Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Boids Algorithmus Parallelisierung mit Intel TBB

Oliver Biwer

Gruppe:

High-Performance Computing

Trier, 1. August 2017

Kurzfassung

Diese Arbeit befasst sich mit `Intel Threading Building Blocks (ITBB)` als Bibliothek zur Parallelisierung von C++ Programmen. Die Bibliothek wird anhand des Boids Algorithmus untersucht, einem Algorithmus zur Simulation von Tierherden (oder auch Vogelschwärmen). Als Grundlage werden die von `Craig Reynolds`[reynold:boids] definierten Regeln verwendet.

In der Arbeit wird der Algorithmus auf verschiedene Arten Parallelisiert und es zeigt sich, eine Parallelisierung des Algorithmus nicht zwingend sinnvoll ist. Vor allem muss abgewägt werden, wie wichtig die Präzision der Simulation ist und wie wichtig die Geschwindigkeit des Programms.

Inhaltsverzeichnis

1

Einleitung und Problemstellung

Mit fortschreitender Technik werden unsere Computer immer Leistungsfähiger. Für das Jahr 2017 wurden von AMD und Intel 8 CPUs mit mehr als 6 Kernen angekündigt[cpu:2017]. Ähnlich sieht der technische Fortschritt bei Grafikkarten aus. Diese werden mit Hilfe von CUDA oder OpenCL zur Parallelisierung von Programmen verwendet. Beispielsweise werden im Spiel `Drone Swarm` von `Stillalive Studios` ein Dronenschwarm mit 32.000 Raumschiffdronen vom Spieler gesteuert[droneswarm:golem] um feindliche Raumschiffe zu zerstören oder Ressourcen abzuarbeiten. Die Berechnung für die Bewegung der einzelnen Dronen findet dabei auf der GPU statt. Auch in anderen Spielen werden Algorithmen zum Schwarmverhalten zur Immersion verwendet.

Ziel dieser Arbeit ist es einen Algorithmus zum Schwarmverhalten zu Realisieren und zu Parallelisieren, verschiedene Methoden gegenüberzustellen und zu analysieren.

Der Quellcode des Projektes ist hier einzusehen:
https://github.com/soraphis/Boids-ITTB/tree/1.0.0

Die Messwerte finden sich hier: *https://goo.gl/QAWpd5*

2

Grundlagen

2.1 Boids

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.[Biwer]

Ein Boid $B = (\vec{p}, \vec{v})$ ist ein Tupel aus Positions- und Geschwindigkeitsvektor. Der Boid-Schwarm $\mathbb{B}$ die Menge aller Boids.

$B_{\vec{p}}$ ist die Positionskomponente des Boid B, $B_{\vec{v}}$ die Geschwindigkeitskomponente.

Regel 1: Cohesion

Diese Regel stellt den Zusammenhalt des Schwarms dar. Dazu wird für jeden Boid B die mittlere Position aller Boids im Umfeld (innerhalb der COHESION_RANGE) berechnet. Anschließend wird der Richtungsvektor von dem Boid B zu dieser mittleren Position ermittelt.

$$B_{\vec{v}_{coh}} = \frac{\sum_C C_{\vec{p}}}{\#C} - B_{\vec{p}} \qquad mit\ C \in \{c | c \in \mathbb{B} \wedge |c_{\vec{p}} - B_{\vec{p}}| < COHESION\_RANGE\}$$

Abbildung 2.1 verdeutlicht dies.

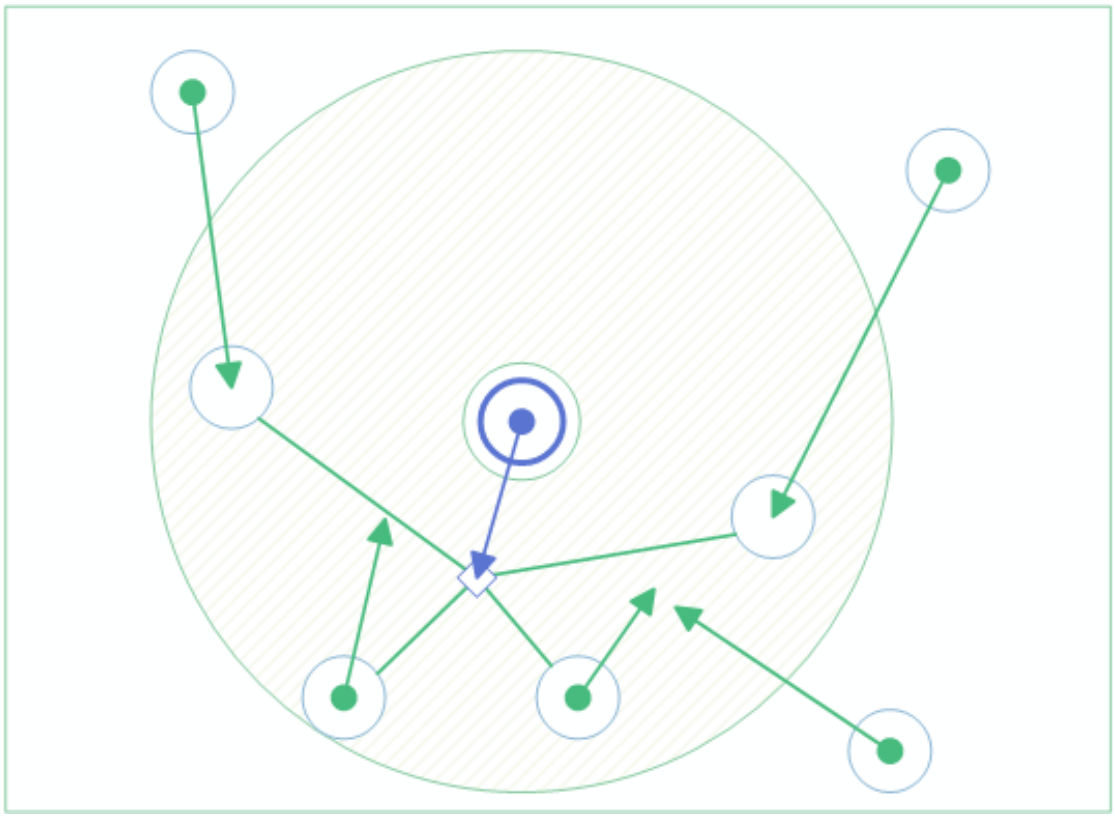

Abb. 2.1. Die grün-straffierte Fläche ist der Sichtbereich des blauen Boid. Die Pfeile entsprechen den Geschwindigkeitsvektoren nach Anwendung der Cohesion-rule. Die Pfeile zielen auf den Mittelpunkt des für ihn Sichtbaren Schwarms, wie am blauen Pfeil verdeutlicht.

