HOCHSCHULE **TRIER**
Trier University of Applied Sciences
Informatik - Computer Science

MPI Parallelisierung am Beispiel des Boids Algorithmus

Oliver Biwer

Seminar zur Vorlesung SW Entwicklung für Many-Core High-Performance Computing Systeme

Betreuer: Prof. Dr. Jörn Schneider, Tillmann Nett, Andreas Schleicher

Trier, 28.07.2014

Kurzfassung

Diese Arbeit befasst sich mit Parallelisierung von Software mittels Massage Passing Interface am Beispiel des Boid Algorithmus. Es wurden zwei verschiedene Ansätze für die Parallelisierung ausgearbeitet, zum einen die Aufteilung der Boids auf Prozesse und zum anderen die Aufteilung der "Spielfläche" auf Prozesse.

Bereits während der Entwicklung wurde klar, das Parallelisierung nicht zwingend ein Geschwindigkeitszuwachs bedeutet. Und bereits während der Entwicklung ist klar geworden, dass es nicht immer einfach ist, Aufgaben parallel zu gestalten.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

Mit fortschreitender Technik werden unsere Computer immer Leistungsfähiger. Die meisten aktuellen Systeme für Zuhause haben mindestens 2-Kern oder 4-Kern Prozessoren. Doch macht es kein unterschied ob Heim-PC, Server oder Supercomputer[1], die Hardware nützt nur wenig, wenn das Programm, welches man ausführen möchte, nicht mit dieser umzugehen weiß. Softwareentwickler müssen dazu ihre Programme parallelisieren, also dafür sorgen, dass sich die verschiedenen Prozessoren oder Prozessor Kerne die Aufgabe(n) des Programms teilen. Leider gibt es keine "All-Zweck-Methode" wie man eine Software parallelisiert. Dies ist von Programm zu Programm unterschiedlich und erfordert viel Zeit und Erfahrung.

Aus diesem Grund beschäftigt sich diese Seminararbeit damit, anhand von einfachen Algorithmen verschiedene Möglichkeiten der Parallelisierung vorzustellen. Eine dieser Möglichkeiten ist MPI, das Massage Passing Interface. Mit diesem befasst sich diese Arbeit und stellt zwei Herangehensweisen zur Parallelisierung des Boids Algorithmus vor.

Das Ziel dieser Seminararbeit war es zu Zeigen, das aus einer Parallelisierung nicht zwingend eine höhere Leistung zu schließen ist und zu erkennen, wann Parallelisierung Sinn macht und wann nicht.

Die folgenden Randbedingungen sind für die Ausarbeitung gegeben:

Entwicklung eines funktionsfähigen Boids Algorithmus

Visuelle Darstellung (optional) grafische Darstellung damit die Funktionsfähigkeit des Algorithmus visuell überprüft werden kann.

Parallelisierung mittels MPI.

Profiling (optional) um das Programm weiter zu optimieren.

Benchmarking auf dem Tilera System, um zu überprüfen ob die Nebenläufigkeit einen Geschwindigkeitsvorteil bringt.

[1] Supercomputer bezeichnet die stärksten Computer ihrer Zeit, meist haben sie eine große Anzahl an Prozessoren

2

Grundlagen

Dieses Kapitel behandelt Grundlagen auf der die, in dieser Arbeit entwickelten Programme basieren.

2.1 Boids Algorithmus

Der Boids Algorithmus ist ein Programm zur Simulation eines Vogelschwarms. Jeder einzelne dieser Vögel folgt in der klassischen Implementierung drei einfachen Regeln. Daraus ergibt sich ein komplex wirkendes Verhalten[Wika], das für Künstliches Leben[Wikb] Programme typisch ist.

Diese drei Regeln sind wie folgt definiert:

Cohesion - Zusammenhalt

Die mittlere Position des Boid-Schwarms wird bestimmt. Jeder einzelne Boid versucht seine Position dieser mittleren Position anzunähern[Kfi, rule 1]. Dies wird in Abbildung 2.1 dargestellt.

Seperation - Auftrennung

Diese Regel versucht Kollisionen zu vermeiden. Kommen sich zwei Boids, wie in Abbildung 2.2 zu sehen, auf einen festgelegten Mindestabstand näher, schlagen beide eine genau entgegengesetzte Richtung ein[Kfi, rule 2].

Alignment - Angleichung

Die mittlere Geschwindigkeit des Schwarms wird bestimmt. Jeder Boid versucht seine Geschwindigkeit in Richtung und Betrag der des Schwarms anzupassen[Kfi, rule 3]. Dies wird in Abbildung 2.3 gezeigt.

