Bachelorthesis

im Studiengang
Medieninformatik

Evaluation verschiedener Techniken zur Parallelisierung von Programmen anhand einer Beispielanwendung

vorgelegt von
Benjamin Binder
Matrikelnummer: 24612

an der
Hochschule der Medien Stuttgart
Fakultät Druck und Medien

am
31.8.2015

zur Erlangung des akademischen Grades eines
Bachelor of Science (B.Sc)

Erstprüfer: Prof. Dr. Jens-Uwe Hahn
Zweitprüfer: M.Sc. Patrick Bader

Hiermit versichere ich, Benjamin Binder, an Eides statt, dass ich die vorliegende Bachelorthesis mit dem Titel Evaluation verschiedener Techniken zur Parallelisierung von Programmen anhand einer Beispielanwendung selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.
Ich habe die Bedeutung der eidesstattlichen Versicherung und prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO der Hochschule der Medien Stuttgart) sowie die strafrechtlichen Folgen (§ 156 StBG) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

_____

Stuttgart, 31.8.2015

Danksagung

Ich möchte mich bei allen bedanken, die mich beim Schreiben dieser Bachelorthesis unterstützt haben.

Danke an Jessi für ihre unendliche Geduld beim Erklären von Physik, sowie an Martin und Sabine für ihre vielen hilfreichen und amüsanten Korrekturen!

Des Weiteren gilt mein Dank Prof. Dr. Jens-Uwe Hahn und Patrick Bader für die Betreuung dieser Thesis.

Abstract

Moderne Programme bewältigen immer komplexere und leistungsfordernde Aufgaben. Mit diesem Anstieg geht jedoch ein höherer Bedarf an Hardware-Ressourcen einher, insbesondere an höheren Prozessorkapazitäten. Diesem Trend wurde mit einer konstanten Erhöhung der Taktraten von Prozessoren begegnet. Doch seit 2005 wurde dieser Trend aufgrund von physikalischen Grenzen gebremst. Stattdessen installieren Prozessorhersteller nun mehrere Prozessorkerne mit geringerer Taktrate auf einem Prozessor. Dies führt auch zu neuen Programmiertechniken, die Programme auf mehreren Prozessorkernen verteilen. Sie stellen einen sicheren Datenzugriff, deterministische Ausführung und Leistungsverbesserungen sicher. Ursprünglich mussten Programmierer diese Techniken manuell programmieren, heute existieren Technologien, die eine solche Verwaltung automatisch durchführen.

In dieser Thesis werden verschiedene High-Level Programmiertechniken anhand einer Beispielanwendung hinsichtlich ihrer Leistung, Ressourcenverwaltung und Bedienbarkeit verglichen. Die Beispielanwendung soll eine tatsächlich einsetzbare Anwendung repräsentieren, die grundlegende Probleme, wie voneinander unabhängige und abhängige Berechnungsschritte aufweist, weshalb eine Physiksimulation gewählt wurde. Die Parallelisierung wurde mit Goroutinen, Java Parallel Streams, Thread Pools und C++ async-Funktionen in ihrer jeweiligen Programmiersprache realisiert.

Um die verschiedenen Parallelisierungstechniken zu vergleichen, wurden mehrere Merkmale der parallelen Implementierungen gemessen und mit einer sequentiellen Referenzimplementierung verglichen. Um die Leistung der Techniken zu messen, wurden die Ausführungszeiten der verschieden Simulationen gemessen und analysiert. Die Ressourcenverwaltung wurde anhand der Prozessorauslastung der verschiedenen Implementierungen verglichen. Um die Bedienbarkeit der verschiedenen Parallelisierungstechniken gegenüberzustellen, wurde die Anzahl der Quelltextzeilen ermittelt und in Relation gesetzt.

Die Analyse dieser Daten zeigt die Unterschiede der Parallelisierungstechniken. Während die Implementierung unter Nutzung von Java Parallel Streams hohe Prozessorauslastung, und, verglichen mit den anderen Techniken, einen hohen Beschleunigungsfaktor sowie geringe Komplexität aufweist, kann die Implementierung mit Hilfe von C++ async-Funktion nicht mehrere Prozessorkerne auslasten und damit nicht die Vorteile von Parallelisierung ausnutzen. Die hohe Komplexität der Implementierung mit Goroutinen zahlt sich durch vergleichsweise geringe Ausführungszeiten trotz niedriger Prozessorauslastung aus.

Modern applications manage more complex and performance-demanding tasks than ever. But with more performance, applications also consume more hardware resources, especially CPU capacity. This demand was handled by constantly increasing the clock rate of CPUs. But since 2005, this trend was stopped due to physical limitations. Instead of increasing the performance of one CPU core, manufacturers put several cores with less clock frequency on one processor. This led to a new way of programming. Spreading a program across several CPU cores requires special programming techniques, ensuring safe data access and deterministic execution of programs, as well as performance gains. First, this techniques had to be handled manually by the programmer, but now, method exists, that manage the overhead of parallel programming.

In this Thesis, an example applications is used to evaluate different high level parallel programming techniques according to their performance, their hardware management and their usability. The example application should represent a real world application with common problems like dependent and independent calculations. Therefore, a physic simulation was chosen. It was parallelized using Goroutines, Java Parallel Streams, Thread Pools and C++ async-functions in the respective programming language.

To evaluate the different parallelization techniques, several characteristics of the implementations using them were measured and compared to a sequential reference implementation. To measure the performance, the execution times of the simulations were captured and interpreted. To measure the hardware management, the CPU utilization was analyzed. The complexity of the techniques was determined by comparing the number of lines of code of the implementations using the respective techniques.

The analysis of these data showed differences between the parallelization techniques. While the implementation using Java Parallel Streams showed high CPU utilization and, in comparison to the other implementations, high speedup as well as low complexity, the implementation using C++ async-functions couldn't use the benefit of using multiple CPU cores at all. The high complexity of the implementation using Goroutines leads to relatively low execution times, despite its lower CPU utilization.

Inhaltsverzeichnis

Abstract ..... **iii**

1 Einleitung **1**
1.1 Zielsetzung 1
1.2 Methodik 2
1.3 Erwartungen 3
1.4 Aufbau der Arbeit 3

2 Verwandte Arbeiten **5**

3 Theoretische Grundlagen **7**
3.1 Einkern- und Mehrkernprozessoren 7
3.1.1 Scheduling 8
3.1.2 Prozesse und Threads 8
3.1.3 Multithreading 9
3.1.4 Shared Memory 9
3.1.5 Asynchrone Datenzugriffe auf gemeinsame Ressourcen 9
3.1.6 Probleme bei Nebenläufigkeit 10
3.1.7 Synchronisierung von Datenzugriffen 12
3.1.8 Probleme der Synchronisierung 13
3.2 Eingesetzte Parallelisierungstechniken 14
3.2.1 Thread Pool 15
3.2.2 C++ 11 – async 16
3.2.3 Java Parallel Streams 18
3.2.4 Goroutinen 19
3.2.5 Nomenklatur 21

4 Implementierung der Testanwendung **23**
4.1 Programmstruktur 23
4.1.1 Emitter 24
4.1.2 Physiklogik 25
4.2 Parallelisierung 27
4.2.1 Aktualisierung der Kreise 28
4.2.2 Kollisionserkennung 32

5 Planung und Durchführung der Datenerhebung **37**
5.1 Vergleichsmerkmale 37
5.1.1 Merkmale zum Leistungsvergleich 37
5.1.2 Merkmale zum Vergleich des Bedienaufwandes 38
5.2 Eingesetzte Messtechniken 39
5.2.1 Techniken zur Zeitmessungen 40
5.2.2 Technik zur Messung der Prozessorbelastung 40
5.2.3 Wiederholbarkeit und Vergleichbarkeit der Messungen 40
5.3 Messumgebung und Simulationsparameter 42
5.3.1 Ausgangswerte der Simulation 42

6 Präsentation und Diskussion der Messergebnisse **45**

6.1 Ressourcennutzung....45
6.1.1 Thread Pool....45
6.1.2 C++ 11 – async....46
6.1.3 Java Parallel Streams....47
6.1.4 Goroutinen....49
6.1.5 Vergleich der Parallelisierungstechniken....51
6.2 Ausführungszeiten....53
6.2.1 Thread Pool....54
6.2.2 C++ 11 – async....54
6.2.3 Java Parallel Streams....55
6.2.4 Goroutinen....56
6.2.5 Darstellung der Ausführungszeiten aller parallelen Implementierungen....57
6.3 Beschleunigungsfaktor....58
6.4 Komplexität....60
6.5 Abschließende Bewertung....62
7 Zusammenfassung und Beantwortung der Forschungsfragen....63
7.1 Beantwortung der Forschungsfragen....63
7.2 Zusammenfassung....64
7.3 Ausblick....65
8 Quellen....67
9 Anhang....i

Abbildungsverzeichnis

Abbildung 1: Amdahlsches Gesetz....10
Abbildung 2: Prozessorauslastung POOL-R-J....45
Abbildung 3: Prozessorauslastung POOL-P-J....46
Abbildung 4: Prozessorauslastung ASYNC-R-C....46
Abbildung 5: Prozessorauslastung ASYNC-P-C....47
Abbildung 6: Prozessorauslastung STREAM-N-J....48
Abbildung 7: Prozessorauslastung STREAM-P-J....49
Abbildung 8: Prozessorauslastung GO-R-G....49
Abbildung 9: Prozessorauslastung GO-N-G....50
Abbildung 10: Prozessorauslastung GO-P-G....51
Abbildung 11: Prozessorauslastung: Vergleich aller parallelisierten Implementierungen (~490 Kreise)....52
Abbildung 12: Prozessorauslastung: Vergleich aller parallelisierten Implementierungen (~4130 Kreise)....53
Abbildung 13: Ausführungsdauer von POOL-R-J und POOL-P-J....54
Abbildung 14: Ausführungsdauer von ASYNC-R-C und ASYNC-P-C....55
Abbildung 15: Ausführungsdauer von STREAM-R-J, STREAM-N-J und STREAM-P-J 56
Abbildung 16: Ausführungsdauer von GO-R-G, GO-N-G und GO-P-G....57
Abbildung 17: Darstellung der Ausführungsdauer aller parallelen Implementierungen.…58
Abbildung 18: Beschleunigungsfaktor aller Implementierungen....59
Abbildung 19: Anzahl der Codezeilen der verschiedenen Implementierungen....60
Abbildung 20: Differenz der Anzahl an Codezeilen zwischen den verschiedenen Implementierungen....61