Regel 2: Separation

Diese Regel verhindert, das Boids miteinander Kollidieren. Dazu wird eine Ausweichrichtung berechnet (Siehe 2.2).

$$B_{\vec{v}_{sep}} = \frac{\sum_C -(C_{\vec{p}} - B_{\vec{p}})}{\#C} \qquad mit\ C \in \{c | c \in \mathbb{B} \backslash B\ \wedge |c_{\vec{p}} - B_{\vec{p}}| < SEPARATION\_RANGE\}$$

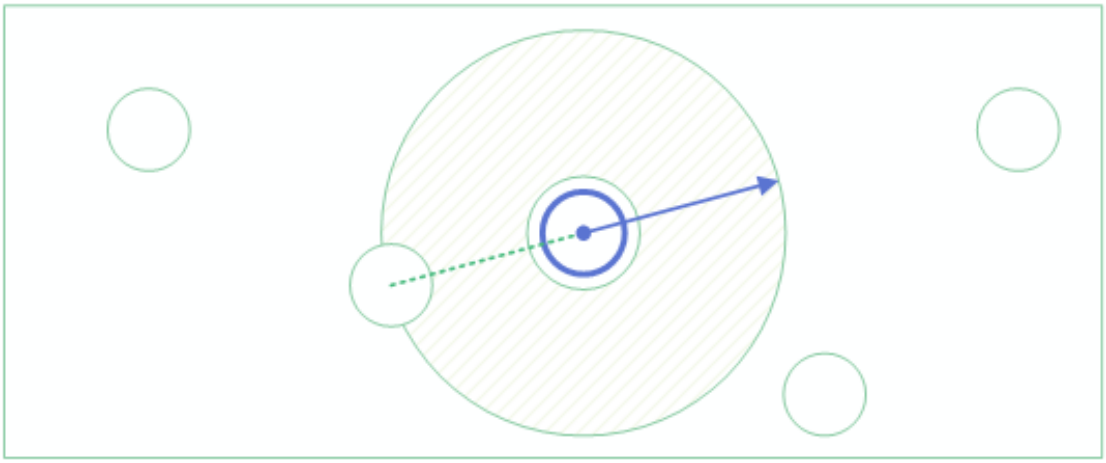

Abb. 2.2. Die grün-straffierte Fläche ist der Mindestabstand, der zwischen zwei Boids angestrebt wird. Der blaue Pfeil ist der resultierende Geschwindigkeitsvektor für den blauen Boid.

Regel 3: Alignment

Die Alignment Regel sorgt dafür, das der Schwarm eine gemeinsame Richtung anstrebt. Dazu wird für jeden Boid die mittlere Geschwindigkeit des Umfelds ermittelt (vlg. 2.3).

$$B_{\vec{v}_{alg}} = \frac{\sum_C C_{\vec{v}}}{\#C} \qquad mit\ C \in \{c | c \in \mathbb{B} \wedge |c_{\vec{p}} - B_{\vec{p}}| < ALIGNMENT\_RANGE\}$$

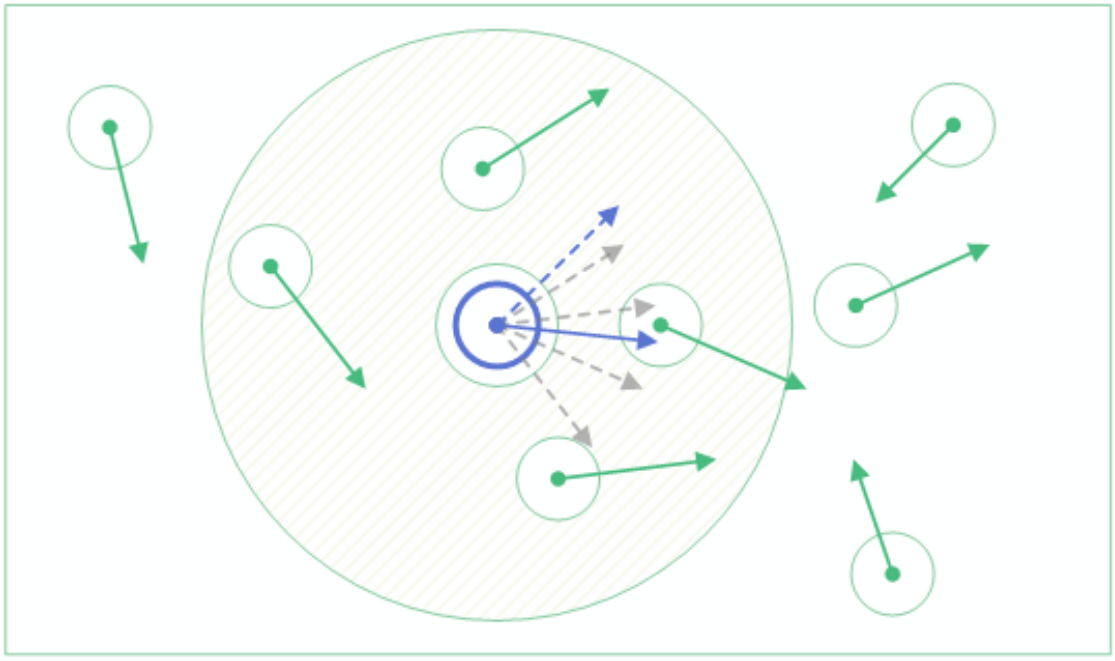

Abb. 2.3. Die grauen Pfeile sind Projektionen der grünen Geschwindigkeit Vektoren. Der blaue, gestrichelte Pfeil entspricht der ursprünglichen Geschwindigkeit, der blaue Pfeil mit der durchgezogenen Linie ist die Geschwindigkeit nach der Angleichung.

Nachdem die Regeln ausgewertet wurden, wird für jeden Boid ein neuer Geschwindigkeitsvektor ermittelt.

$$B_{\vec{v}} = B_{\vec{v}} + B_{\vec{v}_{alg}} + B_{\vec{v}_{sep}} + B_{\vec{v}_{coh}}$$

Optional können die einzelnen Regeln gewichtet und der resultierende Vektor $B_{\vec{v}}$ in seinem Betrag beschränkt werden.

2.2 Intel TBB

Intel Threading Building Blocks (ITBB) ist eine C++ Template Bibliothek zur Daten-Parallelisierung (SIMD)[intel:tbb]. Dazu bietet die Bibliothek eine Abstraktionsebene auf der sich der Programmierer nicht um das erstellen von Threads kümmern muss, sondern Abschnitte des Programmcodes als parallelisierbar kennzeichnet. Dieses spezifizieren logischer Parallelisierbarkeit gelingt mithilfe von Bausteinen die ITTB zur Verfügung stellt.

3

Seeqequenzielles Programm

Als Grundlage wird der Boids Algorithmus erst einmal Sequenziell implementiert. Das Ziel der parallelen Implementierungen sollte es also sein eine höhere Performance zu haben als das Sequenzielle Programm.

Logisch sollten sich die parallelen Implementierungen ebenfalls an dieses Sequenzielle Programm halten (wir werden später sehen, dass dies nicht der Fall ist). Das heißt, dass Konstanten die hier verwendet werden, in den anderen Versionen die gleichen Werte enthalten. Das heißt auch, dass die Boids in den weiteren Implementierungen allen Regeln folgen, denen sie in dieser Implementierung folgen.