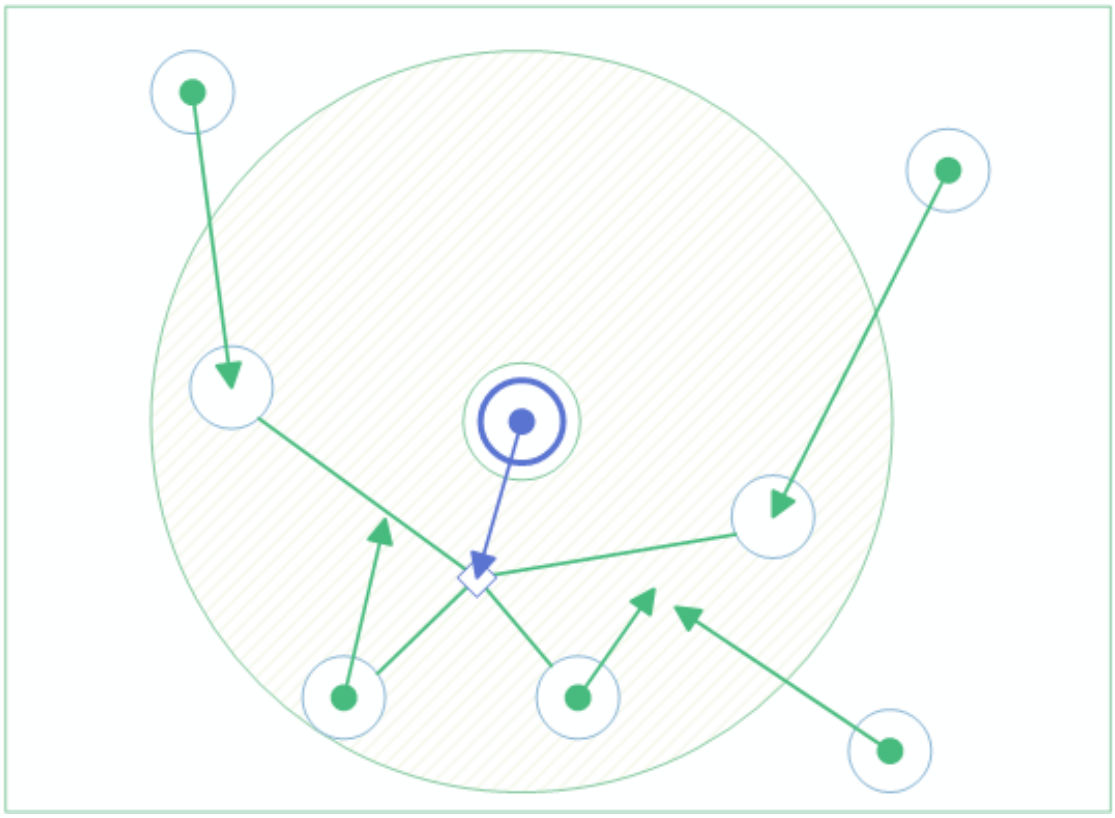

Abb. 2.1. Die grün-straffierte Fläche ist der Sichtbereich des blauen Boid. Die Pfeile entsprechen den Geschwindigkeitsvektoren nach Anwendung der Cohesion-rule. Die Pfeile zielen auf den Mittelpunkt des für ihn Sichtbaren Schwarms, wie am blauen Pfeil verdeutlicht.

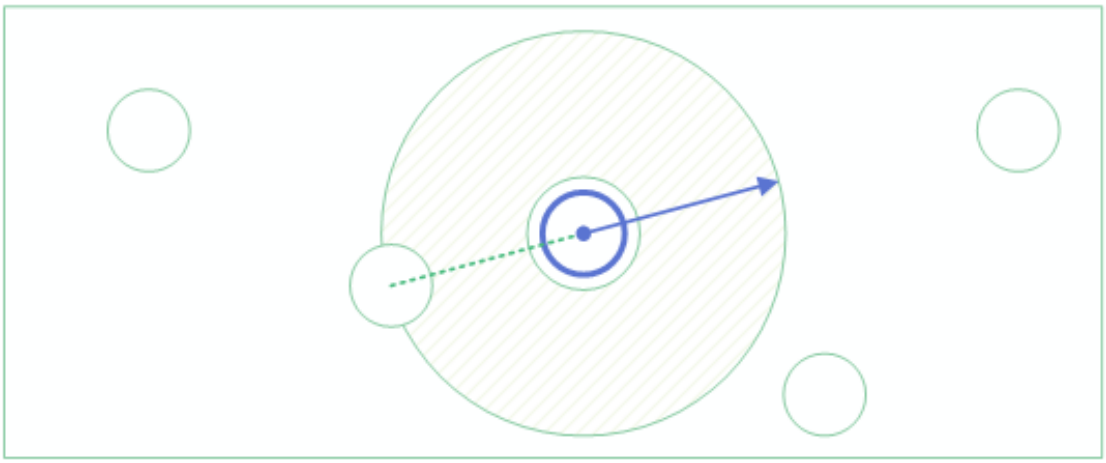

Abb. 2.2. Die grün-straffierte Fläche ist der Mindestabstand, der zwischen zwei Boids angestrebt wird. Der blaue Pfeil ist der resultierende Geschwindigkeitsvektor für den blauen Boid.

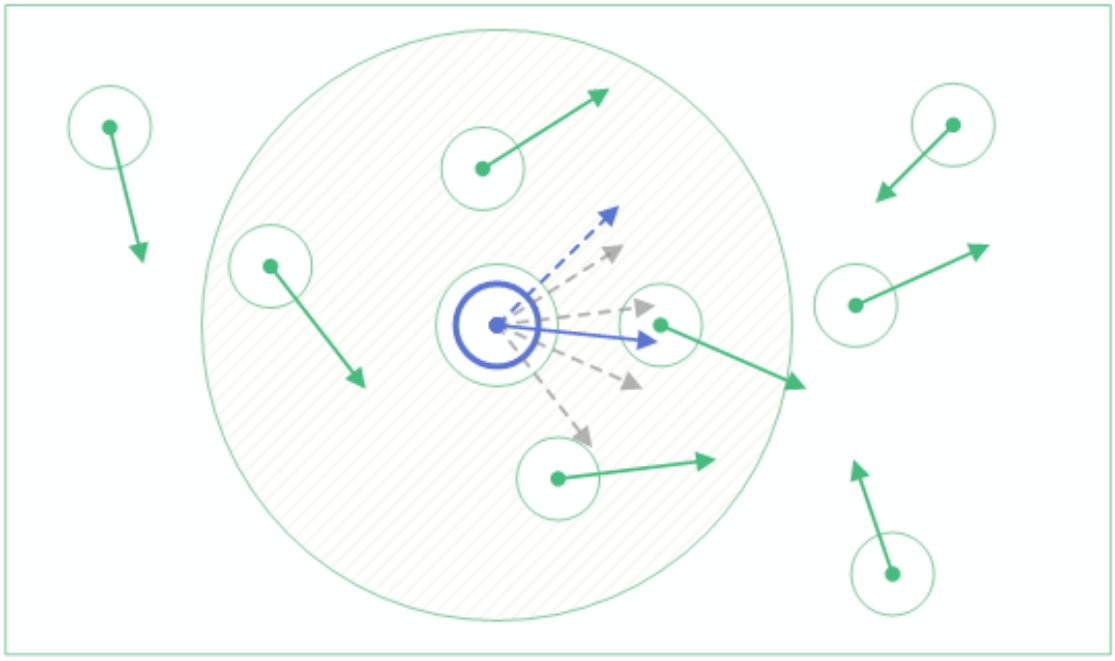

Abb. 2.3. Die grauen Pfeile sind Projektionen der grünen Geschwindigkeit Vektoren. Der blaue, gestrichelte Pfeil entspricht der ursprünglichen Geschwindigkeit, der blaue Pfeil mit der durchgezogenen Linie ist die Geschwindigkeit nach der Angleichung.