Tabellenverzeichnis

Tabelle 1: Nomenklatur: Parallelisierungstechniken....21
Tabelle 2: Nomenklatur: Implementierungsvarianten....21
Tabelle 3: Nomenklatur: verwendete Programmiersprachen....22
Tabelle 4: Messumgebung....42
Tabelle 5: konstante Simulationsparameter....42
Tabelle 6: variable Simulationsparameter: Messung der Prozessorauslastung ~ 490 Kreise....43
Tabelle 7: variable Simulationsparameter: Messung der Prozessorauslastung ~4130 Kreise....43
Tabelle 8: variable Simulationsparameter: Zeitmessung ~ 490 Kreise....44
Tabelle 9: variable Simulationsparameter: Zeitmessung ~ 2030 Kreise....44
Tabelle 10: variable Simulationsparameter: Zeitmessung ~ 4130 Kreise....44

Quelltextverzeichnis

Quelltextbeispiel 1 (C++): Race Conditions....11
Quelltextbeispiel 2 (C++): Deadlocks....14
Quelltextbeispiel 3 (Java): Erstellung eines Thread Pools....16
Quelltextbeispiel 4 (Java): Zuweisung von Tasks an einen Thread Pool....16
Quelltextbeispiel 5 (Java): Beendung eines Thread Pools....16

Quelltextbeispiel 6 (C++): asynchroner Aufruf einer Funktion mit async.....17
Quelltextbeispiel 7 (C++): Zugriff auf den Wert eines Futures.....17
Quelltextbeispiel 8 (Java): Erstellung eines Streams aus einer Collection.....18
Quelltextbeispiel 9 (Java): Anwendung eines Filter-Operators.....19
Quelltextbeispiel 10 (Java): Beendung eines Streams mittels terminalem Operator.....19
Quelltextbeispiel 11 (Go): manuelle Deklaration der zu verwendenden Prozessorkerne…20
Quelltextbeispiel 12 (Go): asynchroner Aufruf einer Goroutine.....20
Quelltextbeispiel 13 (Go): Beispielkommunikation mit Kanälen.....21
Quelltextbeispiel 14 (C++): async – Aktualisierung der Kreise.....28
Quelltextbeispiel 15 (Java): Erstellung eines Thread Pools mit definierter Größe.....29
Quelltextbeispiel 16 (Java): Thread Pool – Aktualisierung der Kreise.....29
Quelltextbeispiel 17 (Java): Thread Pool – Synchronisation und Fehlerbehandlung mit try/catch.....29
Quelltextbeispiel 18 (Java): Stream – Aktualisierung der Kreise.....30
Quelltextbeispiel 19 (Go): Goroutinen – Aktualisierung der Kreise.....31
Quelltextbeispiel 20 (Go): Goroutinen – Kommunikation über Kanäle.....32
Quelltextbeispiel 21 (Java): Thread Pool – Kollisionserkennung.....33
Quelltextbeispiel 22 (Java): Thread Pool – Synchronisierung und Fehlerbehandlung.....33
Quelltextbeispiel 23 (C++): Async – Kollisionserkennung.....33
Quelltextbeispiel 24 (C++): async – Kollisionsbehandlung und Bereinigung.....33
Quelltextbeispiel 25 (Java): Stream – Kollisionserkennung.....34
Quelltextbeispiel 26 (Java): Stream – Kollisionsbehandlung und Bereinigung.....34
Quelltextbeispiel 27 (Go): Goroutinen – Kollisionserkennung.....35
Quelltextbeispiel 28 (Go): Goroutinen – Kollisionsbehandlung mithilfe von Abfragen auf mehrere Kanäle.....36
Quelltextbeispiel 29 (Go): Berechnung der Dauer eines Simulationszyklus.....40

Anhang

Quellcode.....i

1 Einleitung

Die Komplexität von modernen Computerprogrammen, insbesondere von Spielen, nimmt stetig zu. Mit diesem Wachstum steigt jedoch auch der Ressourcenverbrauch der Programme, besonders deren Anforderungen an den Prozessor.

Bis in das Jahr 2005 wurde die Leistungssteigerung von Prozessoren durch eine Erhöhung deren Taktfrequenz erreicht (Gleim und Schüle 2011, S. 9). Dies führte jedoch zu einem immer höheren Energiebedarf der Prozessoren, wie zuletzt beim Intel Pentium 4, der knapp 100 Watt Strom verbrauchte (Wasson 2005), was nebenbei auch zu immer höheren thermischen Emissionen führte (Intel Corporation, o.A. a), die vom Prozessor abgeführt werden mussten.

Um die Leistungsaufnahme und die Wärmeproduktion von Prozessoren zu senken und gleichzeitig deren Leistung zu steigern, wurden stattdessen mehrere, geringer getaktete Prozessorkerne in einen Prozessor integriert (Gleim und Schüle 2011, S. 10).

Die Einführung von sogenannten Mehrkernprozessoren förderte ein Programmierparadigma, die sogenannte parallele Programmierung, das sich mit der Parallelisierung und Verteilung von Programmteilen auf mehrere Prozessorkerne auseinandersetzt. Da sich die anfänglich entwickelten, expliziten Methoden zur Parallelisierung von Programmen als fehleranfällig erwiesen, werden neue Parallelisierungstechniken entwickelt (Kasim, Zhang und See 2008). Ziel dieser Techniken ist es, Fehler bei der Parallelisierung von Programmen durch den Einsatz von Automatismen und Abstraktion zu reduzieren.

1.1 Zielsetzung

Diese Arbeit hat das Ziel, verschiedene Parallelisierungstechniken hinsichtlich ihrer Leistung und Bedienbarkeit zu evaluieren.
Da nicht alle der zu untersuchenden Techniken in einer Programmiersprache verfügbar sind, sollen für den Vergleich mehrere Programmiersprachen verwendet werden. Der Vergleich zwischen den verschiedenen Techniken soll jedoch so erfolgen, dass die gewählte Programmiersprache und deren Einflüsse möglichst nicht ins Gewicht fallen.
Folgende Aufgaben wurden konkretisiert, um das Ziel erreichen zu können:

- Die Beispielanwendung soll keine Sonderfälle abbilden, sondern eine große Relevanz für alltägliche Anwendungen haben.

- Es soll jeweils eine parallele Implementierung der Beispielanwendung unter Verwendung einer Parallelisierungstechnik entwickelt werden. Zusätzlich soll eine sequentielle Referenzimplementierung in der selben Programmiersprache entwickelt werden.
- Die verschiedenen Varianten der Beispielanwendung sollen in gängigen Programmiersprachen implementiert werden.

Um zu ermitteln, ob die Aufgabenstellungen hinreichend erfüllt wurden, sollen folgende Fragestellungen beantwortet werden:

- Welche Merkmale der Implementierungen können genutzt werden, um einen Vergleich zwischen parallelen und sequentiellen Referenzimplementierungen, sowie zwischen unterschiedlichen parallelen Implementierungen hinsichtlich ihrer Stärken und Schwächen zu ermöglichen?
- Wie unterscheiden sich die untersuchten Implementierungen gemessen an den definierten Merkmalen?

1.2 Methodik

In Arbeiten zu Parallelprogrammierungen werden oft Anwendungsbeispiele gewählt, die zwar klare Rahmenbedingungen schaffen, dafür aber wenig Alltagsrelevanz besitzen (z.B. Sortieralgorithmen). Entsprechend der Aufgabenstellung dieser Thesis wird hier ein anderer Weg gewählt.

Als Beispiel wurde eine Physiksimulation gewählt, die Bewegungen von Kreisen mit definierter Größe, Masse und Geschwindigkeit im zweidimensionalen Raum unter Einfluss von Schwerkraft berechnet. Eine solche Simulation kommt z.B. in zahlreichen Spielen zur Anwendung (Havok Group o.A.). Sie bietet die Möglichkeit, sowohl voneinander unabhängige Berechnungsschritte (z.B. die Positionsberechnung für jedes einzelne Objekt in einer Schleife) als auch abhängige (z.B. eine Kollisionserkennung) zu parallelisieren. Diese Vorgehensweise kann auch auf andere Probleme, wie z.B die Implementierung von Webcrawlern, übertragen werden (Google Inc. o.A. a).

Die Simulation wird mit Thread Pools, C++ async-Funktionen Java Streams und Goroutinen parallelisiert. Dabei handelt es sich um vier sehr häufig eingesetzte Parallelisierungstechniken.

Die parallelisierten Simulationen, sowie deren Referenzsimulationen werden mit im Vorfeld definierten Parametern ausgeführt (zur Darstellung der Vergleichskriterien s. Kapitel 5), um einen Vergleich anhand einer definierten Metrik[1] zu ermöglichen.

1.3 Erwartungen

Im Vorfeld der Arbeit werden folgende Erwartungen an die verschiedenen Implementierungen gestellt:

- Es wird erwartet, dass der programmatische Aufwand zur Parallelisierung im Vergleich zur sequentiellen Lösungen nicht wesentlich größer sein wird. Insbesondere sollte wenig Boilerplate-Code entstehen, d.h. Code, der in allen verwandten Varianten für die Verwaltung der Parallelisierung nötig ist.
- Die Ausführungsgeschwindigkeit sollte bei den parallelisierten Varianten der Simulation höher sein, als bei der sequentiellen Variante. Es wird erwartet, dass sich ein Zusammenhang zwischen der Anzahl der für die Berechnung zur Verfügung stehenden Prozessorkerne und der Erhöhung der Simulationsgeschwindigkeit ergibt.
- Die parallelisierte Implementierung sollte möglichst alle zur Verfügung stehenden Prozessorkerne nutzen und möglichst maximal auslasten. Dadurch soll, im Vergleich zur sequentiellen Referenzimplementierung eine höhere Gesamtauslastung des Prozessors entstehen, was zu einer geringeren Ausführungsdauer führen sollte. Während der Ausführung der Simulation sollte die Auslastung möglichst keinen Schwankungen unterliegen, welche auf eine Verzögerung in der Ausführung hindeuten. Des Weiteren sollte während der Ausführung der Simulation der ausgelastete Kern nicht wechseln, was die Ausführungsdauer ebenfalls negativ beeinflussen würde.

1.4 Aufbau der Arbeit

Im ersten Kapitel dieser Arbeit werden die Ziele und Erwartungen dieser Arbeit beschrieben.

Im zweiten Kapitel wird ein Überblick über verwandte Forschungen gegeben und das Forschungsthema gegenüber denselben abgegrenzt.

[1] Der Begriff *Metrik* wird in dieser Arbeit als Bezeichnung für eine Sammlung verschiedener, im Verlauf dieser Arbeit vorgestellter, Messgrößen verwendet.