3.1 Konstanten

Die Konstanten, welche von allen Programmen verwendet werden finden sich unter `src/core/constants.h`.

Die wichtigsten Werte in den Konstanten sind: Die Schwarmgröße ist auf 400 fest gelegt. Die Reichweite der Alignment-Regel ist 10, die der Cohesion-Regel 7 und die der Separation-Regel 1.

3.2 Programmschleife

Das Hauptprogramm ist unter `src/main.cpp` zu finden. Die Main-Loop ist größten Teils für alle Programme gleich und findet sich unter `src/core/update_loop.cpp`

Zunächst wird der Schwarm initialisiert, dazu generieren wir für jeden Boid eine Zufallsposition. Der Startwert der Zufallszahlen wird jedoch auf eine fixe Zahl gesetzt, dadurch wird der Schwarm jedes mal gleich generiert. Das heißt im Idealfall bewegen sich die Boids der parallelen Programm exakt wie in diesem sequenziellen Programm.

Im weiteren werden jetzt alle Boids geupdated, d.h. die neuen Geschwindigkeitsvektoren und Positionen werden ermittelt. Danach werden die Boids in einem Fenster dargestellt.

3.3 Rendering

Abbildung 3.1 zeigt den Schwarm. dargestellt werden die Boids als farbige Pixel, dabei steht ihre Farbe für ihre Position im Speicher. Außerdem wird angezeigt wie viel Millisekunden der letzte update Vorgang gedauert hat.

3.4 Schwarm Update

Wir implementieren eine Art *Doppelbuffer*. Wir Updaten jeden Boid des Schwarms mit Hilfe der Boid-Regeln, Speichern das Ergebnis dieses Updates jedoch als neuen Boid in einen zweiten Schwarm. Am Ende Der `swarm_update` Methode wird der eigentliche Schwarm durch diesen zweiten Schwarm ersetzt.

Dies ist notwendig, damit Boids die später geupdated werden nicht bereits mit den aktualisierten werten der anderen Boids rechnen. Der Fehler der dabei Auftritt ist zwar minimal, addiert sich jedoch über von Update zu Update auf.

Zusätzlich zu den 3 beschriebenen Boid Regeln, erhalten die Boids noch eine Regel, die verhindert dass Sie zu weit weg 'fliegen'.

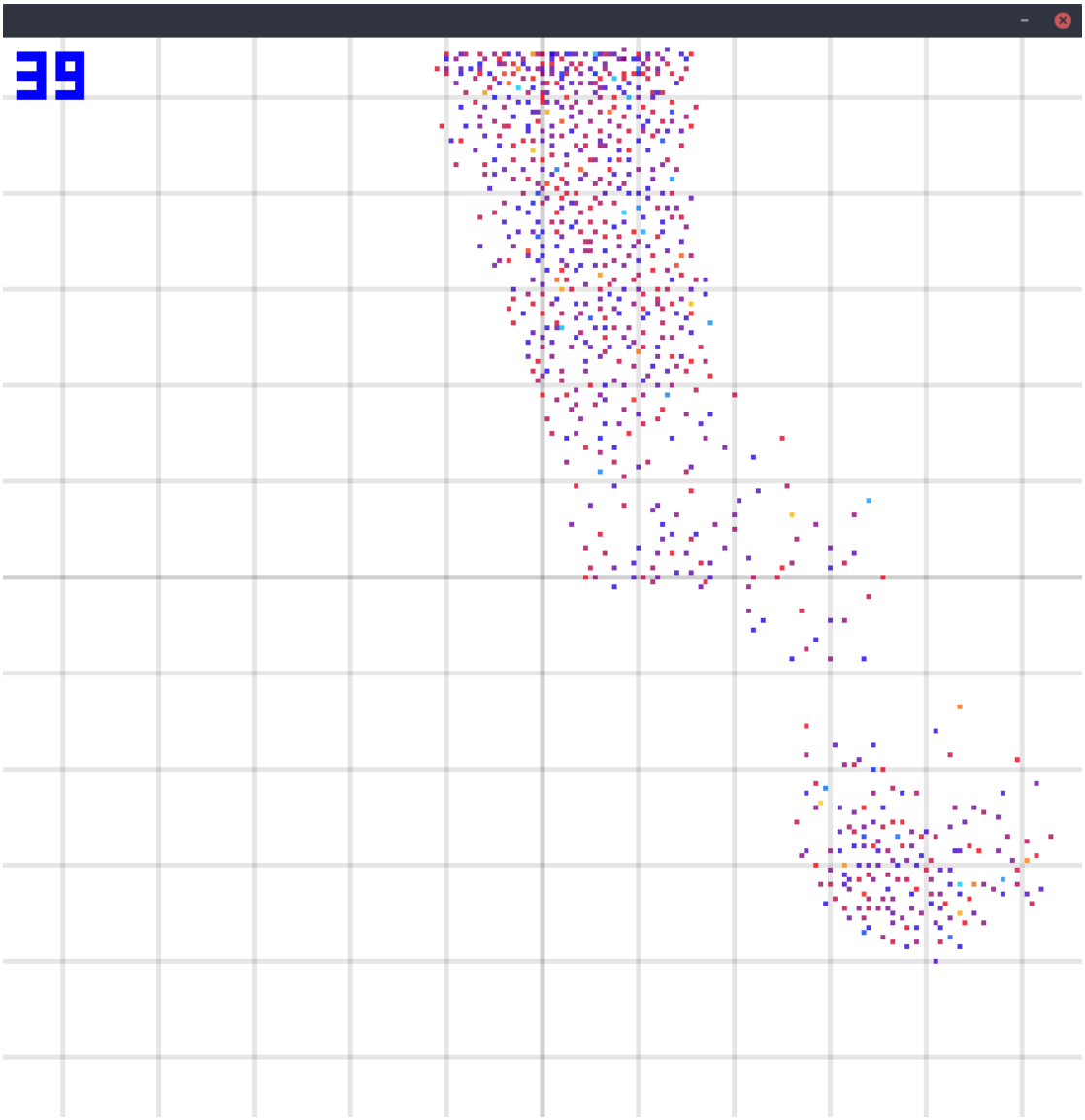

Abb. 3.1.

3.5 Benchmark

Getestet wurde auf einem Notebook mit Intel i5 4200U Prozessor mit 8GB RAM. Ein Testlauf dauerte 25 Sekunden und zwischen den Tests wurden Pausen von 25 Sekunden eingehalten.

