mit Teildaten arbeiten und ansonsten falsche Werte entstehen würden. Die einzelnen Prozesse berechnen dann für ihre Teile der Partikeldaten neue Beschleunigungen und Jerks, indem sie ihre Partikel gegen die zuvor von Root verschickten globalen Positionen und Geschwindigkeiten vergleichen. Zum Abschluss sammelt der Root-Prozess alle neuen Daten in die globalen Arrays zurück und der Ablauf verfährt normal weiter.

Die Berechnung der potentiellen Energie des Clusters war unser zweiter Punkt, allerdings mussten wir feststellen das hier eine Aufteilung der Arbeit wenig Sinn macht, da der Overhead so groß ist, dass gerade bei kleinen Mengen von Partikeln keine Verbesserung der Laufzeit möglich ist, im Gegenteil wird die Laufzeit teilweise sogar verlängert.

Ebenso war die Parallelisierung der Datenausgabe nach jeder Iteration nicht sonderlich erfolgreich, da die Laufzeit dieser Operation immer proportional zur Anzahl der Iterationen steht und eine einzelne Datenausgabe im Schnitt lediglich 5 Millisekunden in Anspruch nimmt. Jeglicher Versuch der Parallelisierung hat hier ebenso wie bei der Berechnung der potentiellen Energie zu längeren Laufzeiten geführt und wurde deswegen vernachlässigt.

6 Laufzeitmessungen

Unsere Anwendung nimmt vom Benutzer mindestens drei Command-Line Argumente entgegen, die alle unterschiedliche Auswirkungen auf die Laufzeit der Anwendung haben können. In den folgenden Abschnitten wollen wir dieses Auswirkungen quantifizieren und die Messungen der sequentiellen und parallelen Laufzeiten gegenüberstellen um ein besseren Eindruck davon zu erhalten, inwieweit die Parallelisierung unserer Anwendung diese beschleunigt.

6.1 Sequentielle Laufzeitmessungen

Zur Messung der Laufzeiten der sequentiellen Anwendung haben wir alle unsere Durchläufe auf der Partition *west* laufen lassen. Begonnen haben wir mit der Messung der Auswirkung der Anzahl an Partikeln auf die Laufzeit. Der Zeitschritt war mit 0.01 für alle Durchläufe gleich, ebenso wie die Endzeit mit 1. Beim Kompilieren der Anwendung wurden keine Optimisierungsflaggen gesetzt.

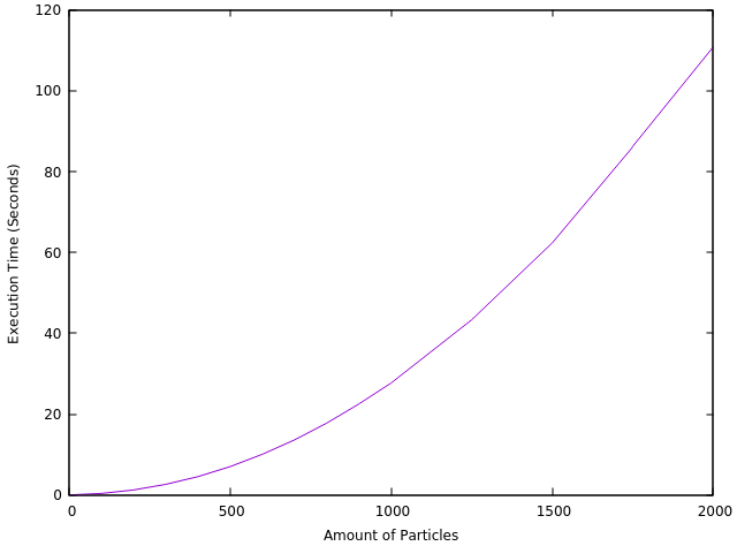

Figure 6.1: Laufzeiten bei steigender Anzahl an Partikeln (sequentiell)

Wir können deutlich den Einfluss der Komplexität $O(n^2)$ auf unsere Laufzeiten sehen: Die Kurve wächst bei steigender Anzahl der Partikel exponential an.

Nach einigen ersten Messungen und Vergleichen von Durchläufen mit gleicher Anzahl an Partikeln und gleicher Anzahl an Iterationen aber unterschiedlichem Zeitschritt haben wir festgestellt, dass dieser keine Auswirkungen auf die Laufzeit unserer Anwendung hat. Folglich betrachten wir diesen Fall im Folgenden nicht weiter.