Im dritten Kapitel werden grundlegende Techniken und Probleme der parallelen Programmierung vorgestellt, sowie eine kurze Einführung in die eingesetzten Techniken gegeben.
Im vierten Kapitel wird genauer auf die Implementierung der Physiksimulation eingegangen. Dabei werden einerseits die physikalischen Grundlagen der Simulation und deren Umsetzung, andererseits die verschiedenen Implementierungen der Parallelisierung und dabei auftretende Auffälligkeiten betrachtet.
Im fünften Kapitel wird eingehend auf die zu erhebenden Daten eingegangen, die nötig sind, um die verschiedenen Implementierungen zu vergleichen. Dabei werden die Gründe für die Wahl der Daten vorgelegt, die Messmethoden, mit denen die Daten erhoben werden, vorgestellt und die Messumgebung definiert.
Im sechsten Kapitel werden die gewonnenen Daten präsentiert und interpretiert.
Abschließend werden im letzten Kapitel die Forschungsfragen dieser Arbeit beantwortet, die entstandenen Ergebnisse zusammengefasst und ein Ausblick über mögliche weiterführende Forschungsansätze auf dem Gebiet der Parallelisierung gegeben.

2 Verwandte Arbeiten

Im Folgenden Kapitel werden drei Arbeiten vorgestellt, die sich ebenfalls mit dem Vergleich von Parallelisierungstechniken auseinandersetzen. Die Autoren entwickeln unterschiedliche Metriken, um die verschiedenen Techniken vergleichen zu können. Dieser methodische Ansatz wird in dieser Thesis übernommen und an die zu erreichenden Ziele und die verwendeten Techniken angepasst, weshalb die nachfolgend vorgestellten Arbeit für diese Thesis von Relevanz sind.

Basierend auf der Salishan High-Speed Computing Conference 1988 trägt J.T. Feo in seinem Buch „A Comparative Study of Parallel Programming Languages: The Salishan Problems“ verschiedene Lösungen der Salishan-Probleme zusammen, die mit Hilfe von parallelen Programmiersprachen entwickelt wurden (Feo 1992). Die Salishan-Probleme stellen eine Sammlung praktischer Programmieraufgaben dar, deren Lösung die Bearbeitung von verschiedenen Programmiermodellen, wie z.B. die Parallelisierung von verschachtelten Schleifen, sowie Datenstrukturen, wie z.B. Arrays und Streams, erfordert. Vorgestellt werden Ada, C*, Haskell, Id, Occam, Program Composition Notation, Scheme und Sisal. Der Vergleich der verschiedenen Lösungen obliegt dem Leser.

Im Gegensatz zur Sammlung von Feo werden in dieser Thesis nicht parallele Programmiersprachen, sondern verschiedene Techniken zur Parallelisierung von Programmen vorgestellt. Diese sollen so verglichen werden, dass die Vor- und Nachteile der jeweiligen Technik herausgearbeitet werden können. Die hier verwendete Beispielanwendung soll ebenfalls die Bearbeitung verschiedener Programmiermodelle abbilden.

Die Arbeit „Survey on Parallel Programming Model“ von Henry Kasim, Verdi March, Rita Zhang, und Simon See gibt einen theoretischen Überblick über verschiedene abstrakte parallele Programmiermodelle (Kasim, Zhang und See 2008). Neben der reinen CPU-Parallelisierung betrachten die Autoren auch Möglichkeiten, Programme, mit Hilfe von Grafikkarten, massiv zu parallelisieren. Die untersuchten Techniken sind: Pthreads, OpenMP, Cuda, MPI, UPC und Fortress. Die Autoren untersuchen dazu beispielsweise die Abbildung von Programmtasks auf Betriebssystemressourcen, oder die Synchronisierung und Kommunikation zwischen verschiedenen Programmtasks. Sie

bieten eine Richtlinie, die bei der Auswahl einer Parallelisierungstechnik für eine bestimmte Anwendungssituation helfen kann.

In der Arbeit „A Survey of High-Level Parallel Programming Models“ betrachten Evgenij Belikov, Pantazis Deligiannis, Prabhat Totoo, Malak Aljabri, und Hans-Wolfgang Loidl verschiedene High-Level Techniken zur parallelen Programmierung (Belikov u.a. 2013). Die Autoren vergleichen die verschiedenen Techniken anhand ihrer grundlegenden Funktionsweisen, wie z.B. deren Speicher- und Kommunikationsmodell, aber auch anhand deren Vermögen, explizite Verwaltungsanweisungen zur Parallelisierung zu abstrahieren. Neben Techniken zur Parallelisierung von Programmen auf dem Prozessor ziehen die Autoren auch Techniken zur Parallelisierung auf der Grafikkarte in Betracht. Zusätzlich werden einige Beispiele für Einsatzgebiete von Parallelisierung vorgestellt, wie z.B. eine n-Körper-Simulation, wie sie in dieser Thesis eingesetzt wird.

Wie auch die Arbeit „Survey on Parallel Programming Model“ vergleicht diese Arbeit die verschiedenen Parallelisierungstechniken anhand theoretischer Kriterien. Im Gegensatz zu den beiden Arbeiten werden in dieser Thesis die Programmiermodelle anhand einer Beispielanwendung hinsichtlich ihrer Praxistauglichkeit verglichen. Dabei wird der Fokus auf die Bedienbarkeit des Modells, sowie dessen Leistung und Ressourcennutzung gelegt.

3 Theoretische Grundlagen

Zunächst werden nun Grundlagen der parallelen Programmierung vorgestellt. Dazu wird zuerst ein kurzer Einblick in die Architektur von Mehrkernprozessoren und deren Nutzung durch moderne Betriebssysteme gegeben. Anschließend werden daraus resultierende, grundlegende Techniken und Probleme von paralleler Programmierung erläutert. Abschließend werden verschiedene, in dieser Arbeit genutzte, Parallelisierungstechniken und deren Lösungsvorschläge für oben genannte Probleme präsentiert.

3.1 Einkern- und Mehrkernprozessoren

Bis zum Jahre 2005 bestanden Prozessoren meist nur aus einem Kern. Da auf diese Weise Programme immer nur nacheinander ausgeführt werden können, war die Verarbeitungsgeschwindigkeit deutlich begrenzt. Um trotz der physikalischen Limitierungen, wie z.B. zu hohen Hitzeemissionen (Intel Corporation o.A. c), die Verarbeitungsgeschwindigkeit weiter zu steigern wurden Mehrkern-Prozessoren entwickelt. Diese erlauben, Teilprogramme zeitgleich auszuführen. Die Programme werden vom Betriebssystem als Tasks angesehen.

Die Ausführungsreihenfolge, sowie die Zeit, die ein Task für dessen Ausführung zugeteilt bekommt, werden durch einen Teil des Betriebssystems bestimmt, den sogenannten Scheduler (Glatz 2015, S. 156).

Im Folgenden wird die Ausführung mehrerer, logisch zusammenhängender Tasks betrachtet. Diese können z.B. verschiedene Komponenten eines Programmes repräsentieren. Es wird davon ausgegangen, dass die Tasks unabhängig voneinander bearbeitet werden können und dabei Fortschritt machen d.h. in endlicher Zeit enden. Die Ausführungsreihenfolge der Tasks hat keinen Einfluss auf die korrekte Funktionsweise des Programmes.

Werden diese Tasks nacheinander ausgeführt, wie es z.B. auf einem Prozessor mit nur einem Kern möglich ist, bezeichnet man die Ausführung als *nebenläufig* oder *pseudo-parallel* (Glatz 2015, S. 107f.). Werden die Tasks zeitgleich ausgeführt, bezeichnet man die Ausführung als *parallel* (Sun Microsystems 2001c). Dies kann, im Gegensatz zu Nebenläufigkeit jedoch nur auf einem Mehrkernprozessor stattfinden. Daher stellt diese Art der Ausführung eine Spezialisierung von Nebenläufigkeit dar. Hier wird Nebenläufigkeit als Oberbegriff für Parallelität und Nebenläufigkeit verwendet.

Startet ein Task einen weiteren Task nebenläufig, ohne mit der eigenen Ausführung zu warten bis dessen Ergebnis vorliegt, erfolgt die Ausführung des nebenläufigen Tasks aus Sicht des aufrufenden Tasks asynchron. Der asynchrone Task wird entweder pseudoparallel oder sogar parallel ausgeführt. Aus diesem Grund wird Asynchronität oftmals mit Nebenläufigkeit oder Parallelität gleichgesetzt.

3.1.1 Scheduling

Die Entscheidung, welcher Task Rechenzeit auf einem Prozessorkern zugewiesen bekommt, nennt man Scheduling.

Eine der einfachsten Formen ist das Round-Robin-Verfahren, bei dem jeder Task die gleiche Rechenzeit zugewiesen bekommt (Glatz 2015, S. 180). Ist diese abgelaufen, wird der laufende Task abgebrochen und zwischengespeichert, sowie ein neuer Task geladen und ausgeführt. Die Unterbrechungen werden so organisiert, dass die Tasks fehlerfrei an der Stelle weiterlaufen, an der sie unterbrochen wurden.

Je mehr Tasks auf dem System laufen, desto geringer wird die Zeitscheibe für jeden einzelnen Task, was die Ausführungsdauer des Tasks negativ beeinflusst. Wird ein Task häufig zwischen Prozessorkernen verschoben, beeinflusst dies seine Ausführungsdauer ebenfalls negativ, da seine Daten stets zwischengespeichert und neu geladen werden müssen.

3.1.2 Prozesse und Threads

Soll ein Task ausgeführt werden, wird dafür vom Betriebssystem mindestens ein neuer Prozess bereitgestellt. Dieser stellt die Ausführungsumgebung für das Programm dar. Dabei wird dem Task simuliert, er greife allein auf Ressourcen, wie z.B. Hardware, zu. Dazu wird z.B. der Arbeitsspeicher des PCs vom Betriebssystem abstrahiert, und dem Prozess nur den für ihn relevanten Teil zur Verfügung gestellt. Auch der Prozessor wird durch das Betriebssystem abstrahiert.

Wird ein Prozess durch den Scheduler unterbrochen, muss dessen gesamter Zustand gesichert werden. Dieser Vorgang ist problematisch, da für die Sicherung des aktuellen Prozesses und die Einspielung eines neuen viel Zeit benötigt wird, da z.B. Zugriffe auf den Arbeitsspeicher nötig sein können.

Um diese Ladezeit zu beschleunigen, wurde eine Untereinheit von Prozessen geschaffen, sogenannte Threads. Threads existieren im Kontext des Prozesses und kön-

nen auf dessen Ressourcen zugreifen. Innerhalb eines Prozesses können nahezu beliebig viele Threads existieren.

Das Betriebssystem wird in der Regel hierarchisch zusammengehörige Threads so schedulen, dass häufige Prozesswechsel vermieden werden.