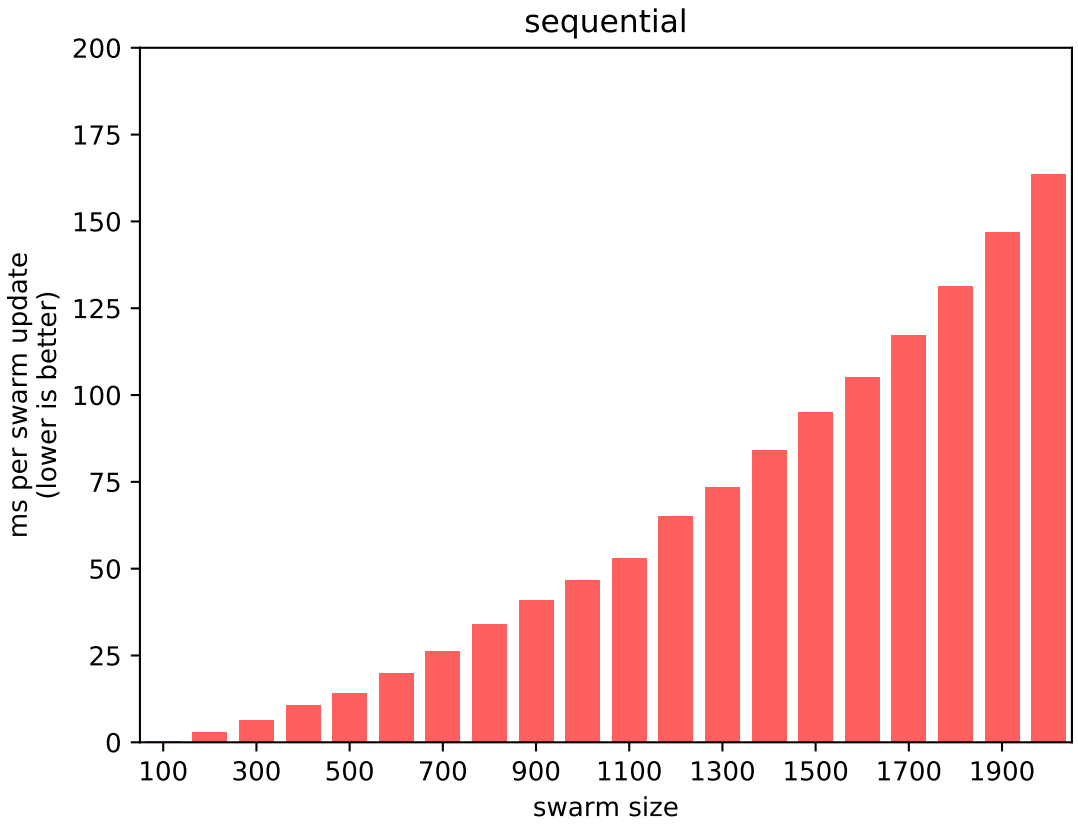
sequential
ms per swarm update
(lower is better)
200
175
150
125
100
75
50
25
0
100
300
500
700
900
1100
1300
1500
1700
1900
swarm size

Abb. 3.2.

3.6 Profiling

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
manhattan_distance	boids	4.103s
boid_update	boids	3.281s
Vector3D::x	boids	2.821s
std::abs	boids	2.295s
is_inrange	boids	2.003s
[Others]		10.130s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

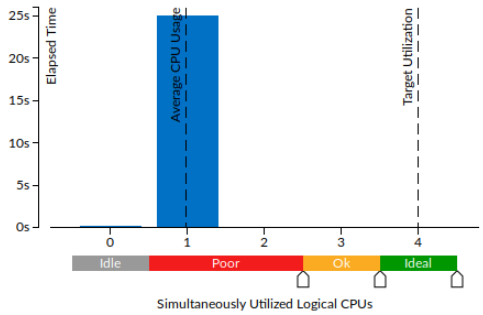

Abb. 3.3. Profiling Ergebnis mit Intel Vtune Amplifier

Das Profiling zeigt, dass die meiste Zeit mit dem Updaten der Boids verbracht wurde, genauer: Mit dem berechnen des Abstandes zwischen zwei Boids.

4

Boid Parallelisierung

Das Hauptprogramm ist unter `src/main_tbb1.cpp` zu finden.

Der erste Gedanke, der bei der Parallelisierung des Boid Algorithmus kommt, ist das Aufteilen der Boid Updates. Für jeden Boid müssen alle Boid-Regeln aktualisiert werden. Beim Auswerten der Regeln verbringt das Programm die meiste Zeit.

In der Methode `swarm_update` wird nun die `for-each` schleife durch die `parallel_for_each` Methode der Intel TBB Bibliothek ersetzt.

Nun kann es jedoch passieren, dass zwei (oder mehr) Threads gleichzeitig in den *Doppelbuffer Schwarm* (siehe 3.4) Schreiben wollen und das Programm stürzt ab. Dies kann durch `Locks` oder `Mutexes` gelöst werden, verschenkt jedoch einen Großteil der durch Parallelisierung gewonnenen Laufzeit und im Worst-Case läuft unser Programm langsamer als zuvor. Daher wird in dieser Implementierung einfach auf den *Doppelbuffer Schwarm* verzichtet, die Boids verden direkt aktualisiert und der Fehler, der erwähnt wurde, wird in Kauf genommen.

Das Problem lässt sich auch anders Lösen, beispielsweise durch eine Kopie des Schwarms bei dem die Indizes mit denen des ersten Schwarms übereinstimmen und das Update an die Entsprechende Indexposition geschrieben wird.