2.2 Message-Passing Interface

Die Parallelisierung des Boid Algorithmus wird mittels Message-Passing Interface (MPI) Realisiert.

> MPI ist ein Standard der den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen beschreibt[Wikc].

Das Programm, welches mittels MPI parallelisiert wird, wird dabei auf jedem Prozess separat ausgeführt. Dabei können auch Prozessoren mehrerer Computer verwendet werden. Die Prozesse senden sich untereinander Teil- oder Endergebnisse zu. Prozesse sind in so genannte *Communicatoren* Gruppiert. Innerhalb einer Gruppe wird jedem Prozess eine eindeutige ID zugeteilt. Die ID ist eine fortlaufende Nummer von 0 bis $n - 1$ für n Prozesse.

3

Implementierung

Der Quellcode des Projektes ist hier einzusehen:
https://github.com/soraphis/Boids-MPI/tree/v1.0

Die Implementierung des Projekts geschieht mittels C++ und der MPI Bibliothek MPICH in der Version 3.0.4. Der grundlegende Aufbau des Programms sieht wie folgt aus. Zu Beginn muss MPI initialisiert werden, woraufhin auf jedem Prozess ein Modell angelegt, welches die Boid-Logik und die Boids selbst enthält. Nach der Initialisierung Arbeitet das Programm für eine fest definierte Zeit eine Update-Schleife ab.

3.1 Boids

Die Boid Klasse besitzt *Attribute* zum Speichern der `Position` und der `Geschwindigkeit` des Boids. Außerdem hat die Klasse ein *Vektor* auf einen `Regel` *Zeiger*. Dies erlaubt es Boids unterschiedliche Regeln zu zuweisen und gestaltet das Programm insgesamt dynamischer.

`Position` und `Geschwindigkeit` sind Objekte einer Vektor-Klasse. Diese besteht aus zwei *float* Werten und stellt ein paar Methoden der Vektorrechnung zur Verfügung.

Alle Regeln erben von der *rein virtuellen* Oberklasse `IBoidRule`, welche die Methode `followRule` vorschreibt. Ein Boid ruft für jede an ihm angemeldete Regel diese Methode auf.

Regeln

Die in dieser Arbeit entworfenen Boids folgen 5 Regeln. Dies sind zum einen die drei bereits vorgestellten Regeln, die den Boids Algorithmus ausmachen. In dieser Implementierung sind alle drei Regeln mit einer Sichtweite des Boids implementiert. Die vierte Regel limitiert die Geschwindigkeit auf eine definierte maximal Geschwindigkeit. Die letzte Regel biegt die ”Spielfläche” zu einem Kreis. Das heißt,

wenn ein Boid die Spielfläche rechts verlässt, so betritt er sie wieder auf der linken Seiten.

3.2 Modell

BoidModel

Dieses Modell weist jedem Prozess eine gleich hohe Anzahl an Boids zu. Aus der Hauptfunktion wird die `init`-Methode des BoidModel aufgerufen. Der Hauptprozess (Prozess 0) wählt nun für jeden zu erstellenden Boid Zufallszahlen für dessen Position aus. Diese Zahlen teilt er den anderen Prozessen mit und jeder Prozess erstellt nun einen Boid an eben diesen Koordinaten. Dadurch wird sicher gestellt, dass sich für jeden Prozess alle Boids an den gleichen Positionen befinden.

Danach wird in einer Update-Schleife immer wieder die `update`-Methode des Modells aufgerufen. Die Boids werden gleichmäßig auf alle Prozesse aufgeteilt. Läuft das Programm zum Beispiel auf 4 Prozessen so übernimmt der Prozess mit der ID 0 die Boids an den Indexpositionen {0, 4, 8, 12, ...} und der Prozess mit der ID 1 die Boids {1, 5, 9, 13, ...}, usw.

Die Prozesse wenden auf jeden Boid die Regeln an und aktualisieren seine Position durch Addition mit dem aktuellen Geschwindigkeitsvektor. Damit im nächsten Update-schritt die Regeln wieder korrekt Angewendet werden können, teilen die Prozesse nun die Positionen und Geschwindigkeiten ihrer Boids den anderen Prozessen mit.

Die Implementierung gestaltet sich vergleichsweise einfach, denn es werden nur an einer Stelle Daten zwischen den Prozessen ausgetauscht. Die Aufteilung der Boids funktioniert mittels Modulo-Operation und die `update`-Methode besteht im wesentlichen nur aus 3 Schleifen. Abbildung 3.1 zeigt den Ablauf schematisch. Die Schleifen können aus verschiedenen Gründen nicht zusammengefasst werden. So müssen die Regeln auf die Positionen der Vorangegangenen `update`-Methode ausgeführt. Außerdem sollte das Versenden der Nachrichten an die anderen Prozesse in einer Schleife sein, die nicht rechenintensiv ist.