3.1.3 Multithreading

Um den Wechsel zwischen verschiedenen Threads zu beschleunigen und zeitintensive Zugriffe auf den Arbeitsspeicher zu verbergen, sind auf modernen Prozessoren Register und Pufferspeicher doppelt vorhanden. Techniken für das Nachladen eines Threads werden von alle Threads gemeinsam genutzt (Rauber und Rünger 2010, S. 19). Dadurch kann während der Ausführung eines Threads zeitgleich ein weiterer geladen werden (Valles 2009). Wird ein Thread ausgewechselt, kann der nächste direkt aus dem zweiten Speichersatz ausgeführt werden.

3.1.4 Shared Memory

Da Threads auf die Ressourcen des Prozesses zugreifen, können Situationen entstehen, in denen gemeinsamer Speicher geteilt wird. Aktuelle Prozessoren bilden dies über eine Shared-Memory-Architektur ab (Gleim und Schüle 2011, S. 10). Mehrere Prozessorkerne greifen dabei beispielsweise auf gemeinsame Caches oder den Hauptspeicher zurück. Da auf den verschiedenen Kernen unterschiedliche Threads, oder Threads mit einem unterschiedlichen Zustand ausgeführt werden können, kann es auch zu asynchronen Datenzugriffen kommen.

3.1.5 Asynchrone Datenzugriffe auf gemeinsame Ressourcen

Asynchrone Programme, die Datenzugriffe auf gemeinsame Ressourcen durchführen, müssen einige grundsätzliche Voraussetzungen beachten:

- Je weniger Zugriffe auf gemeinsame Ressourcen erfolgen, desto besser eignet sich ein Programm, um nebenläufig ausgeführt zu werden.
  Zugriffe auf geteilte Ressourcen müssen besonders abgesichert werden, um gewährleisten zu können, dass das Programm korrekt abläuft. Diese Absicherung kommt meist einer zeitlichen Synchronisierung, d.h. einer Hintereinanderschaltung von eigentlich parallelen Programmteilen, gleich. Vor allem schreibende Zugriffe sind problematisch, da andere asynchron laufende Programmteile über die Aktualisierung informiert werden müssen.

- Jeder Synchronisierungspunkt im Programm beeinträchtigt die Effizienz des nebenläufigen Programmes, da eventuell schnellere Programmteile auf Daten von langsameren warten müssen. Die Effizienz einer nebenläufigen Lösung nähert sich mit zunehmendem Synchronisierungsaufwand der einer sequentiellen Lösung an (Herlihy und Shavit 2008, S. 13).
- Im Umkehrschluss steigt die Effizienz eines Programms nie linear zur Anzahl an verfügbaren Prozessen/Threads, da immer ein gewisser Aufwand zur Synchronisierung bleibt (Herlihy und Shavit 2008, S. 13).

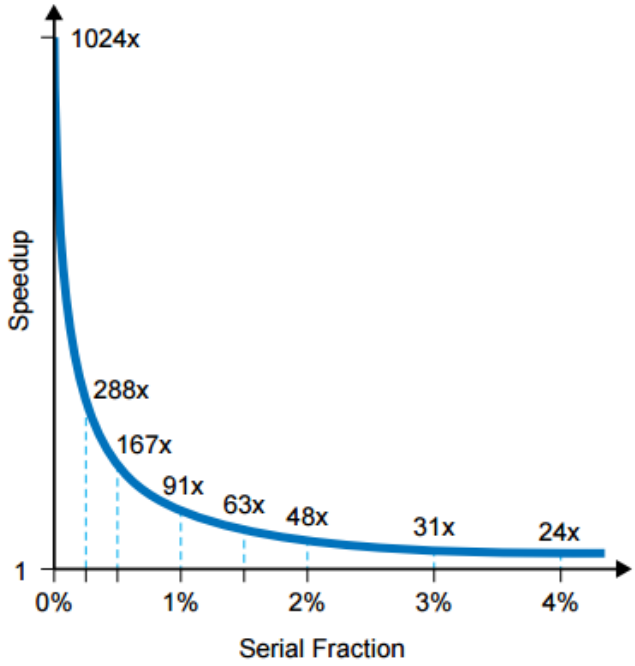

Abbildung 1: Amdahlsches Gesetz

Quelle: (Gustfason 1988)

Abbildung 1 verdeutlicht die letzten beiden Punkte graphisch. Dabei wird ersichtlich, dass der Leistungsgewinn (Speedup) durch Parallelisierung nicht linear abläuft, sondern durch Synchronisierungsaufwand gebremst wird. Dieser entsteht entweder bei einer sehr hohen Prozessoranzahl, oder steigendem Anteil des sequentiellen Teiles (Serial Fraction) des Programmes. Je höher der parallelisierte Teil eines Programmes ist, desto höher ist auch der Leistungsgewinn.

3.1.6 Probleme bei Nebenläufigkeit

Die Ausführungsreihenfolge von nebenläufigen Tasks innerhalb eines parallelen Programmes hängt nicht vom Programm selbst, sondern vom Betriebssystem ab. Dieses führt Tasks abhängig vom aktuellen Zustandes des gesamten Systems aus. Dabei wird

sowohl die Anzahl der aktuell laufenden Tasks, als auch deren Auslastung von Ressourcen in Betracht gezogen.
Da ein Programm auf diese Daten keinen Zugriff hat, ist die Ausführungsreihenfolge von Operationen aus seiner Sicht nicht deterministisch (Grama u.a. 2003, S.287).

3.1.6.1 Race Conditions

Greifen mehrere Tasks gleichzeitig schreibend auf eine Ressource zu, ist deren endgültiger Zustand von der Reihenfolge der Zugriffe abhängig (Gleim und Schüle 2011, S. 76). Da diese jedoch vom Betriebssystem abhängt und nicht durch das Programm selbst beeinflusst werden kann, kann der Endzustand der Berechnung durch das Programm nicht vorhergesagt werden.

Im folgenden Beispiel wird eine globale Variable (Z. 1) in verschiedenen Threads inkrementiert und dekrementiert. Da zuerst der inkrementierende (Z. 18) und anschließend der dekrementierende Thread (Z. 19) gestartet wird, wird erwartet, dass die Variable nach Bearbeitung den Wert Null annimmt (Z. 24). Tatsächlich variiert das Ergebnis jedoch mit jedem Durchlauf. Dieses Phänomen wird „Wettlaufsituation“ (Race Condition) genannt.

```
1   int counter = 0;
2
3   void incrCount() {
4       for( int i = 0; i < 10000; ++i ) {
5           counter++;
6       }
8   }
9
10  void decrCount() {
11      for( int i = 0; i < 10000; ++i ) {
12          counter--;
13      }
14  }
15
16
17  int main() {
18      std::thread t1(incrCount);
19      std::thread t2(decrCount);
20
21      t1.join();
22      t2.join();
23
24      std::cout << counter << std::endl;
25
26      return 0;
27   }
```

Quelltextbeispiel 1 (C++): Race Conditions

3.1.7 Synchronisierung von Datenzugriffen

Um Fehler beim Datenzugriff auf geteilte Ressourcen zu vermeiden, muss dieser synchronisiert werden, der Zugriff wird damit deterministisch geregelt. Dazu werden einige grundlegende Techniken verwendet. Diese stellen einen wechselseitigen Ausschluss (mutual exclusion) sicher (Herlihy und Shavit 2008, S. 22), erlauben also nur jeweils einer ausführenden Einheit, auf geteilte Ressourcen zuzugreifen. Im Folgenden werden die für diese Arbeit relevanten Techniken vorgestellt.

a) **Mutex**

   Ein Mutex (oder Lock) stellt eine Sperr-Variable dar, die Zugriffe auf einen kritischen Bereich vereinzelt. Dieser kann sowohl eine, als auch mehrere Instruktionen umfassen. Jeder kritische Bereich wird über eine eigene Sperrvariable verwaltet.

   Greift ein Task auf eine kritische Ressource zu, versucht er zuerst, deren Sperrvariable zu erlangen:

   - Wenn kein anderer Task auf die Ressource zugreift, wird ihm die Sperrvariable zugeeilt. Nach Zugriff auf die Ressource sollte der Task die Sperrvariable wieder freigeben um anderen Tasks den Zugriff auf den Bereich zu ermöglichen.
   - Wenn zum Zeitpunkt des Zugriffes die Variable von einem anderen Thread erlangt wurde, wartet der Aufrufer, bis der andere Thread die Variable freigibt, um sie dann selbst zu erlangen (Herlihy und Shavit 2008, S.23).

b) **Semaphore**

   Wenn eine kritische Ressource mehrere zeitgleiche Zugriffe erlaubt, können diese durch eine Semaphore geregelt werden. Semaphoren erweitern das Konzept des binären Locks um eine Zählvariable. Das Lock bekommt eine Obergrenze an parallelen Zugriffen auf die Ressource zugewiesen. Bei jedem Versuch, Zugriff auf die Ressource zu erhalten, wird die Semaphore inkrementiert, beim Freigeben der Ressource dekrementiert (Herlihy und Shavit 2008, S.189).

   Die Korrektheit der Zählvariable bleibt dabei immer garantiert. Semaphoren können z.B. genutzt werden, um das Consumer-Producer-Problem zu lösen (Maurer 2012, S.56) (Sun Microsystems 2001a).

c) **Atomare Operationen**

Eine Möglichkeit, auf kritische Ressourcen zuzugreifen, ohne einen wechselseitigen Ausschluss zu benötigen, sind atomare Operationen (atomic operations). Deren Ausführung erscheint anderen Threads instantan, sie können also nicht unterbrochen werden. Dadurch können Konflikte beim Zugriff auf die Daten verhindert werden. Atomare Operationen sind jedoch meist nur eingeschränkt verfügbar, z.B. zum Inkrementieren und Dekrementieren einer Variablen (Intel Corporation o.A. b). Eine atomare Ausführung gesamter Quelltextblöcke ist meist nicht möglich, diese müssten in einer Transaktion zusammengefasst werden[2].

d) **Barrieren**

Wenn mehrere Tasks an einer bestimmten Stelle im Programmablauf synchronisiert werden müssen, kann eine Barriere (barrier) verwendet werden. Diese stellt sicher, dass alle, beim Erstellen der Barriere definierten, Tasks diese Stelle durchlaufen haben (Sun Microsystems 2001b).