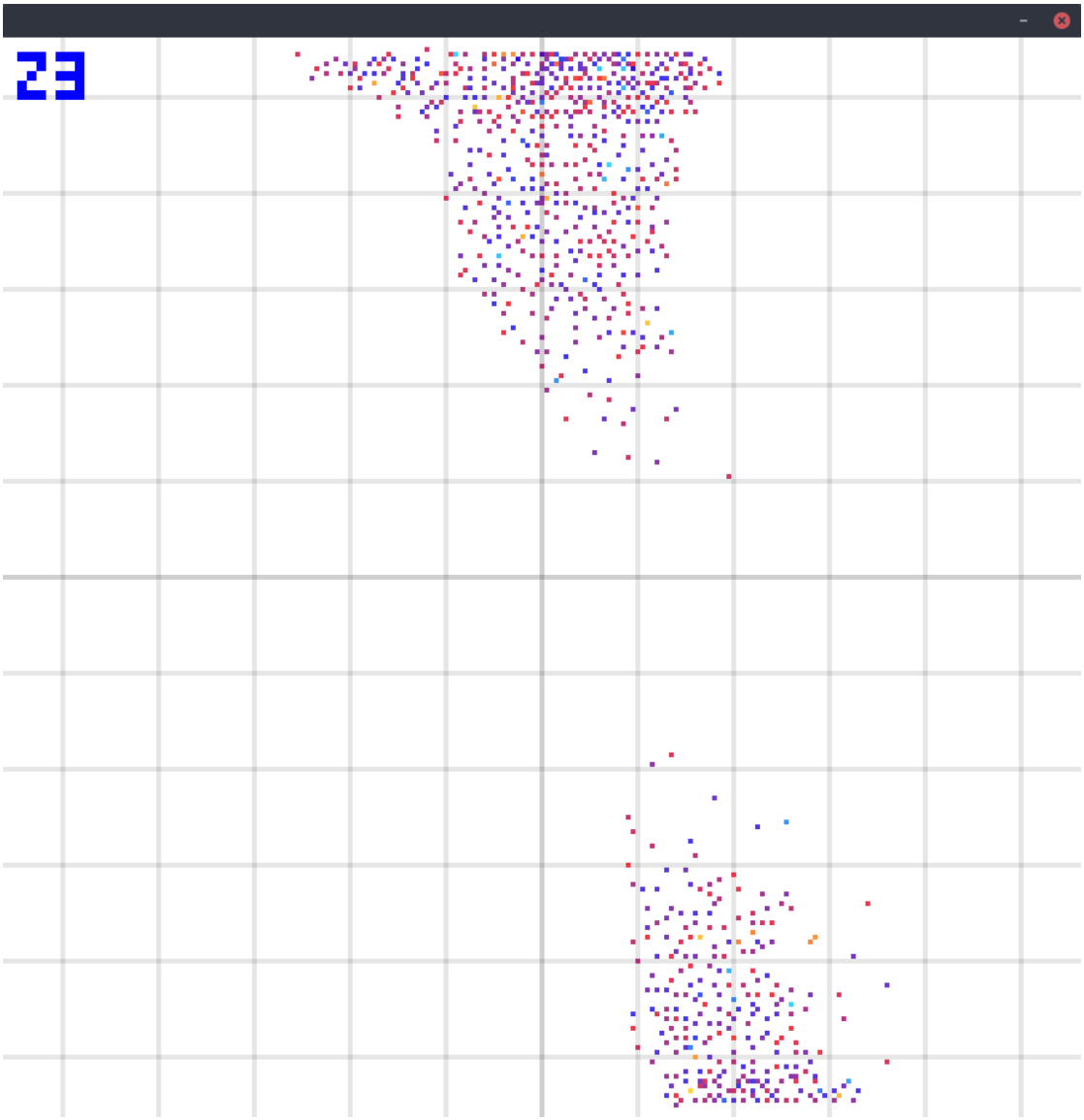

Abb. 4.1.

4.1 Benchmark

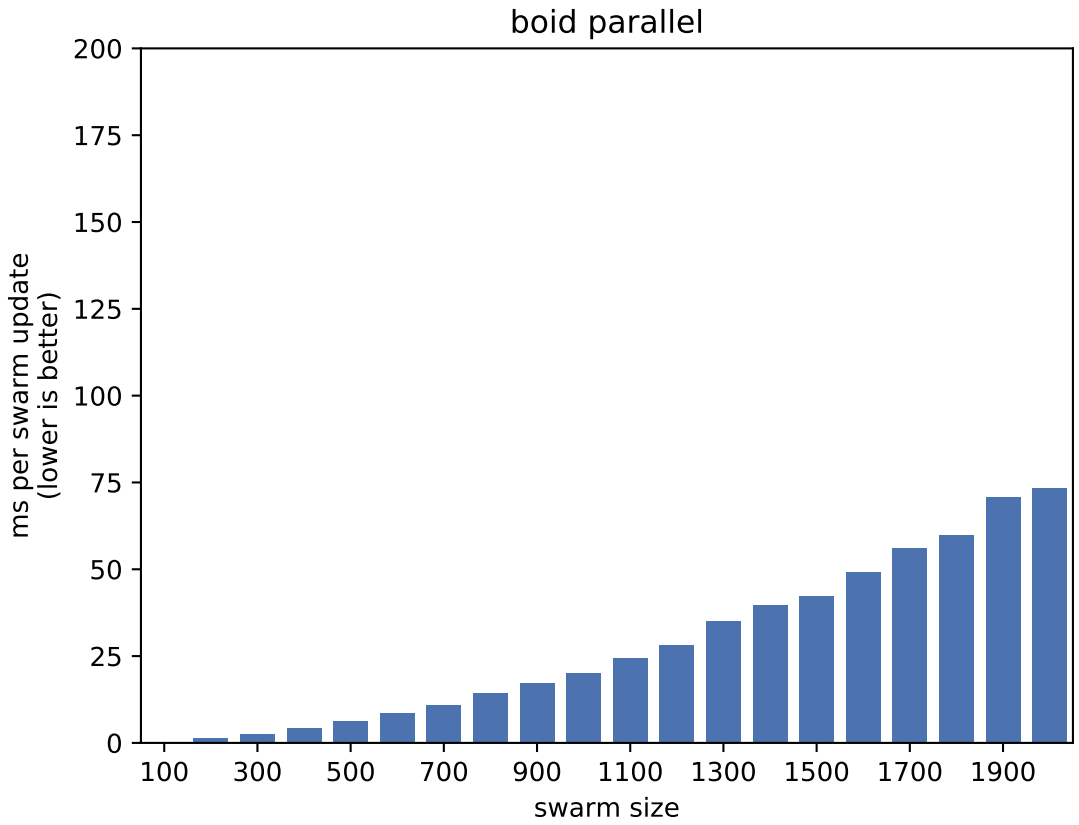

Abb. 4.2.

5

Spatial Hashing

Das Hauptprogramm ist unter `src/main_tbb2.cpp` zu finden.

Wie bereits erwähnt, verbringt das Sequenzielle Programm die meiste Zeit damit, die Regeln der Boids auszuwerten. Dabei verbringt es die meiste Zeit damit, Distanzen zwischen Boids auszurechnen. Das Liegt daran, das jeder Boid, jeden anderen Boid abgleicht ob dieser Beispielsweise innerhalb der Cohesion-Range ist oder nicht. Kann man nun eine Vorauswahl an Boids machen müsste nicht mehr der gesamte Schwarm durchlaufen werden.

Eine solche Vorauswahl kann man dadurch erreichen, das man den Raum, in dem sich die Boids bewegen, in Zellen unterteilt. So müssen beispielsweise nur noch Boids der Nachbarzellen geprüft werden. Und die verschiedenen Zellen können parallel abgearbeitet werden.

Der Boid Algorithmus simuliert jedoch einen Schwarm und Schwärme neigen dazu zu 'Clustern'. Das heißt, fast alle Boids befinden sich ungefähr am selben Ort. Teilt man nun den Raum gleichmäßig in Zellen auf, führt das dazu, das ein Großteil der Zellen leer ist. Eine Möglichkeit dies zu Lösen ist Bestandteil des `Barnes-Hut Algorithmus`[Josh Barnes][barnes-hut:demo] und zwar wird der Raum in Quadtrees dynamischer Tiefe unterteilt.

Neben Quad-/Octrees gibt es noch die Möglichkeit `Spatial Hashing` zu nutzen[spatial-hash:compare]. Dazu errechnet man aus den Boid Koordinaten einen Hashwert der als Schlüssel in einer `map` (z.B. `unordered_map` der C++ Standard Bibliothek) verwendet wird. Hinter jedem Schlüssel in dieser `map` wird eine Liste an Boids gespeichert. Diese Listen können Parallel abgearbeitet werden.