AreaModel

Dieses Modell teilt die "Spielfläche" gleichmäßig auf die Prozesse auf. Boids werden untereinander von einem Prozess zum anderen geschickt falls sie ein Areal verlassen. Die Initialisierung dieses Modells beginnt ebenfalls damit, das der Hauptprozess zufällige Koordinaten bestimmt. Aus diesen Koordinaten berechnet er, in welchem Areal sich der Boid befindet und teilt dies dem Prozess mit, der sich um dieses Areal kümmert. Dieser Prozess legt dann ein Boid mit den errechneten Koordinaten an.

Beim Update arbeitet jeder Prozess alle Boids ab, die sich in seinem Areal befinden. Das heißt, er wendet die Regeln auf die Boids an und aktualisiert ihre Position. Befindet sich die Position außerhalb des jeweiligen Areals, wird die Position und Geschwindigkeit des Boids von Prozess A an Prozess B gesendet. Der nun für

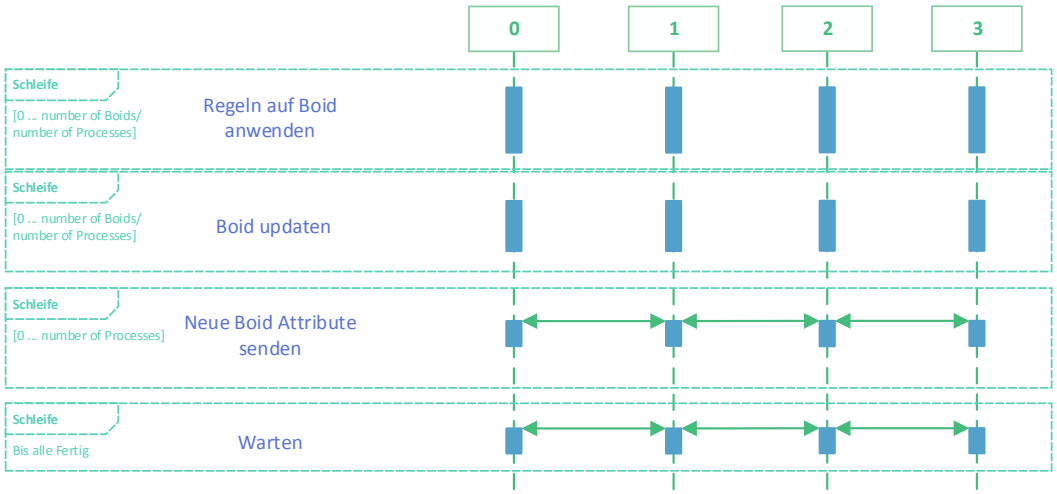

Abb. 3.1. Beispiel für 4 Prozesse. Pfeile stellen Nachrichtenaustausch dar.

den Boid verantwortlich ist. Prozess A legt den Boid entsprechend bei sich an und Prozess B entfernt den Boid.

Abbildung 3.2 verdeutlicht diesen Prozess noch einmal. Beim Vergleich mit dem BoidModel fällt auf, dass das Updaten der Boid Positionen und das Anwenden der Regeln vertauscht sind. Dies dient dazu eine höhere Ausfallsicherheit zu gewährleisten. Beim ”Boid Positionen auswerten” geht es darum, herauszufinden welcher Boid das Areal des Prozesses verlassen hat, und welches nun sein neues Areal ist. Jeder Prozess teilt den jeweils anderen mit, wie viele Boids er ihm senden wird. Alle Boids, die an den selben Prozess geschickt werden in ein Array zusammengefasst. Jeder Prozess bekommt von jedem anderen ein Array, welches Attribute für Boids beinhaltet.

3.3 Visualisierung

Zur Visualisierung wurde die SDL Bibliothek eingebunden. Es wird ein Fenster angezeigt welches die ”Spielfläche” symbolisiert. Wie in Abbildung 3.3 zu sehen, befinden sich auf der weißen ”Spielfläche” 3×3 Pixel große, schwarze Rechtecke, diese Stellen die Boids dar.

Im AreaModel werden mehrere Fenster erstellt, wobei jeder Prozess sein eigenes rendert sein Fenster. Die Boids können sich über die Fensterrahmen hinweg bewegen, wie die Abbildungen 3.4 und 3.5 veranschaulichen. An einigen Stellen kann es so aussehen als ob Boids verschwunden sind, dieser Eindruck entsteht dadurch, dass die Boids bereits bei Prozess A entfernt wurden und in Prozess B zwar angelegt, aber noch nicht gerendert. Zur Erfassung von Messwerten wurde die grafische Darstellung abgeschaltet.