Weiterhin haben wir noch die Auswirkung der Anzahl an Iterationen auf die Laufzeit unserer sequentiellen Implementation betrachtet. Dazu haben wir wieder alle Messungen auf der Partition *west* durchgeführt und eine konstante Anzahl an Partikeln von 1000 und konstanten Zeitschritt von 0.01 für alle Durchläufe gewählt. Entsprechend veränderte sich nur die Endzeit bei jedem Durchlauf und damit die Anzahl an Iterationen.

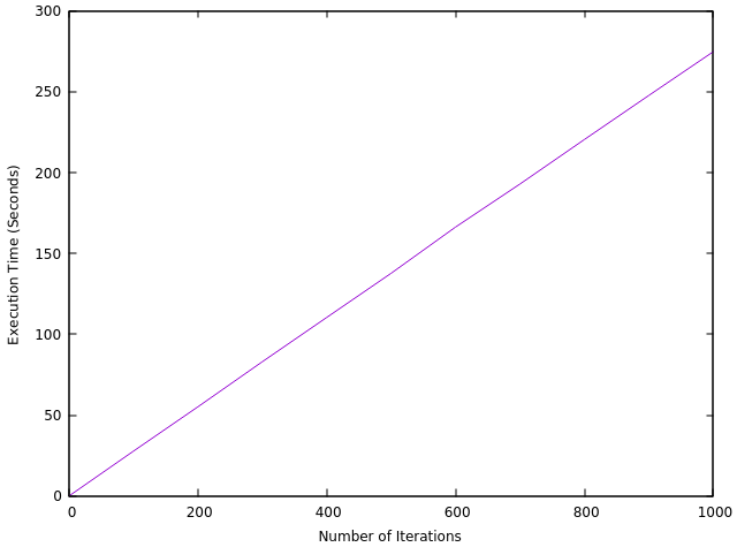

Figure 6.2: Laufzeiten bei steigender Anzahl an Iterationen (sequentiell)

Es lässt sich ein deutlicher Anstieg der Laufzeiten erkennen, allerdings ist das Wachstum der Kurve diesmal linear. Dennoch lässt der Vergleich der konkret gemessenen Laufzeiten erkennen, dass das Wachstum nicht ganz stetig ist: Im Mittel schwankt der Faktor um den sich die Laufzeit verlängert zwischen 1,2 und 1,5.

6.2 Parallele Laufzeitmessungen

Die Laufzeiten der parallele Anwendung haben wir mit der selben Methodik gemessen, wie die der sequentiellen. Auch hier haben wir zuerst die Laufzeiten bei unterschiedlicher Anzahl an Partikeln gemessen, wobei der Zeitschritt mit 0.01 und die Endzeit mit 1 für alle Durchläufe gleich waren. Diese Messungen wurden ebenso alle auf der Partition *west* durchgeführt. Alle Durchläufe wurden auf einer Node und mit vier Prozessen gemessen. Wieder lässt sich ein deutliches exponentielles Wachstum erkennen, jedoch verläuft dieses nicht so steil wie noch bei der sequentiellen Anwendung.

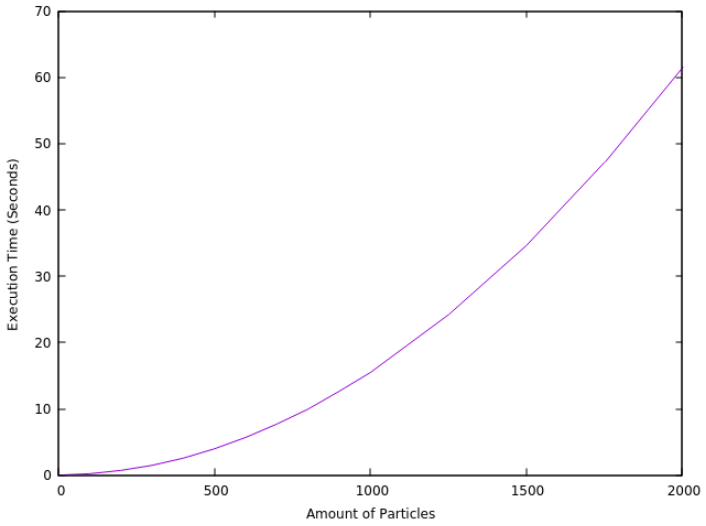

Figure 6.3: Laufzeiten bei steigender Anzahl an Partikeln (parallel)

Ebenso haben wir auch die Auswirkungen der Anzahl an Iterationen auf die parallele Anwendung gemessen, mit den selben Vorgaben wie bei der sequentiellen Laufzeitmessung: Anzahl der Partikel ist 1000 und Zeitschritt ist 0.01 für alle Durchläufe. Alle Durchläufe wurden auf Partition *west* mit einer Node und vier Prozessen durchgeführt.