Die Implementierung wirft jedoch einige Schwierigkeiten auf. Verlässt ein Boid seine Zelle, muss er der Nachbarzelle zugewiesen werden, wird diese jedoch von einem anderen Thread bearbeitet stürzt im schlimmsten Fall das Programm ab. Um dies zu lösen können wieder `Mutex`es verwendet werden. Eine weitere Möglichkeit bietet ITTB jedoch mit `concurrent_unordered_map` und `concurrent_vector`. Zwar ist

dadurch das Problem gemeinsamen Speicherzugriffs gelöst[1] jedoch geht ein Teil der Parallelität dadurch verloren.

Dies zeigt sich deutlich in den Benchmarks.

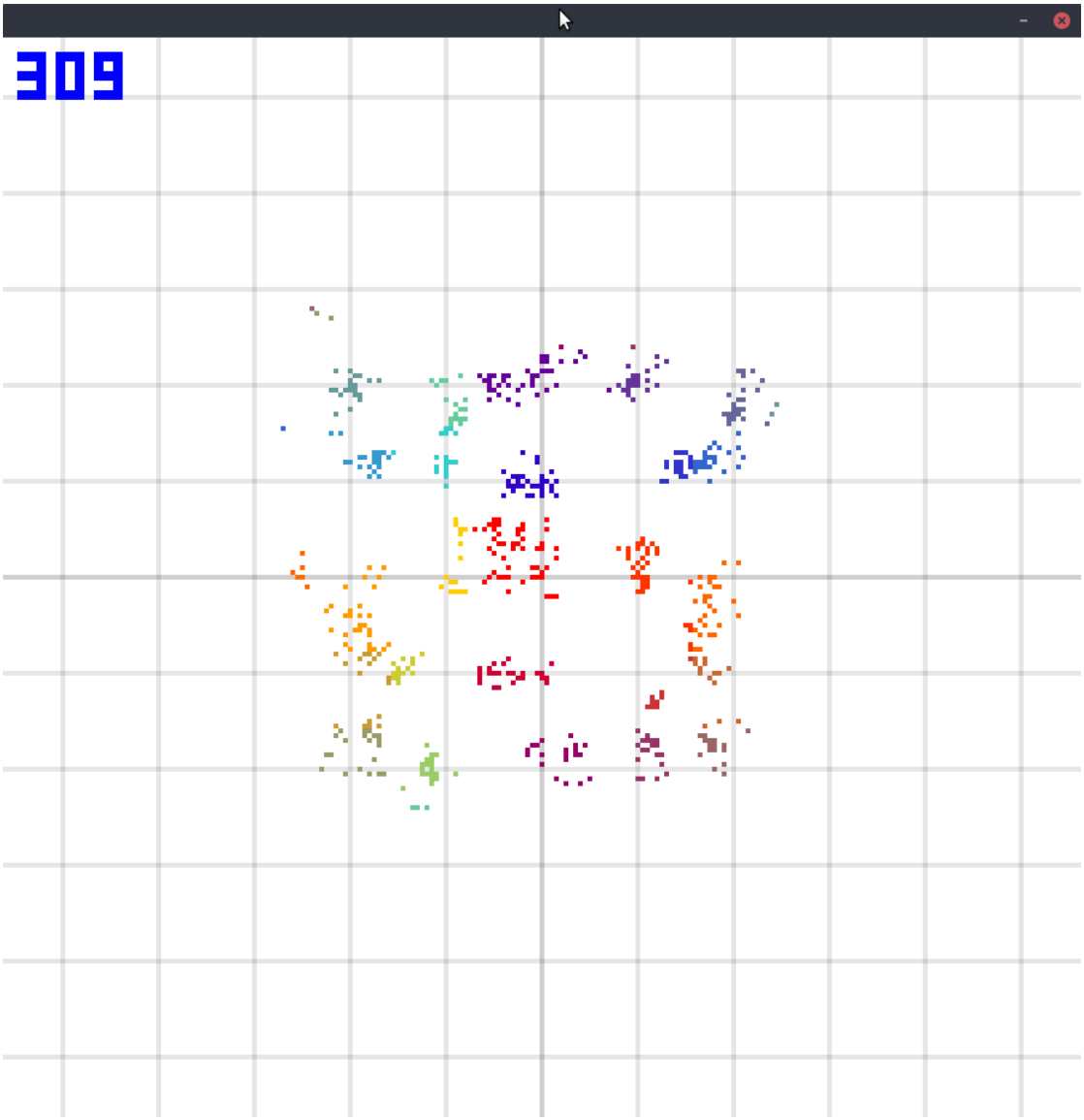

Abb. 5.1. Boids neigen dazu, in viele kleine Schwärme zu Clustern. Um dies zu verhindern müssten mehr Zellen bei der Berechnung der Boid Regeln ausgewertet werden. Die Farbkodierung zeigt, dass Boids die nah beieinander liegen, in den gleichen Zellen landen.

[1] *Anm.: Im Test mit 100 Boids kam es dennoch immer wieder zu Problemen*

5.1 Benchmark

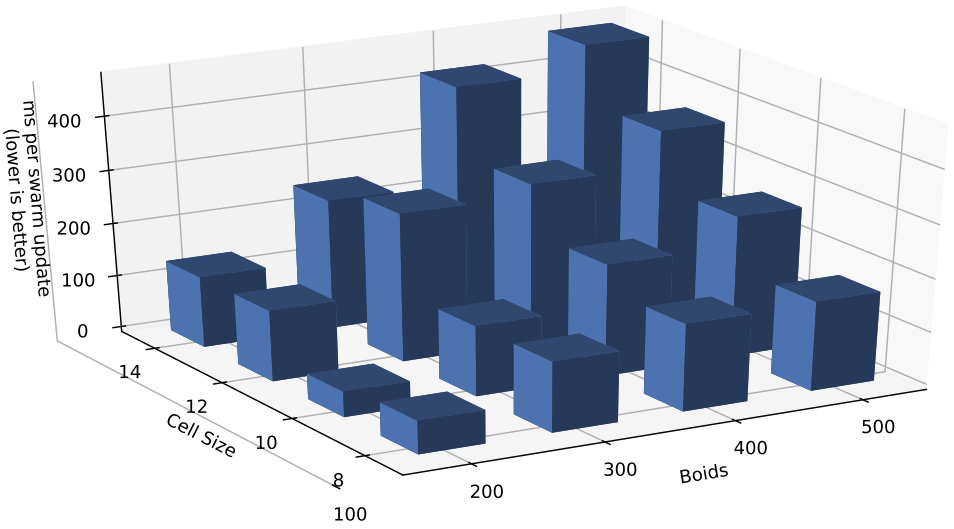

Abb. 5.2. Vergleich wie verschiedene Zellengrößen Einfluss auf die Laufzeit haben