3.1.8 Probleme der Synchronisierung

Durch die Synchronisierung von parallelen Zugriffen auf gemeinsame Daten können jedoch weitere Probleme entstehen:

3.1.8.1 Deadlocks

Wenn Threads versuchen, in umgekehrter Reihenfolge Zugriff auf geteilte Mutexe oder Semaphoren zu erlangen, kann es zu Deadlocks (Verklemmungen) kommen. Im folgenden Beispiel setzt der erste Thread Mutex A (Z. 4) und versucht, Mutex B zu erlangen (Z. 8). Dieser kann sich jedoch im Besitz des zweiten Threads befinden(Z. 14), der wiederum auf die Freigabe von Mutex A wartet (Z. 17).

```
1   std::mutex mutexA, mutexB;
2
3   void func1() {
4       mutexA.lock();
5       std::cout << "holding lock A, trying to aquire mutex B" <<
6   std::endl;
8       mutexB.lock();
9
10      return;
```

[2] Eine Transaktion ist eine Zusammenfassung mehrerer Operationen zu einer einzigen. Kommt es bei Ausführung einer Operation zu Fehlern, so wird die gesamte Operationskette revidiert. Dabei wird der Datenbestand vor Ausführung der bereits durchgeführten Operationen wiederhergestellt. Eine Transaktion gilt erst dann als erfolgreich, wenn alle Operationen korrekt durchgeführt wurden (Herlihy und Shavit 2008, *S. 421*).

```
11  }
12
13  void func2() {
14      mutexB.lock();
15      std::cout << "holding lock B, trying to aquire mutex A" <<
16  std::endl;
17      mutexA.lock();
18
19      return;
20  }
21
22  int main() {
23      std::thread t1(func1);
24      std::thread t2(func2);
25
26      t1.join();
27      t2.join();
28
29      return 0;
30  }
```

Quelltextbeispiel 2 (C++): Deadlocks

Eine Möglichkeit, Deadlocks zu vermeiden, ist die Änderung der Reihenfolge des Zugriffes. Mehrere Mutexe sollten immer in gleicher Reihenfolge erlangt werden (Microsoft Corporation o.A. e).

Eine weitere Möglichkeit wäre, nur eine gewissen Zeit auf das Erlangen eines Locks zu warten, und, falls das Lock nicht erlangt wurde, eine Fehlerbehandlung durchzuführen. Dadurch wäre sichergestellt, dass das Programm Fortschritt machen kann (Oracle o.A. h).

3.1.8.2 Starvation

Starvation (Verhungern) tritt auf, wenn ein Task eine Ressource sehr lange blockiert, während ein oder mehrere andere versuchen, Zugriff auf diese zu bekommen. Da ihnen das nicht gelingt, „verhungern“ sie. Im Unterschied zu Deadlocks macht in einer Starvation-Situation ein einzelner Task jedoch weiterhin Fortschritt (Oracle o.A. l).

3.2 Eingesetzte Parallelisierungstechniken

Da beim Einsatz der verschiedenen Synchronisierungstechniken oft Fehler unterlaufen, werden diverse Techniken entwickelt, die diese Probleme abstrahieren und eine einfach zu bedienende Programmierschnittstelle (API, Application Program Interface) zur Verfügung stellen.

Sie erlauben es, Programmteile asynchron zu starten, und damit nebenläufig auszuführen. Die dafür nötigen Ressourcen, wie z.B. Threads, werden, soweit möglich, durch die Technologie automatisch verwaltet und bestenfalls auf das jeweilige System ange-

passt. Die Synchronisation von Datenzugriffen muss teilweise manuell erfolgen, teilweise stellen die Techniken bestimmte Datencontainer zur Verfügung, die nebenläufige Zugriffe auf sich selbst organisieren. Im Folgenden werden die in dieser Arbeit verwendeten Parallelisierungstechniken vorgestellt:

3.2.1 Thread Pool

Thread Pools stellen eine Gruppe von Threads bereit, die zur Parallelisierung genutzt werden können. Da die Threads für beliebige Tasks wiederverwendet werken können, reduziert sich der Verwaltungsaufwand (Overhead), der beim Erstellen eines Threads entsteht. Die Verwaltung der Threads wird dabei vom Pool selbst übernommen. Durch verschiedene Verwaltungsprinzipien werden zwei wesentliche Typen von Thread Pools unterschieden:

- Fixed Thread Pools stellen eine definierte Anzahl an Threads bereit, die zur Abarbeitung von Tasks verwendet werden können. Werden mehr Tasks an den Thread Pool übergeben, als dieser freie Threads besitzt, werden die Tasks in eine Warteschlange verschoben und, sobald ein Thread frei wird, bearbeitet. Somit kann der Fixed Thread Pool auch eine sehr hohe Anzahl Tasks bearbeiten, ohne sich selbst durch Scheduling-Overhead zu blockieren.
- Cached Thread Pools stellen genau so viele Threads zur Verfügung, wie es Tasks gibt. Wenn Threads für eine bestimmte Zeit nicht genutzt werden, beendet sie der Pool. Dadurch verändert sich die Größe des Cached Thread Pools dynamisch. Bei hoher Anzahl an Tasks und damit Threads wird jedoch der Aufwand durch Scheduling sehr hoch.

Da die Anzahl an Tasks in der verwendeten Anwendung sehr groß werden kann, wurde für die Parallelisierung ein Fixed Thread Pool gewählt.

Für diese Arbeit wurde die Implementierung von Thread Pools in der Programmiersprache Java genutzt. Diese steht in der Bibliothek *util.concurrent* zur Verfügung (Oracle o.A. f). Da die Bibliothek seit der Java-Version 5 zur Verfügung steht (Oracle o.A. e), ist sie sehr weit verbreitet, was sie als Beispielwerkzeug hervorhebt.

3.2.1.1 Initialisierung

Ein Fixed Thread Pool wird über den Aufruf

```
 ExecutorService pool = newFixedThreadPool(int nThreads, ThreadFactory
threadFactory);
```

Quelltextbeispiel 3 (Java): Erstellung eines Thread Pools

gestartet. *nThreads* gibt dabei die Größe des Pools an, der optionale Parameter threadFactory kann eine spezielle Methode zur Erstellung der Threads angeben (Oracle o.A. c bzw. Oracle, Executors (Java Platform SE 7).).

3.2.1.2 Zuweisung von Tasks

Über den Aufruf von

```
pool.execute( Runnable command );
```

Quelltextbeispiel 4 (Java): Zuweisung von Tasks an einen Thread Pool

kann dem Pool ein Tasks in der Form eines Runnables, oder Callables[3] zugewiesen werden (Oracle, o.A. k).

3.2.1.3 Beendung

Über den Funktionsaufruf

```
pool.shutdown()
```

Quelltextbeispiel 5 (Java): Beendung eines Thread Pools

kann der Thread Pool geordnet beendet werden. Nach dem Funktionsaufruf nimmt der Pool keine weiteren Tasks an und arbeitet alle laufenden Tasks ab. Der Pool wartet jedoch nicht, bis alle laufenden Tasks beendet sind, dies muss separat gesteuert werden (Oracle, o.A. n).

3.2.2 C++ 11 – async

Die Funktion async erlaubt es, Funktionen asynchron auszuführen. Die nötige Synchronisierung wird dabei durch Async selbst geregelt (Stroustrup 2015, S. 1346).

Als Grundlage für Async dienen Threads. Für jeden asynchronen Aufruf wird ein neuer Thread gestartet, der die Funktion ausführt (cppreference.com 2015a) (Will 2012, S. 329). Um zu verhindern, dass zu viele Threads gestartet werden, und dadurch die Effizienz des Programmes gesenkt wird, können die Threads in einem Thread Pool

[3] Ein Runnable oder Callable stellt ein Objekt dar, das von anderen Instanzen ausgeführt werden soll. Dazu implementiert es die Methode *run()*. Im Gegensatz zu einem Runnable gibt ein Callable einen Wert zurück (Oracle o.A. k) (Oracle o.A. a).

gestartet werden, der auf die Bedingungen des PCs angepasst ist (cppreference.com 2015a).

Async wurde in mehreren anderen Programmiersprachen implementiert, unter anderem in C# und Visual Basic (Microsoft Corporation o.A. a bzw. Microsoft Corporation o.A. b). Da async erst in Version 11 in den Sprachkern von C++ aufgenommen wurde (Toit 2012, S. 1155), konnte die Implementierung von Erfahrungen aus verwandten Techniken profitieren.

3.2.2.1 Initialisierung

Ein asynchroner Aufruf einer Funktion wird mit dem Funktionsaufruf *async()* gestartet (Smith 2015, S. 1203).

```
auto future = std::async( asyncFunc );
```

Quelltextbeispiel 6 (C++): asynchroner Aufruf einer Funktion mit async

Die Funktion erwartet dabei ein Callable (cppreference.com 2015b). Dies kann z.B. eine Funktion oder ein Lambda[4] sein.

3.2.2.2 Beendung

Die Funktion gibt einen Platzhalter für einen zukünftigen Wert, ein Future zurück, über das später mit dem Aufruf der Funktion *.get()* auf den Rückgabewert der Funktion zugegriffen werden kann.

```
future.get();
```

Quelltextbeispiel 7 (C++): Zugriff auf den Wert eines Futures

Die Funktion *get()* erfüllt dabei mehrere Aufgaben (Smith 2015, S. 1199):

- Sie blockiert solange, bis die asynchron aufgerufene Funktion beendet und ein Ergebnis bereitsteht.
- Sie liefert das Ergebnis, das die asynchrone Funktion über ein Future zurück gibt.
- Werden in der asynchronen Funktion Exceptions geworfen, gibt *get()* diese an die aufrufende Funktion weiter.

[4] Im Gegensatz zu „normalen“ Funktionen sind Lambda-Funktionen anonym, d.h. nicht mit einem Namen verknüpft (Oracle o.A. g bzw. Microsoft Corporation o.A. d). Dies ermöglicht eine sehr kompakte Schreibweise. Lambda-Funktionen basieren auf dem Lambda-Kalkül von A. Church (Church 1936).

3.2.2.3 Kommunikation

Asynchrone Funktionen verwenden Futures und Promises zur Kommunikation und Übergabe von Rückgabewerten.

- Futures stellen zukünftige Rückgabewerte einer Funktion dar (Smith 2015, S. 1198). In der aufrufenden Funktion können sie wie ein normaler Wert verwendet werden. Ein Aufruf der Funktion *get()* blockiert den aufrufenden Thread und wartet, bis der Wert zur Verfügung steht. Eine andere Möglichkeit der Synchronisierung stellt die Funktion *wait[_until]()* dar, die entweder solange blockiert, bis die Funktion ausgeführt, oder ein Zeitlimit erreicht wird.
- Promises stellen den Rückgabewert einer Funktion aus Sicht der aufgerufenen Funktion dar. Sie werden mit einem Future assoziiert. Ist die Berechnung der Funktion beendet, wird über *set_value()* ein Rückgabewert gesetzt, der dann mit dem Future synchronisiert wird.

3.2.3 Java Parallel Streams

Java Parallel Streams erlauben es, verkettete Operationen parallel auf alle Elemente einer Datenquelle auszuführen und diese dabei zu modifizieren. Als Quelle kann dabei ein beliebiger Datencontainer wie z.B. eine Collection (Oracle o.A. d), ein Array, oder BitStreams dienen (Winterberg 2015).