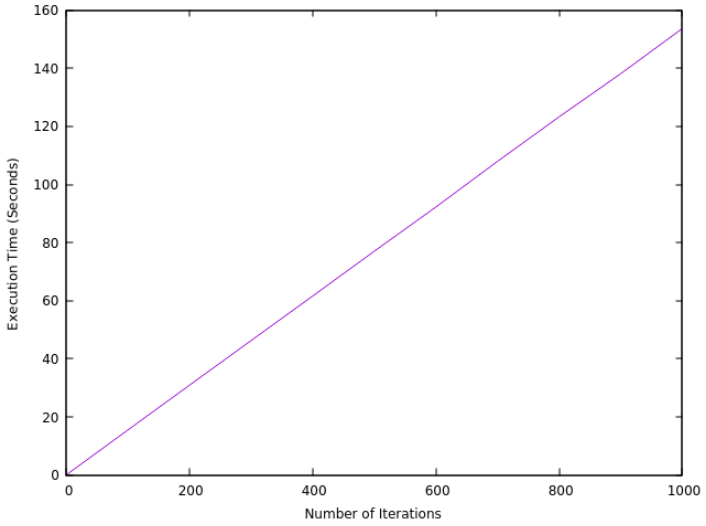

Figure 6.4: Laufzeiten bei steigender Anzahl an Iterationen (parallel)

Die Kurve lässt keinen Unterschied zu der sequentiellen Anwendung erkennen, allerding weist die y-Achse darauf hin, dass dieser Graph wesentlich flacher verläuft als die der sequentiellen Messungen. Auch die konkreten Messwerte zeigen, dass unsere parallele Anwendung zwar schneller ist als die sequentielle, aber dennoch das selbe lineare Wachstum der Laufzeiten zeigt.

6.3 Vergleich

Im direkten Vergleich der Messungen von sequentieller und paralleler Anwendung lässt sich deutlich erkennen, dass die parallele Anwendung nicht nur schneller ist als die sequentielle, sondern deren Laufzeiten auch wesentlich weniger stark ansteigen bei zunehmender Anzahl an Partikeln bzw. Iterationen.

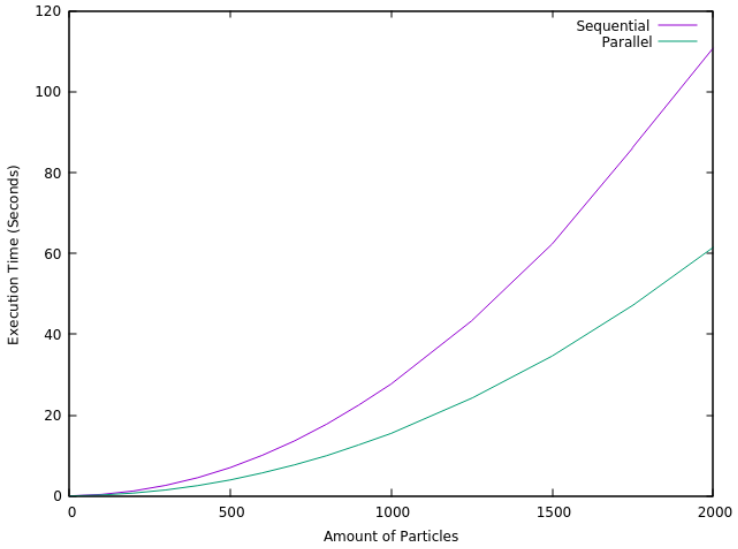

Figure 6.5: Laufzeiten bei steigender Anzahl an Partikeln (kombiniert)