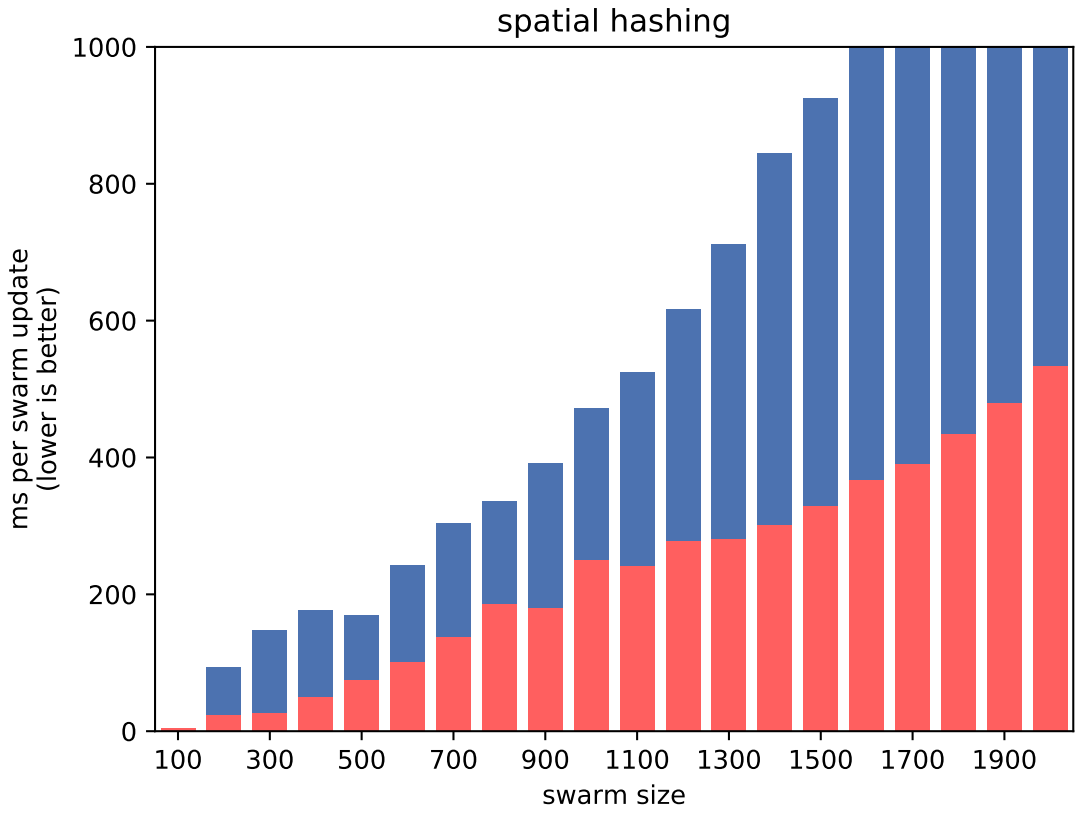

Abb. 5.3. Laufzeit bei Verwendung von Spatial Hashing. Die Roten Balken zeigen die Laufzeit bei Sequenzieller Verwendung dieser Methode. Dies lässt darauf schließen, dass der overhead bei Verwendung der `concurrent` Objekte zu groß ist. Die Zellengröße wurde auf 8 gesetzt.

Anm.: Die y-Skala weicht von den anderen Benchmarks ab.

6

Indizieren durch Z-Curve

Das Hauptprogramm ist unter `src/main_tbb4.cpp` zu finden.

Der Grundgedanke, eine Vorauswahl treffen zu können, um die Anzahl der Paarweisen vergleiche zwischen den Boids zu minimieren, bleibt weiterhin bestehen, doch sieht die Umsetzung anders aus. Im vorherigen Kapitel schränkten die Datenstrukturen die Parallelisierbarkeit ein. Aus diesem Grund befinden sich in diesem Kapitel alle Boids wieder in einer Liste. Diese Liste gilt es nun, so zu sortieren, dass durch Indexbereiche Boids so ausgewählt werden können dass alle Boids zwischen diesen zwei Indices sich so nah beieinander befinden, dass sie für die Berechnung der Boid-Regeln relevant sind.

Die Z-Kurve[zcurve:wik] ist eine Raumfüllende Kurve (wie die Hilbert-Kurve). Mit ihrer Hilfe können wir die Raumposition der Boids auf eine Indexposition der Z-Kurve abbilden.

Nachteil dieser Methode ist es, dass wir den Boid Schwarm nach jedem Update sortieren müssen. Hier eignen sich Sortierverfahren wie Insertion-Sort, da die Boids je Frame nur kleine Änderungen in ihrer Position haben ist die Liste fast perfekt vorsortiert. In diesem Beispiel wurde zum Sortieren die Intel TBB Methode `tbb::parallel_sort` verwendet. Um weiter bessere Laufzeiten zu erhalten wurde die Sortierung nur zu jedem 10. Update durchgeführt, dadurch verringert sich jedoch die Präzision der Simulation.