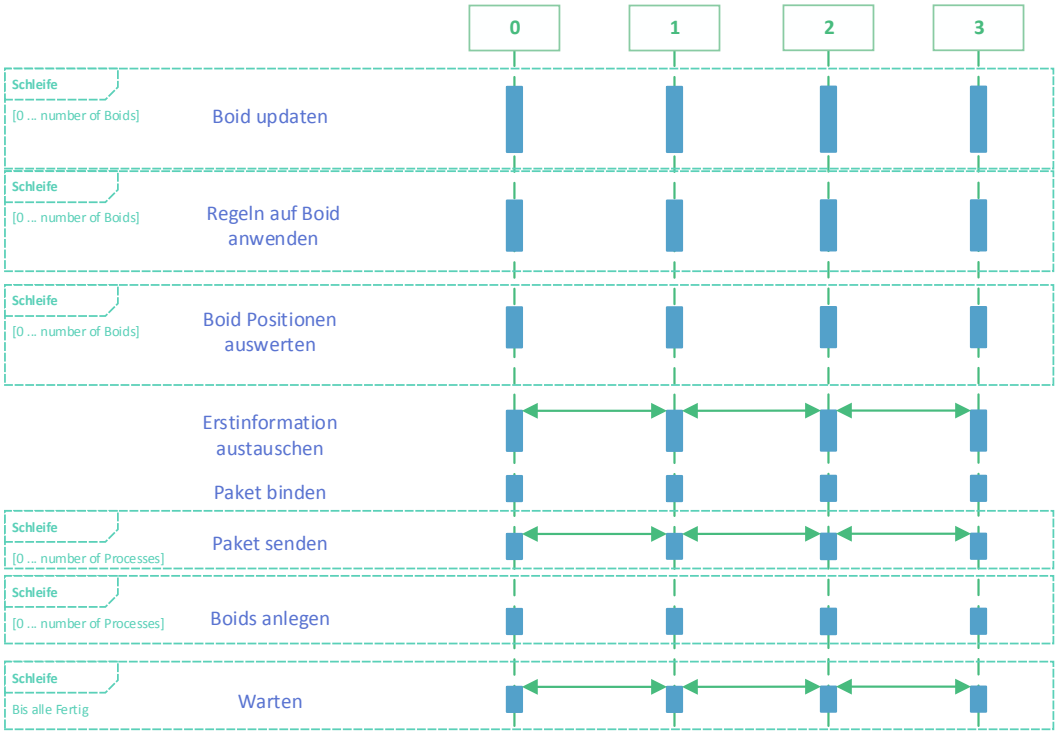

Abb. 3.2. Beispiel für 4 Prozesse. Pfeile stellen Nachrichtenaustausch dar. ”Paket binden” ist ein Ausdruckt dafür, das alle zu versendenden Boids in ein großes Array geschrieben werden

Abb. 3.3. Die schwarzen Punkte stellen Boids dar. Man erkennt wie sich Schwärme gebildet haben.

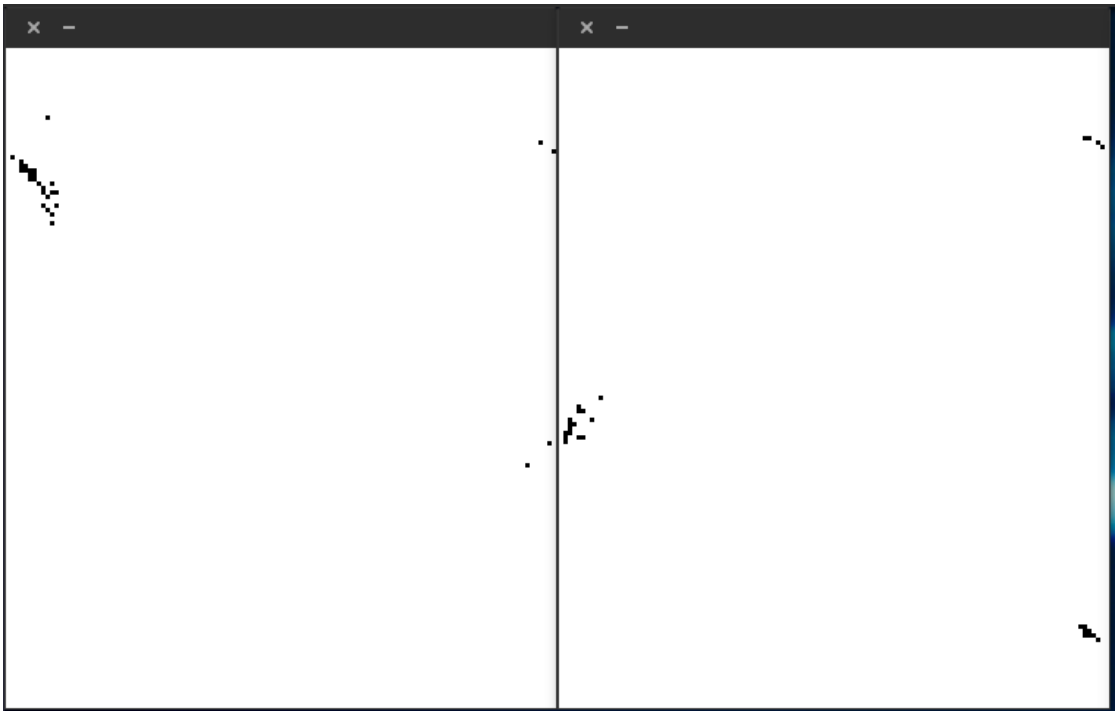

Abb. 3.4. Eine Szene im AreaModel mit zwei Prozessen.