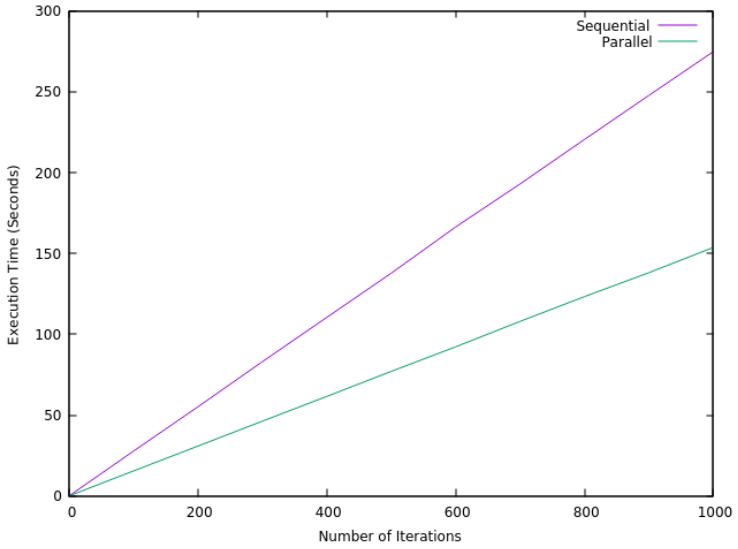

Figure 6.6: Laufzeiten bei steigender Anzahl an Iterationen (kombiniert)

7 Leistungsanalyse und Skalierbarkeit

7.1 Sequentielle Leistungsanalyse

Die sequentielle Leistungsanalyse beschränkte sich vorallem auf die Laufzeitmessungen und Tests mit Valgrinds Tools *memcheck* und *cachegrind*. *Memcheck* hat keinerlei memory leaks aufgezeigt und sonst auch keine weiteren Fehler offengelegt, was unser Vertrauen in unsere Implementation nur noch stärkte. Auch *cachegrind* zeigte keine Anomalien auf, zwar hatten wir einige Cache Misses zu verzeichnen, allerdings beschränkten sich diese zum größten Teil auf Zugriffe und waren im Vergleich zu der Gesamtzahl and Reads und Writes nicht signifikant, sodass Valgrind uns insgesamt eine 0,0% Miss Rate anzeigte. Des Weiteren haben wir Spurdatenanalysen durchgeführt, um herauszufinden, wo wir am Besten mit unserer Parallelisierung ansetzen.

7.2 Parallele Leistungsanalyse

Da die Tools von Valgrind nicht oder nur enorm beschränkt für mit MPI parallelisierte Anwendungen funktionieren, haben wir diese bei der parallelen Leistungsanalyse außen vor gelassen und uns vorallem auf die Spurdatenanalyse mit Vampir beschränkt.

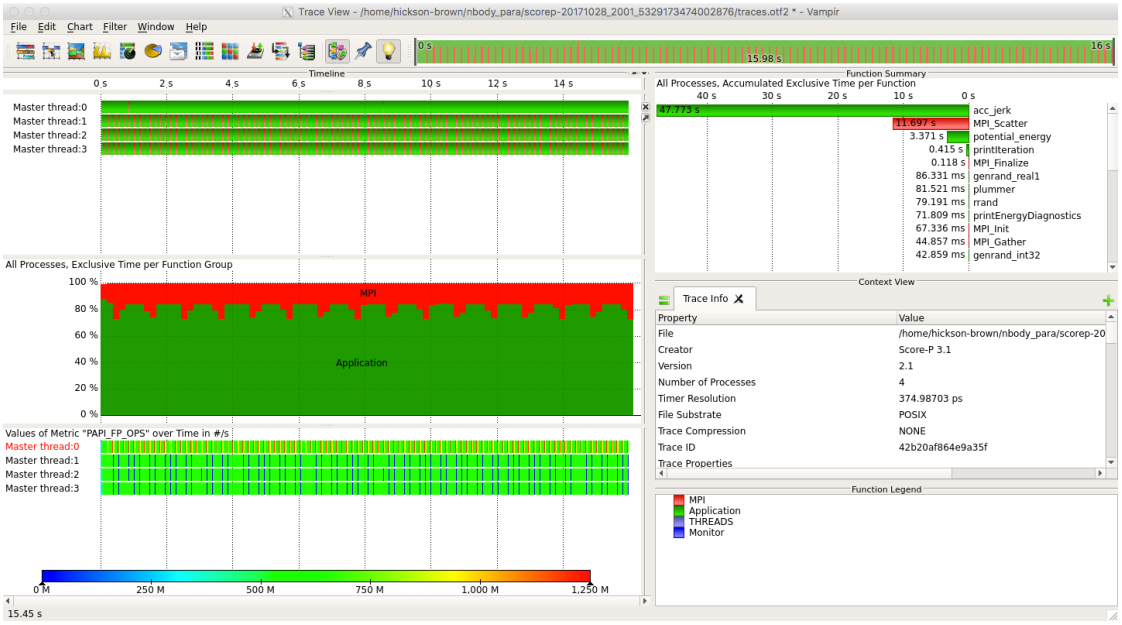

Figure 7.1: Spurdatenanalyse eines Durchlaufs mit 1000 Partikeln und 100 Iterationen