Die Laufzeitumgebung von Java, die Java Virtual Machine (Lindholm u.a. 2015) garantiert jedoch keine parallele Ausführung. Wäre diese ineffizient, da beispielsweise die Anzahl der zu bearbeitenden Objekte zu gering ist, wird der Stream sequentiell ausgeführt (Oracle o.A. j).

Da Parallel Streams nur in der Java-Version 8 verfügbar sind (Oracle o.A. j), und in dieser Form in keine weiteren Programmiersprachen vorliegen, wurde für diese Arbeit auf diese Implementierung zurückgegriffen.

3.2.3.1 Initialisierung

Parallel Streams können über den Aufruf *source.parallelStream()* erzeugt werden.

```
source.parallelStream()
```

Quelltextbeispiel 8 (Java): Erstellung eines Streams aus einer Collection

Der Aufruf liefert einen Stream zurück, an den nun Operationen angefügt werden können.

3.2.3.2 Intermediate Operators

Zwischenliegende Operationen (intermidiate Operators) geben nach ihrer Abarbeitung immer einen neuen Stream zurück, wodurch sie verkettet (konkateniert) werden können. Der Ausgabestream muss jedoch nicht dem Eingabestream entsprechen. Dies trifft z.B. auf den Filter-Operator zu, der, im Vergleich zum Eingabestream, ein den Kriterien entsprechenden, kleineren Ausgabestream erzeugt.

```
source.parallelStream()
   .filter( p -> p.lifeTime >= 0 )
```

Quelltextbeispiel 9 (Java): Anwendung eines Filter-Operators

Intermediate Operations werden immer „lazy“ ausgewertet, d.h. nicht direkt beim eigentlichen Aufruf der Funktion, sondern erst beim Aufruf eines terminalen Operators. Dies ermöglicht höhere Effizienz bei der Ausführung, da Befehle optimiert werden können (Oracle o.A. i).

3.2.3.3 Terminal Operators

Terminale Operatoren (terminal Operators) beenden den Stream, indem sie keinen Stream, sondern einen primitiven Datentyp, einen Datencontainer, oder überhaupt keinen Datentyp zurück geben (Inden 2014, S. 37).
Über einen terminalen Operator wie z.B. collect() kann ein Stream beendet werden.

```
source.parallelStream()
    .collect( Collectors.toList() );
```

Quelltextbeispiel 10 (Java): Beendung eines Streams mittels terminalem Operator

In diesem Beispiel gibt der Operator eine Liste aller Objekte zurück, die durch den parallelen Stream bearbeitet wurden.

3.2.4 Goroutinen

Goroutinen stellen eine Möglichkeit zur Verfügung, Funktionsaufrufe asynchron zu gestalten.

Diese werden, bis zu Version 1.5, jedoch nicht automatisch parallelisiert, sondern nur nebenläufig ausgeführt (Google Inc. o.A. e). Um die Ausführung zu parallelisieren, muss der Laufzeitumgebung von Go die Anzahl der zu nutzenden Prozessorkerne explizit über den Parameter *runtime.GOMAXPROCS* zur Verfügung gestellt werden. Im folgenden Beispiel wird zuerst die Anzahl der Kerne ermittelt, um diese als maximal zu nutzende Anzahl an Kernen zu deklarieren.

```
NumberOfCores = runtime.NumCPU()
runtime.GOMAXPROCS( NumberOfCores )
```

Quelltextbeispiel 11 (Go): manuelle Deklaration der zu verwendenden Prozessorkerne

Für Version 1.5 von Go ist geplant, diesen standardmäßig auf die Anzahl der verfügbaren Prozessorkerne zu setzen (Google Inc. 2015).
Goroutinen werden über einen eigenen Scheduler der Go-Laufzeitumgebung auf Betriebssystemthreads abgebildet (Google Inc. o.A. d). Da mehrere Goroutinen in einem Thread ausgeführt werden können, ist der Wechsel zwischen verschiedenen Goroutinen sehr ressourcenschonend, da nur die Befehle innerhalb eines Threads ausgetauscht werden müssen, nicht jedoch der komplette Thread (Gerrand 2013a). Dies ermöglicht es, sehr viele Goroutinen auf einmal zu starten, ohne dass diese durch zu hohen Verwaltungsaufwand blockiert werden.
Die Kombination von Goroutinen mit der Kommunikation über Kanäle und das starke Wachstum der Programmiersprache Go (Gerrand 2013b) waren der Grund für die Auswahl dieser Implementierung.

3.2.4.1 Initialisierung

Um eine Goroutine zu starten, wird vor den „normalen“ Funktionsaufruf das Schlüsselwort *go* gestellt.

```
go testFunc()
```

Quelltextbeispiel 12 (Go): asynchroner Aufruf einer Goroutine

3.2.4.2 Beendung

Eine Goroutine beendet, genau wie eine normale Funktion, mit dem return-Statement, oder am Ende des Funktionsrumpfes.

3.2.4.3 Kommunikation

Die Kommunikation zwischen verschiedenen Goroutinen kann über Kanäle (Channels) erfolgen (Google Inc. o.A. c).
Diese stellen ein unidirektionales, typisiertes Kommunikationsmedium dar, deren Zugriff standardmäßig synchronisiert erfolgt (Ben-Ari 2006, S. 181).

Im folgenden Beispiel wird ein Kanal erstellt, über den Ganzzahlen (*int*) gesendet werden können (Z. 8). Die asynchron ausgeführte Goroutine schreibt einen Wert in den Kanal (Z. 2), die aufrufende Funktion liest den Wert (Z. 11). Der lesende Zugriff auf

den Kanal erfolgt im Beispiel blockierend, der Aufrufer wartet also so lange, bis er einen Wert gelesen hat.

```
1   func testFunc(channel chan int) {
2      channel <- 42
3
4      return
5   }
6
7   func main() {
8      channel := make(chan int)
9      go testFunc(channel)
10
11     fmt.Println( <-channel )
12  }
```

Quelltextbeispiel 13 (Go): Beispielkommunikation mit Kanälen

Kanäle können auch mit einem Puffer initialisiert werden, in diesem Fall muss die gewünschte Größe angegeben werden.

3.2.5 Nomenklatur

Im Verlauf der Arbeit werden die jeweiligen Implementierungen mit folgenden Bezeichnungen angesprochen:
Parallelisierungstechnik – Implementierungsvariante – Programmiersprache

Mögliche Werte für die Parallelisierungstechnik sind:

ASYNC	C++11 – async
POOL	Thread Pool
STREAM	Java Parallel Streams
GO	Goroutinen

Tabelle 1: Nomenklatur: Parallelisierungstechniken

Für die Implementierungsvariante können folgende Werte stehen:

P	parallel
N	nebenläufig
S	sequentiell, jedoch keine Referenzimplementierung
R	sequentielle Referenz

Tabelle 2: Nomenklatur: Implementierungsvarianten

Folgende Programmiersprachen werden eingesetzt:

C	C++ 11
J	Java SE 8
G	Go 1.3.3

Tabelle 3: Nomenklatur: verwendete Programmiersprachen

Um die parallele Implementierung, die async unter C++ verwendet wird, anzusprechen, lautet die Beizeichnung:

ASYNC-P-C

Die Bezeichnung für die nebenläufige Implementierung unter Verwendung von Goroutinen lautet:

GO-N-G

4 Implementierung der Testanwendung

In diesem Kapitel werden Gründe für die Wahl der Testanwendung und deren Implementierung vorgestellt.

Als Beispielanwendung dient eine Physiksimulation. Für die konkrete Umsetzung wird eine zweidimensionale Simulation gewählt. Diese simuliert starre Kreise mit einer definierten Masse, Größe und Geschwindigkeit, die sich unter dem Einfluss von Gravitation bewegen. Kollisionen zwischen Kreisen werden erkannt und behandelt.

Für die Wahl der Simulation sprechen folgende Gründe:

- Die Implementierung der eigentlichen zweidimensionalen Kreisphysik kann relativ schnell umgesetzt werden, da z.B. die Kollisionserkennung ohne Berücksichtigung von Aufprallwinkeln erfolgen kann. Dies erlaubt, stärkeren Fokus auf die Parallelisierung der Anwendung zu richten.
- Teile der Anwendung lassen sich naiv parallelisieren. Die Berechnungen von Position und Geschwindigkeit können unabhängig für jedes Objekt durchgeführt werden und erfordern daher geringen Synchronisierungsaufwand. Anhand dieses Teiles der Anwendung kann gezeigt werden, wie die verschiedenen Parallelisierungstechniken voneinander unabhängige Berechnungen umsetzen.
- Für die Kollisionserkennung müssen die Positionen von allen Kreisen gegenseitig geprüft werden. Vor diesem Schritt muss die Simulation einen definierten Zustand einnehmen, das Programm muss also synchronisiert werden. Dies erlaubt es, verschiedene Synchronisierungsmethoden zu testen.
- Die gewählte Physiksimulation stellt eine Anwendung dar, die einen sehr allgemeinen Problembereich abdeckt, wie z.B. die Parallelisierung von voneinander unabhängigen oder abhängigen Schleifen, die auch in anderen Kontexten zu finden sind. Daher bietet die Anwendung einen alltäglicheren Ansatz als z.B. ein Sortieralgorithmus, bei dem die Parallelisierung anhand eines sehr eng beschriebenen Problemfeldes optimiert werden kann.

4.1 Programmstruktur

Wesentliche Bestandteile der Anwendung sind eine simulierte Welt, die den physikalischen Rahmen für die Simulation vorgibt, sowie ein Emitter, der dieser Welt Kreise

hinzufügt. Alle Komponenten werden durch ein Teilprogramm kontrolliert, das auch die Aufzeichnung der Messdaten übernimmt.

4.1.1 Emitter

Um dem System neue Kreise hinzuzufügen, wird ein Emitter implementiert.
Dieser schießt Kreise mit definierter Geschwindigkeit und definiertem Winkel in die simulierte Welt. Zusätzlich wird jedem Kreis eine definierte Lebenszeit zugewiesen.
Als physikalische Grundlage für den Emitter dient der schiefe Wurf mit Anfangsgeschwindigkeit im Schwerefeld. Dieser bietet den Vorteil, dass Kreise mit einer Anfangsgeschwindigkeit definiert werden können, ohne weitere Parameter, wie sie z.B. für einen Federstoß notwendig wären, anzugeben[5].