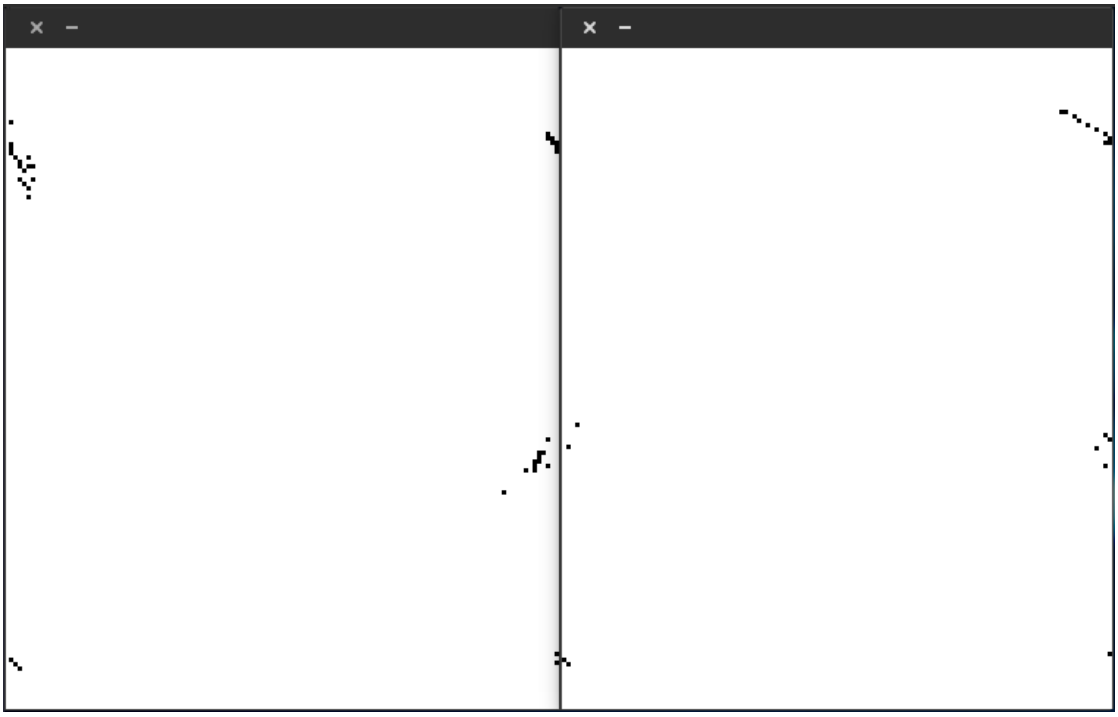

Abb. 3.5. Zeigt die selbe Situation wie Abbildung 3.4. Man sieht wie sich die Schwärme durch Fensterkanten bewegen

4

Profiling

Beim Profiling geht es darum, herauszufinden wo im Programm die meisten Berechnungen stattfinden, wo viel Zeit unnötig verbraucht wird und was optimieren werden sollte damit das Programm schneller bzw. effizienter läuft.

Zum Profiling wurde "Allinea Map"[All] verwendet. Das Programm ist speziell auf MPI ausgerichtet und bietet neben CPU Profiling auch eine Anzeige, wie lange die Nachrichten, die mittels MPI versendet werden, benötigen um von einem Prozess versendet und beim anderen empfangen zu werden. Mithilfe des Programms konnte bereits zu Beginn der Entwicklung feststellen, dass eine unnötig hohe Zeit im Dekonstruktor der Vektor-Klasse verbracht wurde, dies konnte minimiert werden, indem auf möglichst viele lokale Variablen dieser Klasse verzichtet habe. Auch konnte durch Profiling herausfinden das der CPU über 80% der Zeit zur Berechnung der Länge eines Vektors in Anspruch genommen hat. Dies ist jedoch keineswegs negativ. Die Methode wird zum Beispiel dazu benötigt die Strecke zwischen zwei Boids zu ermitteln, was in vier von fünf Regeln benötigt wird.

Abbildung 4.1 zeigt ein Auszug aus dem Profiling-Programm. Der Auszug zeigt deutlich eine Schwachstelle im Code, bis zu 32.8 ms wird auf einen Prozess gewartet. Dies liegt an der Art und Weise, wie das AreaModel Implementiert wurde. Sind die Teilfelder sehr groß, kann der gesamte Schwarm in einem Feld sein und wird von einem einzigen Prozess berechnet, die anderen Prozesse verbringen diese Zeit wartend.

Abb. 4.1. Ein Profiling des AreaModels zeigt, dass 78% der Zeit in MPI verbracht wurde und 70% davon in der "Warte"-Methode "MPI_Barrier"

5

Benchmarking

Beim Benchmarking soll erfasst werden, wie schnell ein Programm auf einem System unter bestimmten Einstellungen ausgeführt werden kann.

Damit diese Werte Rückschlüsse auf die Art und Weise der Parallelisierung liefern, wurden verschiedene Kombinationen getestet.

- Verwendung des BoidModel. Geschwindigkeit und Position der Boids als Fließkommazahl.
- Verwendung des BoidModel. Geschwindigkeit und Position der Boids als Festkommazahl.
- Verwendung des AreaModel. Geschwindigkeit und Position der Boids als Fließkommazahl.
- Verwendung des AreaModel. Geschwindigkeit und Position der Boids als Festkommazahl.

Durchgeführt wurden die Messungen auf folgenden zwei Systemen:

	Notebook	**Tilera**
Prozessor	i5 4200U	tile-gx 36
Frequenz	2,6 GHz	1,2 GHz
Kerne	2	36
Threads	4	36
GNU Compiler	4.8.2	4.4.7
MPICH	3.04	1.2.1

Alle Messwerte können hier eingesehen werden:
goo.gl/qvy6L8

Alle Messwerte basieren auf 30 Sekunden in denen 100 Boids berechnet wurden. Sie unterscheiden sich durch die Anzahl der gestarteten Prozessen.

Aufgrund der fehlenden Hardware für Fließkommaoperationen in früheren Generationen des Tilera Systems, wurde zwischen Fließ- und Festkommazahlen unterschieden.

5.1 BoidModel

Fließkommazahl gegen Festkommazahl

Abbildung 5.1 und 5.2 zeigen den Unterschied zwischen Fließ- und Festkommaoperationen am Beispiel des BoidModel. Hierbei zeigt die Hardware für Fließkommaoperationen deutlich ihre Leistung. Doch besonders auf dem Tilera System wird der Unterschied zwischen Fließ- und Festkommazahlen mit zunehmender Anzahl an gestarteten Prozessen verschwindend gering.

Beide Methoden sind aber weniger effizient sobald mehr[1] Prozesse gestartet werden. Ein deutliches Zeichen dafür, dass der Mehraufwand, der durch die Verwaltung der Prozesse entsteht, wesentlich größer ist, als die gewonnene Leistung durch Parallelisierung.