Deutlich zu erkennen ist, dass unsere Anwendung immer noch die meiste Zeit in der Funktion *acc_jerk* verbringt, allerdings wesentlich weniger Zeit als zuvor. Allerdings sehen wir auch, dass unsere Anwendung am zweitlängsten mit *MPI_Scatter* verbringt. Zwar ist der Anteil von MPI nur bei 20%, allerdings sind die MPI Anteile regelmäßig in Prozessen 1 bis 3 vertreten. Verwunderlich ist nur, dass dies nicht der Fall im Root-Prozess 0 ist.

Mit einigen gezielt gesetzten *MPI_Barrier* Aufrufen konnten wir dann unsere Hypothese bestätigen: Der hohe Anteil an MPI ist unserem Parallelisierungschema und der Tatsache geschuldet, dass Root die meiste Arbeit erledigt, während alle anderen Prozesse auf ihn warten müssen.

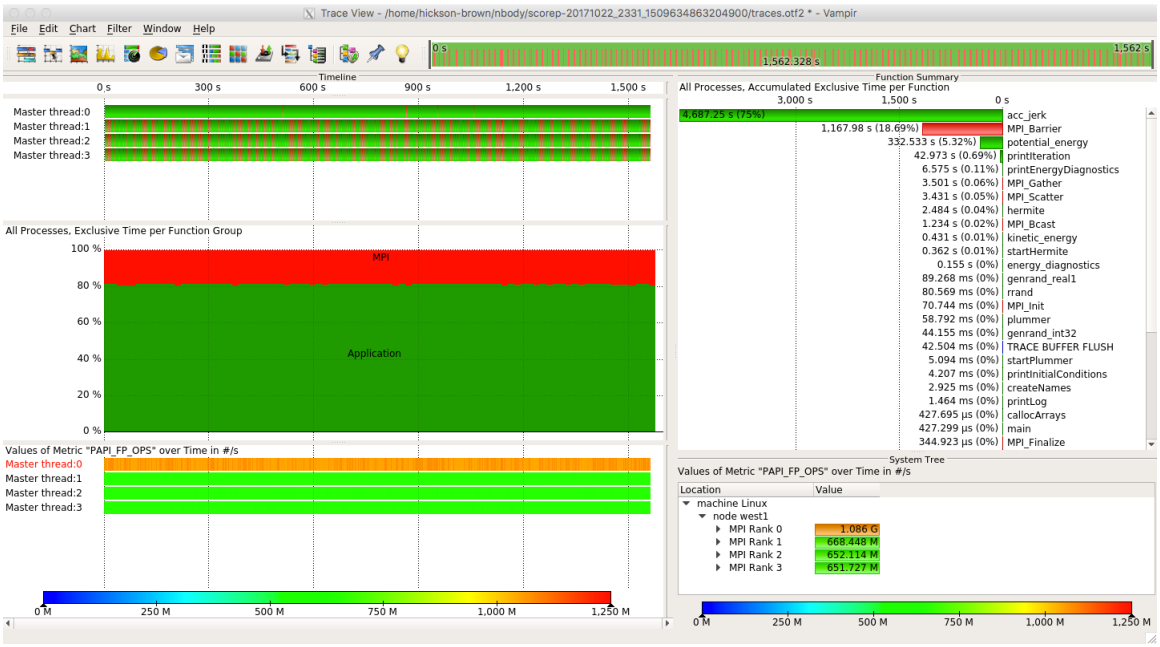

Figure 7.2: Spurdatenanalyse eines Durchlaufs mit 1000 Partikeln und 10000 Iterationen

Wir können deutlich sehen, dass die Prozesse 1 bis 3 ca. 99% des MPI-Anteils damit verbringen, auf den Root-Prozess 0 zu warten, der in der Zeit vorallem I/O-Operationen durchführt. Trotzdem ist unserer Ansicht nach der Anteil an MPI in unserer Anwendung nicht zu hoch, denn immerhin läuft unsere parallel Anwendung nun wesentlich schneller als die sequentielle und der Overhead, der durch die Arbeitsteilung erzeugt wird, kann durch eine entsprechende Anzahl an Iterationen bzw. Partikeln kompensiert werden.