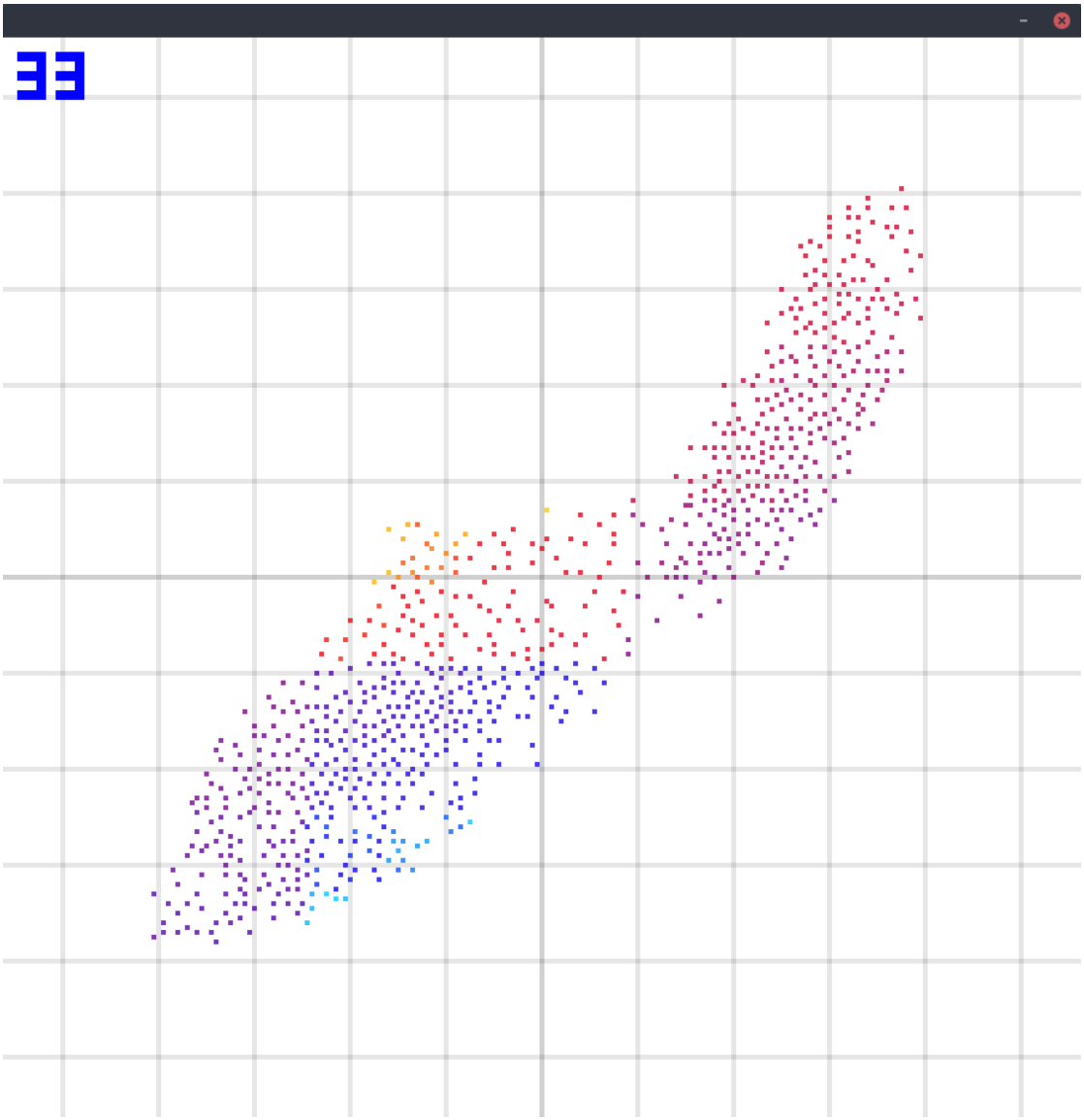

Abb. 6.1. Die Farbkodierung zeigt deutlich das die Indexpositionen der Boids mit ihrer geografischen Position zusammenhängt - also das die Z-Kurven Indizierung stimmt.

6.1 Benchmark

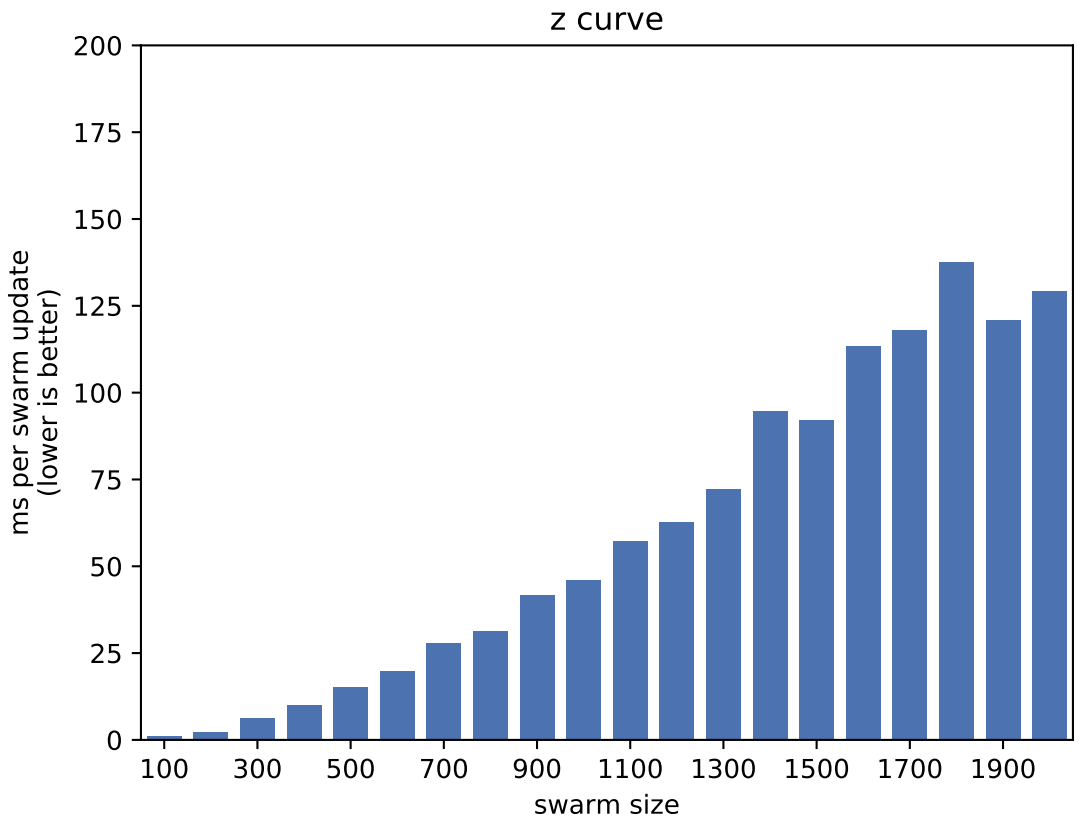

Abb. 6.2. Die Benchmarks zeigen ungefähr die selbe Laufzeit wie bei der sequenziellen Methode.

7

Zusammenfassung und Ausblick

Wie die Benchmarks zeigen, erreicht die erste Variante, in der der Boid Schwarm parallel geupdated wird, die besten Ergebnisse. Dazu kommt, das diese Variante die höchste Präzision aller parallelen Varianten hat. Wie in Kapitel 4 beschrieben lässt sich diese Präzision noch weiter steigern. Allgemein zeigt sich, dass die komplexere Parallelisierungen kaum lohnenswert sind.

Die Z-Kurven Variante könnte durch eine parallele Insertion-Sort Implementierung profitieren. Der nutzen der Z-Kurven Indizierung zeigt sich stärker je verteilter die Boids sind.

Literaturverzeichnis

[droneswarm:golem] : *32.000 Drohnen unter galaktischer Kontrolle.* – URL `https://www.golem.de/news/drone-swarm-32-000-drohnen-unter-galaktischer-kontrolle-1608-122788.html`

[cpu:2017] : *CPUs für 2017.* – URL `http://www.cpu-world.com/Releases/Desktop_CPU_releases_(2017).html`

[reynold:boids] : *Craig Reynold: Boids.* – URL `http://www.red3d.com/cwr/boids`

[barnes-hut:demo] : *Live Quadtree with 10000 Particles.* – URL `https://www.youtube.com/watch?v=ouO6wmKqycc`

[spatial-hash:compare] : *Quadtree vs Spatial Hashing - a Visualization.* – URL `http://zufallsgenerator.github.io/2014/01/26/visually-comparing-algorithms/`

[intel:tbb] : *threading building blocks website.* – URL `https://www.threadingbuildingblocks.org`

[zcurve:wik] : *Z-Kurve.* – URL `https://de.wikipedia.org/wiki/Z-Kurve`

[Biwer] Biwer, Oliver: *MPI Parallelisierung am Beispiel des Boids Algorithmus*

[Josh Barnes] Josh Barnes, Piet H.: *A hierarchical O(N log N) force-calculation algorithm*

Sachverzeichnis