4.1.1.1 Startwinkel

Der Startwinkel bildet die Grundlage zur Berechnung der zweidimensionalen Anfangsgeschwindigkeit der Kreise. Zur besseren Verständlichkeit wird dieser in Grad angegeben. Da für die Berechnung der Geschwindigkeit in die jeweilige Dimension der Winkel im Bogenmaß vorliegen muss, wird er mit folgender Formel umgerechnet:

$$\alpha' = \alpha * \frac{\pi}{180}$$

Formel 1: Umrechnung von Bogen- in Gradmaß
(Merzinger und Wirt 2006, S. 25)

4.1.1.2 Anfangsgeschwindigkeit

Aus Anschaulichkeitsgründen wird die Anfangsgeschwindigkeit als Skalar angegeben. Mithilfe des Startwinkels kann diese dann in einen zweidimensionalen Vektor umgerechnet werden. Um die Komponenten für x-, und y-Richtung zu ermitteln, wird das System aus Geschwindigkeiten als rechtwinkliges Dreieck angesehen. Die Gesamtgeschwindigkeit stellt die Hypotenuse dar, die gesuchten Werte sind Gegen- und Ankathete. Um diese zu ermitteln, werden folgende Formeln angewandt:

$$v_{0,x} = \cos(\alpha) * v_{0,ges}$$
$$v_{0,y} = \sin(\alpha) * v_{0,ges}$$

Formel 2: Berechnung der Anfangsgeschwindigkeit der Kreise
(Merzinger und Wirt 2006, F2)

Nachdem der Emitter diese Werte errechnet, fügt er die Kreise der Welt hinzu.

[5] Für weitere Informationen zur Kollisionserkennung und -behandlung in physikalischen Simulationen, siehe (Hahn 2000, S.11 f)

4.1.2 Physiklogik

Die Berechnungen für jeden Kreis in der Welt werden kontinuierlich durchgeführt. Dies wird über eine Schleife realisiert.

Da die Geschwindigkeit als zweidimensionaler Vektor betrachtet wird, müssen bei der folgenden Berechnung keine Winkel beachtet werden. Um eine nachvollziehbare Bewegung zu simulieren wurde die Schwerkraft (g) auf 9.81 m/s² gesetzt. Die Zeitabschnitte (dt) sind konstant.

4.1.2.1 Aktualisierung der Lebenszeit

Die Lebenszeit, die für jeden Kreis durch den Emitter gesetzt wird, wird in jedem Simulationszyklus dekrementiert. Ist diese abgelaufen, wird der Kreis aus der Simulation entfernt.

4.1.2.2 Positionsberechnung

Die aktuelle Position wird über die Formel des schiefen Wurfes berechnet:

$$x = v_{0,x} * dt$$

$$y = \frac{-1}{2} * g * dt^2 + v_{0,y} * dt$$

Formel 3: Berechnung der Anfangsposition der Kreise
(Kurzweil, Frenzel, und Gebhard 2009, S. 11)

4.1.2.3 Geschwindigkeitsberechnung

Für die Kollisionsbehandlung muss die aktuelle Geschwindigkeit der Objekte bekannt sein. Diese wird mit diesen Formeln ermittelt:

$$v_x = v_{0,x}$$

$$v_y = -g * dt + v_{0,y}$$

Formel 4: Berechnung der Geschwindigkeit der Kreise
(Kurzweil, Frenzel, und Gebhard 2009, S. 11)

4.1.2.4 Kollisionsbehandlung

Die Kollisionsbehandlung wurde in zwei Probleme unterteilt: die Erkennung und Behandlung von Kollisionen mit einer Weltbegrenzung, und die mit anderen Kreisen. Weltbegrenzungen sind dabei x- und y-Achsen des Systems, sowie Parallelen zu denselben in definiertem Abstand.

a) Kollision mit der Weltbegrenzung

   Die Kollision mit einer Weltbegrenzung wird über die x-/y-Koordinaten ermittelt. Diese triviale Methode bietet sich hierfür an, da nur mit parallelen Geraden zur x- und y-Achse gerechnet wird.

- Erkennung

  Eine Kollision findet statt, wenn der Abstand eines Kreises zur Weltbegrenzung geringer ist als deren Radius.

- Behandlung
  - Wenn eine Kollision stattgefunden hat, wird als erstes die Position des Kreises korrigiert. Diese muss dahingehend geändert werden, dass der Abstand zwischen Kreis und betreffender Gerade kleinstmöglich ist, also mindestens dem Radius des Kreises entspricht.
  - Der Kreis wird mit dem gleichen Winkel an der Weltbegrenzung abprallen, mit dem sie aufgetroffen ist (Halliday und Resnick 1994, S. 1261). Da der Winkel in den x-/y-Koordinaten der Geschwindigkeit enthalten ist, kann deren y-Koordinate mit -1 multipliziert werden:

    ($v_y' = -v_y$).

    Da die Begrenzung beim Aufprall Energie absorbiert, muss auf die neue Geschwindigkeit ein Restitutionskoeffizient (k) multipliziert werden

    ($v_y' = v_y * k$).
  - Abschließend wird ein neuer Wurf simuliert. Dabei werden die aktuelle Geschwindigkeit als Anfangsgeschwindigkeit und die aktuelle Position als neue Startposition angenommen. Mit diesen Daten kann dann im nächsten Schleifendurchlauf gerechnet werden.

b) Kollision mit anderen Kreisen

Um die Erkennung durchzuführen müssen die Abstände zwischen den Kreisen geprüft werden.

- Erkennung

  Eine Kollision findet statt, wenn der Abstand zwischen den Kreisen kleiner gleich der kombinierten Radien der Kreise ist (Kurzweil, Frenzel, und Gebhard 2009, S. 42). Die Distanz zwischen den Kreisen berechnet sich mit

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Formel 5: Distanz zwischen zwei Kreisen
(Merzinger und Wirt 2006, S. 48)

- Behandlung
  - Im ersten Schritt wird die Position der Kreise so korrigiert, dass der Abstand zwischen ihnen mindestens den kombinierten Radien entspricht. Dazu werden die Kreise, auf Basis ihrer Masse, von ihrer ursprünglichen Position bewegt (Conrod 2015). Auf die Distanz zwischen alter und neuer Position wird zusätzlich ein Sicherheitsfaktor von 0.1 gerechnet, um Ungenauigkeiten bei der Fließkommaberechnung auszugleichen.
  - Für den Rückstoß der beiden Kreise wird ein elastischer, zentraler und gerader Stoß angenommen. Die neue Geschwindigkeit der Kreise berechnet sich für jeden einzelnen aus:

$$v_1 = \left(\frac{m_1 - m_2}{m_1 + m_2} * v_1 + \frac{2 * m_2}{m_1 + m_2}\right) * v_2 * e_x$$
$$v_2 = \left(\frac{2 * m_1}{m_1 + m_2} * v_1 - \frac{m_1 - m_2}{m_1 + m_2}\right) * v_2 * e_x$$

Formel 6: Distanz zwischen zwei Kreisen
(Kurzweil, Frenzel, und Gebhard 2009, S. 43)

Da im letzten Schritt der Rechnung der Einheitsvektor von x aufmultipliziert wird, können v1 und v2 durch ihre jeweiligen x- und y-Anteile ersetzt werden ($v_{1,x}, v_{1,y}, v_{2,x}, v_{2,y}$).
(Kurzweil, Frenzel, und Gebhard 2009, S. 43)

  - Im letzten Schritt wird ebenfalls ein neuer Wurf simuliert. Die aktuelle Geschwindigkeit und Position werden, wie bei der Kollision mit einer Wand, als neue Startparameter gesetzt.

4.2 Parallelisierung

Der in dieser Arbeit verwendete Algorithmus zur Parallelisierung der Simulation konnte bei allen Parallelisierungstechniken eingesetzt werden. Dies fördert eine bessere Vergleichbarkeit der Varianten.
Im Folgenden wird die generelle Funktionsweise des Algorithmus und dann die Einzelheiten der jeweiligen Implementierung aufgezeigt.

4.2.1 Aktualisierung der Kreise

Die Berechnung der aktuellen Position und Geschwindigkeit, sowie die Kollisionserkennung mit Wänden werden zusammengefasst und für jeden Kreis nebenläufig berechnet. Dadurch kann einerseits die Abfolge der Schritte definiert werden, ohne für jeden Zwischenschritt zu synchronisieren, andererseits kann die Berechnungsmenge pro Task erhöht werden, was zu einer höheren Effizienz führen sollte. Da die Berechnung keinen Einfluss auf andere Kreise hat, erfordert dieser Schritt keine Synchronisation während der Berechnung. Nach diesem Schritt muss die Anwendung jedoch synchronisiert werden, damit die Kollisionserkennung zwischen Kreisen mit aktuellen Daten arbeitet.

4.2.1.1 Async (ASYNC-P-C)

Um die Aktualisierung jedes Kreises vornehmen zu können, wird mit Hilfe eines Iterators (Smith 2015, S. 850) jeder Kreis angesprochen.

```
1   for ( auto it = m_world->m_circles.begin(); it != m_world-
    >m_circles.end(); ++it ) {
2       std::future future = std::async( &Physic::updateAndWallCollision,
    this, std::ref(*it) );
3       results.push_back( std::move(future) );
4
5   }
6
7   for (auto it = results.begin(); it != results.end(); ++it) {
8       it->get();
9   }
10
11  results.clear();
```

Quelltextbeispiel 14 (C++): async – Aktualisierung der Kreise

Der Aufruf von *async()* innerhalb der Schleife (Z. 2) startet die asynchrone Bearbeitung der Aktualisierung. Durch die Übergabe des Parameters *this* wird *async()* der aktuelle Klassenkontext mitgeteilt, sodass auf Klassenvariablen und -funktionen zugegriffen werden kann. Da die Funktion eine Referenz auf das zu bearbeitende Objekt erwartet, wird zuerst der Iterator dereferenziert und dann über den Aufruf von *std::ref()* die Referenz auf das Objekt übergeben (Z. 2). Das zurückgegebene Future wird in eine Liste gespeichert, um später Zugriff auf das Ergebnis der Aktualisierung zu erhalten.

Nachdem für jeden Kreis die Aktualisierung gestartet wurde, wird das Programm synchronisiert. Dazu wird auf jedes Future die Methode *get()* gerufen.

Um die Liste wiederverwenden zu können, wird sie nach diesem Vorgang geleert (Z. 11). Da es sich um eine verkettete Liste handelt, muss deren Speicher jedoch neu reserviert werden.