7.3 Skalierbarkeit der parallelisierten Anwendung

Um herauszufinden, wie unsere parallelisierte Anwendung über verschiedene Anzahlen an Prozessen skaliert, haben wir weitere Laufzeitmessungen mit einer festen Anzahl an Partikeln, Zeitschritt und Endzeit (strong scaling) durchgeführt. Die Anzahl an Partikeln betrug dabei 1080, der Zeitschritt 0.01 und die Endzeit 1, womit 100 Iterationen pro Durchlauf erreicht werden. Alle Messungen wurden auf der Partition *west* mit variablen Anzahlen an Prozessen durchgeführt.

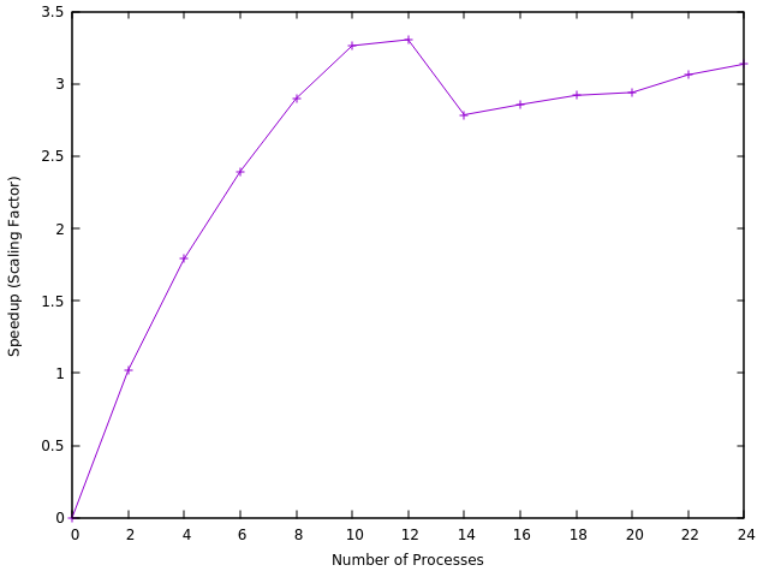

Figure 7.3: Speedup-Diagramm mit strong scaling

Unsere parallele Implementation skaliert unserer Ansicht nach sehr gut und ist insbesondere für hohe Anzahlen an Prozessen gut einsetzbar. Den höchsten Speedup erreicht man mit 12 Prozessen und einem Speedup von 3.30587. Danach fällt dieser zwar leicht ab, scheint sich aber wieder zu erholen. Unsere Vermutung ist, dass sich ein weiterer Speedup-Peak in der Nähe von 30 Prozessen befindet, diese Vermutung lässt sich aber leider von uns nicht überprüfen auf Grund der Limitierungen der Partition *west*, die auf 24 Prozesse beschränkt ist.

Im Groben und Ganzen sind wir aber mit unserer Implementierung zufrieden, wir konnten zeigen, dass sich auch eine Anwendung wie die unsere mit einer Kompexität von $O(n^2)$ gut parallelisieren lässt und damit signifikante Leistungssteigerungen zu erzielen sind, bei gleichbleibender Genauigkeit der Berechnungen.

8 Ausblick: Verbesserungen

Zwar sind wir mit der Laufzeitverbesserung, die wir erreicht haben, schon sehr zufrieden, dennoch gibt es noch Verbesserungen die man vornehmen könnten. So ist zum Beispiel das Parallelisierungsschema noch nicht optimal. Wesentlich effizienter wäre es, wenn jeder Prozess an seinem eigenen Stück des Puzzles arbeiten würde und nicht ein Prozess alle Stücke im Blick haben müsste.