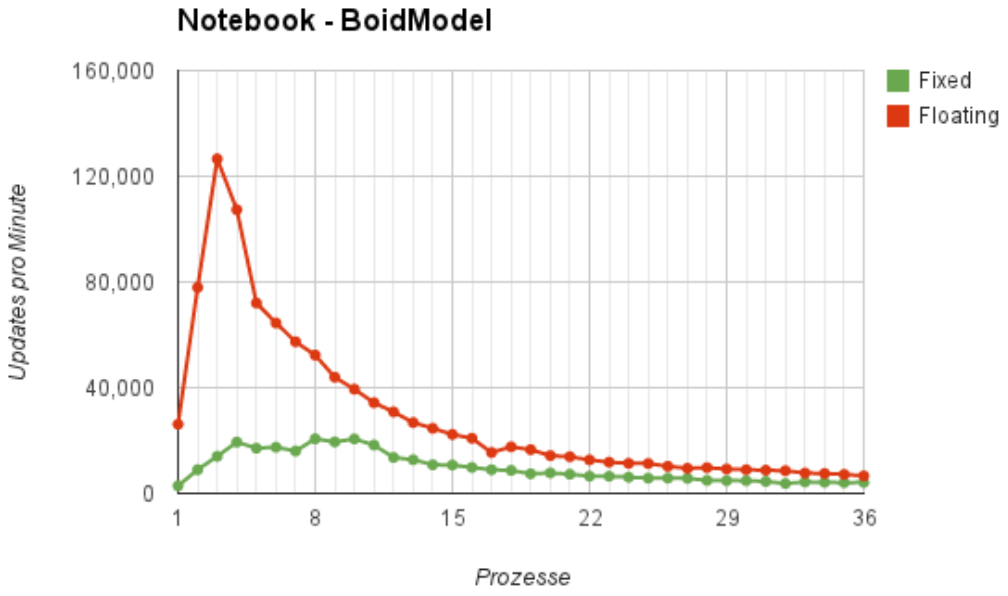

Abb. 5.1. Grün: Festkommazahl, Rot: Fließkommazahl. Messwerte: [Mes]

[1] Mehr heißt zwei oder mehr als das System Kerne hat.

Abb. 5.2. Grün: Festkommazahl, Rot: Fließkommazahl. Messwerte: [Mes]

5.2 Fließkommazahl

BoidModel gegen AreaModel

Die Abbildungen 5.3 und 5.4 zeigt den Vergleich zwischen den Beiden entworfenen Modellen. Das erste was auffällt ist, das im AreaModel Messwerte fehlen. Dies liegt daran, dass das Feld möglichst gleichmäßig aufgeteilt wird. Dies ist jedoch mit Primzahlen nicht oder nur bedingt möglich, deshalb bricht das Programm an dieser Stelle direkt nach dem Start ab. Auf dem Tilera (Abbildung 5.4) erkennt man, das der Parallelisierungsaufwand schon ab zwei Prozessen größer ist, als die Leistung die man dazu gewinnt. Interessant ist allerdings, dass auf dem, für den Heimgebrauch üblichen, Notebookprozessor das BoidModel wesentlich besser abschneidet als das AreaModel. Auf dem Tilera System jedoch das AreaModel besser geeignet ist.

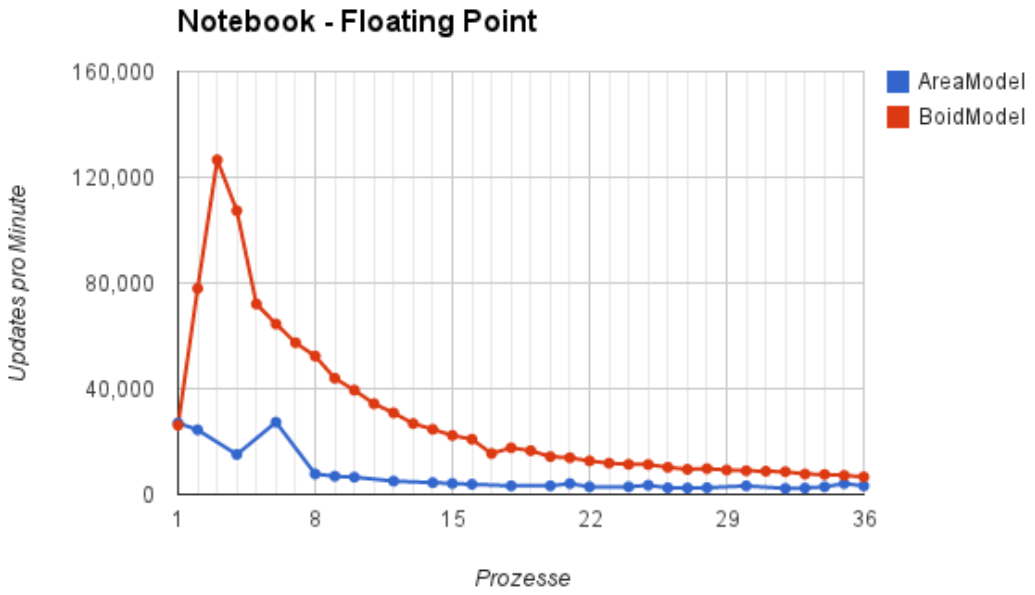

Abb. 5.3. Rot: BoidModel, Blau: AreaModel. Messwerte: [Mes]