Eine Verbessserung des Parallelisierungsschemas wäre es, wenn lediglich die Erzeugung initialer Konditionen auf dem Root-Prozess stattfindet und dann auf die Prozesse aufgeteilt wird. Sämtliche I/O-Operationen können dann mit MPI I/O durchgeführt werden, sodass die Prozesse parallel in Dateien schreiben bzw. sich beim Schreiben abwechseln. An Stellen wo kommuniziert werden muss kann eine Art Pipeline implementiert werden, mit dem Effekt, dass jeder Prozess seine Partikel zu einem Zeitpunkt nur dem nächsten Prozess kommuniziert, ähnlich einem Token-Ring wie er in der Netztwerktechnik verwendet wird. Damit würde man auch das Problem der Doppelberechnung eliminieren und nicht nur noch mehr Zeit gewinnen, sondern auch die Skalierbarkeit erhöhen.

Zusätzlich könnte man die Thematik der Datenspeicherung näher betrachten, die in unserer jetzigen Implementation etwas unter den Tisch gefallen ist. Sämtliche erzeugten Daten werden momentan in von Menschen lesbare Dateien geschrieben, meistens im *.csv-Format, die sehr viel Speicherplatz in Anspruch nehmen. So kann ein Durchlauf mit einigen tausend Iterationen bereits mehrere Gigabyte groß sein. Alternative Methoden zur Speicherung wären sinnvoll, beispielsweise könnte man ein Dateiformat verwenden, welches nur maschinenlesbaren Output erzeugt, um damit Speicherplatz einzusparen, wenn möglich.

Auch ist die Frage, ob C die effizienteste Sprache für ein solches Vorhaben ist, interessant zu erörtern. So könnte man unseren N-body Code in Fortran übersetzen und durch gezielte Messungen und Vergleiche beider Implementationen herausfinden, ob Fortran einen Vorteil gegenüber C besitzt, da heutzutage die meisten Anwendungen, die änhlich viele numerische Daten wie die unsere bearbeiten, immer noch in Fortran geschrieben sind.

Ebenso wäre die Visualisierung noch ausbaufähig. Da die Daten beim Visualisieren in Echtzeit von der Festplatte gelesen werden, stellt hier die Lesegeschwindigkeit den limitierenden Faktor dar. Weil die Grafikberechnungen vergleichsweise simpel sind, müsste die GPU bei einer hohen Anzahl an Partikeln auf die Festplatte warten. Sobald das Auslesen einer Iteration länger dauert als ca. 1/25 einer Sekunde wäre die Darstellung nicht mehr flüssig. Die von uns verwendeten ca. 1000 Partikel konnten jedoch stets schnell genug ausgelesen und damit zufriedenstellend dargestellt werden.

Ein möglicher Ansatz wäre es die Daten in einem Puffer vorzuladen, damit diese direkt aus dem Arbeitsspeicher gelesen werden können. Außerdem könnten die Daten auch, wie bereits erwähnt, in einem effizienteren Format gespeichert und eventuell auch komprimiert werden.

Viele unserer Ideen zur Visualisierung konnten wir leider aus Zeitgründen nicht umsetzen. Erwähnt wären hier, beispielsweise diverse Effekte zur realistischeren Darstellung. Ebenso wäre eine Markierung der Partikel per Mausklick oder Auswahl inklusive einer Anzeige von Partikelspezifischen Daten in einer ansprechenden GUI interessante Ideen die umgesetzt werden könnten. Da der Fokus dieses Praktikums jedoch auf der Parallelisierung liegt und auf jede neuen Idee zwei weitere folgen haben wir einen unserer Problemstellung entsprechenden Schnitt ziehen und uns auf das Wesentliche beschränken müssen.

Insgesamt lässt sich festhalten, dass uns die in das Projekt investierte Zeit zu wichtigem und in Zukunft anwendbarem Wissen verholfen hat. Das Lösen der Problemstellung war zwar mit viel Aufwand aber auch mit viel Spaß verbunden und die von uns erzielten Ergebnisse sind unserer Meinung nach sehr bemerkenswert

Bibliography

[AHW74] S. J. Aarseth, M. Henon, and R. Wielen. A comparison of numerical methods for the study of star cluster dynamics. *Astronomy and Astrophysics*, 37(12):183–187, Dezember 1974.

[HM07] Piet Hut and Jun Makino. *The Art of Computational Science: The Kali Code*, volume 2 of *The Maya Open Lab for Dense Stellar Systems*. Hut, Piet and Makino, Jun, September 2007.

[MN07] Makoto Matsumoto and Takuji Nishimura. Mersenne twister in c, c++, c#. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/c-lang.html`, 2007. [Online, abgerufen 09.10.2017].