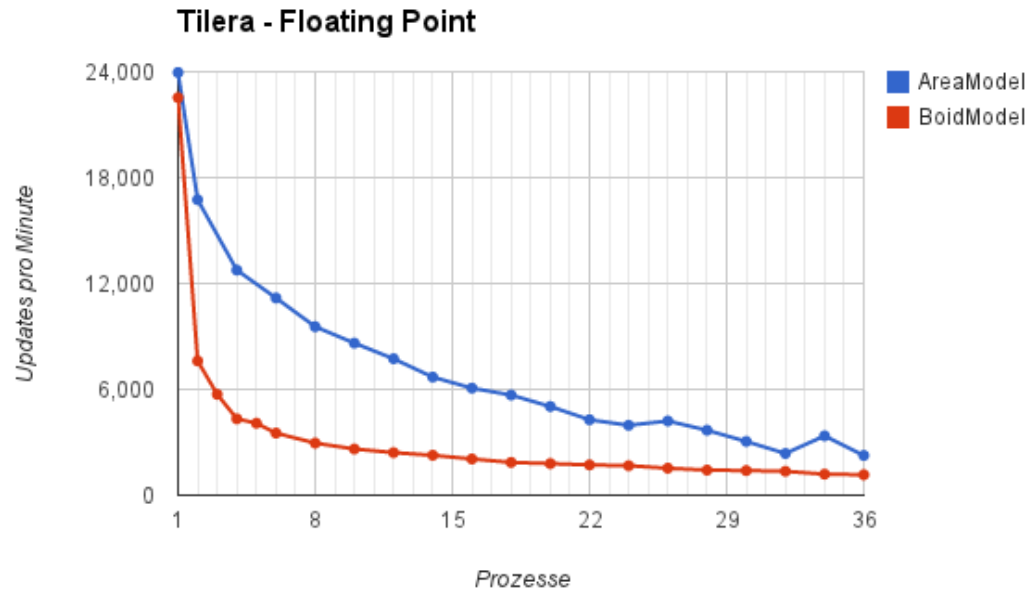

Abb. 5.4. Rot: BoidModel, Blau: AreaModel. Messwerte: [Mes]

5.3 AreaModel

Fließkommazahl gegen Festkommazahl

Auch In Abbildung 5.5 gilt, dass es keine Messwerte gibt, falls die Anzahl der Prozesse einer Primzahl entspricht.

Hier zeigt sich, dass die Festkommazahlen viel schwächer abfallen. Dies mag der Prozessorarchitektur zuzuschreiben sein. Dennoch erreichen die Spitzenwerte im AreaModel weder mit Fest- noch mit Fließkommazahl die Spitzenwerte im BoidModel.

Die Messung für Festkommazahlen im AreaModel konnte auf dem Tilera bisher nicht durchgeführt werden. Sie führt zu einem unerwarteten Laufzeitfehler.

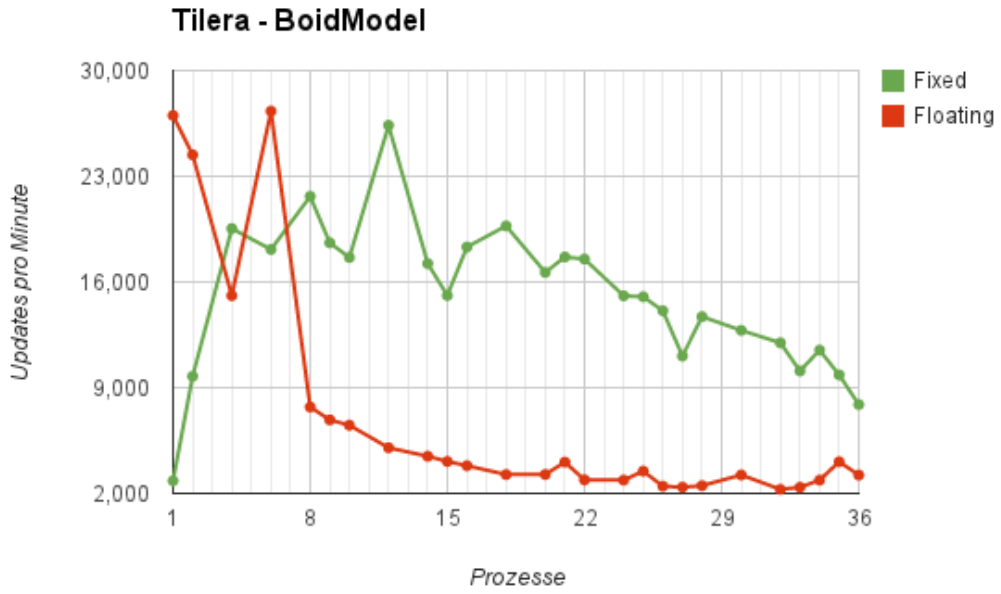

Abb. 5.5. Gün: Festkommazahl, Rot: Fließkommazahl. Messwerte: [Mes]

6

Zusammenfassung und Ausblick

Parallelisierung ist ein Schwieriges Thema, welches viel Zeit und Arbeit in Anspruch nimmt. Dabei übersteigt der Nutzen nicht immer die Kosten. Die Messungen haben unter anderem gezeigt, dass es nur wenig Sinn macht wesentlich mehr Prozesse zu starten, als das System Kerne hat. Das ist zum Teil auch gut nachvollziehbar, da auf einem Kern keine zwei Aufgaben echt-parallel ausgeführt werden können. Dies führt dazu, dass das Betriebssystem sehr viel Zusatzarbeit mit der Verwaltung von Prozessen hat.

Wie das Profiling gezeigt hat, können die Modelle noch sehr stark optimiert werden. Zum Beispiel ist es vorstellbar, dass man die Prozesskommunikation, für das AreaModel, Asynchron gestaltet. Dies ist mit MPI möglich und war auch einer der ersten Ansätze. Jedoch konnte dieser nicht erfolgreich abgeschlossen und somit verworfen werden.

Die Messungen auf dem Tilera haben insgesamt niedrigere Werte gezeigt, als die des Notebooks. Dies ist, bei geringer Prozessanzahl, zu erwarten gewesen, da das Notebook eine höhere Frequenz pro Kern besitzt. Jedoch ist der Verwaltungsaufwand bei vielen Prozessen zu groß, so dass keine höhere Leistung mit zunehmender Prozessanzahl gewonnen werden kann. Hier ist ein Update auf die neuste MPICH Version und eine Wiederholung der Tests sinnvoll.

Literaturverzeichnis

All. *Allinea*.
`http://www.allinea.com/`.

Kfi. *Boids Pseudocode*.
`http://www.kfish.org/boids/pseudocode.html`.

Mes. *Messwerte*.
http://goo.gl/1nTUax.

Wika. *Emergenz*.
`http://de.wikipedia.org/wiki/K%C3%BCnstliches_Leben`.

Wikb. *Künstliches Leben*.
`http://de.wikipedia.org/wiki/K%C3%BCnstliches_Leben`.

Wikc. *Message-Passing Interface*.
`http://de.wikipedia.org/wiki/Emergenz`.