4.2.1.2 Thread Pool (POOL-P-J)

Um die Aktualisierung der Kreise vorzunehmen wird zuerst ein Thread Pool erstellt. Dieser wird mit der festen Größe *vier* für die Anzahl der zur Verfügung stehenden Kerne dimensioniert. Um den Pool sowohl für die Aktualisierung, als auch für die Kollisionserkennung verwenden zu können, wird dieser global deklariert.

```
world.pool = Executors.newFixedThreadPool(4);
```

Quelltextbeispiel 15 (Java): Erstellung eines Thread Pools mit definierter Größe

Für die Aktualisierung der Kreise wird eine Lambda-Funktion erstellt, welche die Aktualisierungsfunktion aufruft. Die Lambdafunktion wird dem Thread Pool zur Bearbeitung übergeben (Z. 4). Der Aufruf von *submit()* gibt ein future zurück, das in einer Liste zur späteren Bearbeitung gespeichert wird.

```
1  for(int i = 0 ; i < world.circles.size(); ++i) {
2      Circle circle = world.circles.get(i);
3
4      Future<Integer> future = world.pool.submit( () -> {
5          updateAndWallCollision(circle);
6          return 0;
7      });
8
9      results.add(future);
10 }
```

Quelltextbeispiel 16 (Java): Thread Pool – Aktualisierung der Kreise

Um die Ergebnisse der Aktualisierung zu erlangen, wird jedes Ergebnis über den Aufruf *get()* abgerufen (Z. 3). Da der Aufruf so lange blockiert, bis das Ergebnis zur Verfügung steht, ist das Programm nach diesem Schritt wieder synchronisiert. Da der Aufruf von *get()* eine Exception werfen kann, muss dieser Schritt durch einen *try-catch*-Block (Oracle o.A. b) abgesichert werden.

```
1  for( Future result : results ) {
2      try {
3          result.get();
4      } catch (InterruptedException e) {
5          ...
6      }
7  }
```

Quelltextbeispiel 17 (Java): Thread Pool – Synchronisation und Fehlerbehandlung mit try/catch

4.2.1.3 Parallel Streams (STREAM-P-J)

Die Aktualisierung jedes Kreises wird in dieser Implementierung auf ein Stream-Element abgebildet. Zu Beginn wird der parallele Stream gestartet.

Das Herausfiltern von Kreisen, deren Lebenszeit abgelaufen ist, wird mit Hilfe des *filter()*-Operators durchgeführt, der nur Elemente weitergibt, die eine positive Lebenszeit aufweisen.

Die Aktualisierung der Kreise und die Kollisionserkennung mit Wänden wird in einem *map()*-Operator durchgeführt. Dieser bildet ein Eingangselement (p) auf ein Ausgangselement ab, welches am Ende der Funktion zurückgegeben wird.
Wird eine Kollision erkannt, wird ein Flag gesetzt und dementsprechend ein neuer Wurf veranlasst.

Da während der Aktualisierung einzelne Kreise aus der Welt entfernt werden, entspricht die Eingabemenge des bearbeitenden Streams nicht mehr dessen Ausgabemenge. Daher muss die Liste der Kreise aktualisiert werden. Dazu wird der Stream mit einem *collect()* Operator in eine Liste umgewandelt. Die entstehende Liste ersetzt die Originalliste. Das Umwandeln des Streams in eine Liste erfolgt blockierend, der *collect()* Operator wartet also so lange, bis alle Elemente des Streams ihre Ausführung beendet haben. Dadurch ist das Programm nach Beendigung des *collect()* Operators synchronisiert.

```
1   world.circles = world.circles.parallelStream()
2       .filter(p -> p.lifeTime-- >= 0)
3       .map(p -> {
4           p.dt += world.timeStep;
5
6           // Aktualisierung
7           ...
8
9           // Kollisionsbehandlung
10          ...
11
12          // neuer Wurf
13          if( p.collision ) {
14              ...
15          }
16
17          return p;
18  })
19  .collect(Collectors.toList());
```

Quelltextbeispiel 18 (Java): Stream – Aktualisierung der Kreise

4.2.1.4 Goroutines (GO-P-G)

Die Aktualisierung jedes Kreises wird in dieser Implementierung über eine Goroutine durchgeführt. Diese erhält neben der Referenz auf die zu bearbeitende Kreise auch einen Kanal, der zur Synchronisation genutzt wird.

Die Synchronisationskanäle werden in einer eigenen Liste gespeichert. Dies ist nötig, da innerhalb der Physikschleife Kreise, deren Lebenszeit abgelaufen ist, aus der ursprünglichen Liste entfernt werden. Dadurch ändert sich während der Iteration die Größe der Kreisliste. Dies bedingt, dass vor Ausführung der Schleife nicht vorhergesagt werden kann, wie viele Kreise tatsächlich aktualisiert werden müssen und damit, wie viele Kommunikationskanäle erforderlich sind. Daher wird diese Liste dynamisch zur Laufzeit erstellt (Z. 6).

Mit ihrer Hilfe kann eine einfache Synchronisationsbarriere implementiert werden. Auf jedem Kanal wird die Leseoperation aufgerufen, die solange blockiert, bis ein Wert gelesen wurde (Z. 12-13).

```
1   for index := 0; index < len(p.World.Circles); index++ {
2       circle := &p.World.Circles[index]
3
4       ...
5
6       channelGroup1 = append( channelGroup1, make(chan bool) )
7
8       channelIndex := len(channelGroup1)-1
9       go p.UpdateAndWallCollision(circle, channelGroup1[channelIndex])
10  }
11
12  for i := range channelGroup1 {
13      <- channelGroup1[i]
14  }
```

Quelltextbeispiel 19 (Go): Goroutinen – Aktualisierung der Kreise

Innerhalb der Aktualisierungsfunktion wurde die Kollisionserkennung mit den Weltgrenzen, aus Gründen der Übersichtlichkeit, in eine eigene Funktion ausgelagert (Z. 4).

Endet die Aktualisierungsfunktion, sendet sie über den Synchronisationskanal einen Booleanwert als Signal (Z. 6). Dieser kann dann in der Synchronisationsbarriere gelesen werden.

```
1   func (p* Physic) UpdateAndWallCollision(c* Circle, syncChannel chan
    bool) {
2       ...
3
4       p.WallCollisionHandling(circle)
5
```

```
6       syncChannel <- true
7
8       return
9   }
```

Quelltextbeispiel 20 (Go): Goroutinen – Kommunikation über Kanäle

4.2.2 Kollisionserkennung

Die Kollisionserkennung und -behebung wird in zwei Funktionen aufgeteilt. Grund hierfür ist die zunehmende Komplexität der Synchronisierung, da vor allem die Kollisionsbehebung immer mit korrekten Daten arbeiten muss.

a) Die Kollisionserkennung kann unabhängig für jeden Kreis berechnet werden. Wird eine Kollision zwischen zwei Kreisen erkannt, werden diese in einer separaten Liste als Paar zur späteren Bearbeitung gespeichert. Der Zugriff auf diese Liste muss synchronisiert werden, da er schreibend erfolgt. Da die Erkennung asynchron durchgeführt wird, ist es möglich, dass die selben Paare in unterschiedlichen Permutationen erkannt und gespeichert werden. Da zum Zeitpunkt der Erkennung jedoch nicht bekannt ist, ob bereits alle möglichen Permutationen aufgetreten sind, muss dieses Problem in der Kollisionsbehandlung behoben werden.

   Nach diesem Schritt muss das Programm ebenfalls synchronisiert werden.

b) Die Kollisionsbehandlung arbeitet die von der Kollisionserkennung gefundenen Paare ab. Dabei wird zuerst überprüft, ob das Paar in einer seiner möglichen Permutationen bereits abgearbeitet wurde. Dieser Schritt erfordert jedoch eine sequentielle Abarbeitung der Daten, da ansonsten nicht ausgeschlossen werden kann, dass ein Paar doppelt behandelt wird.

   Wurde das Paar noch nicht bearbeitet, wird, wie in der sequentiellen Lösung, die Kollision behoben. Anschließend wird es als bearbeitet markiert.

Im Folgenden werden die einzelnen Implementierungen vorgestellt. Die gezeigten Codebeispiele wurden auf wesentliche Funktionsaufrufe und Parallelisierungsoperationen gekürzt. Der vollständige Quellcode findet sich im Anhang.

4.2.2.1 Thread Pool

Wie auch bei der Aktualisierung der Kreise wird dem bereits existierenden Thread Pool eine Lambda-Funktion zur Bearbeitung übergeben (Z. 5), die die eigentliche Kol-

lisionserkennung aufruft. Das zurückgegebene Future wird in einer Liste gespeichert (Z. 10).

```
1   for(int i = 0 ; i < world.circles.size(); ++i) {
2       Circle circle = world.circles.get(i);
3       ...
4
5       Future<Integer> future = world.pool.submit( () -> {
6           circleCollisionDetection(circle);
7           return 0;
8       });
9
10      results.add(future);
11  }
```

Quelltextbeispiel 21 (Java): Thread Pool – Kollisionserkennung

Nach diesem Schritt wird das Programm synchronisiert.

```
1   for( Future result : results ) {
2       try {
3           result.get();
4       } catch (InterruptedException e) {
5           ...
6       }
7   }
```

Quelltextbeispiel 22 (Java): Thread Pool – Synchronisierung und Fehlerbehandlung

4.2.2.2 Async

Die Kollisionserkennung zwischen Kreisen wird ebenfalls asynchron für jeden Kreis ausgeführt werden. Das Starten der asynchronen Funktionen funktioniert prinzipiell wie im oberen Beispiel:

```
 results.push_back( std::async( &Physic::circleCollisionDetection, this,
std::ref(*it) ) );
```

Quelltextbeispiel 23 (C++): Async – Kollisionserkennung

Nachdem alle Aufrufe gestartet werden, muss das Programm auch nach diesem Schritt wie oben synchronisiert werden.

Anschließend wird die Kollisionsbehandlung durchgeführt und alle angefallenen Daten bereinigt.

```
circleCollisionResolution();

alreadyDone.clear();
collisionPairs.clear();
```

Quelltextbeispiel 24 (C++): async – Kollisionsbehandlung und Bereinigung

4.2.2.3 Parallel Streams

Auch die Kollisionserkennung zwischen Kreisen kann asynchron für jeden Kreis ausgeführt werden. Der Stream wird wie im obigen Beispiel gestartet.

Da die Kollisionserkennung keine Werte zurückliefert, sondern die erkannten Kollisionspaare in eine globale Liste speichert, wird die Erkennung mit Hilfe eines *forEach()*-Operators durchgeführt (Z. 2). Der schreibende Zugriff auf die Liste muss nicht manuell synchronisiert werden, da es sich hierbei um eine Thread-sichere CopyOnWriteArrayList aus der java.util.concurrent-Bibliothek handelt.

Der *forEach()*-Operator beendet zusätzlich den Stream nach Ausführung aller Elemente (Z. 1).

```
1   world.circles.parallelStream().forEach( p -> {
2       for(Circle oCircle : world.circles){
3           if( oCircle != p) {
4               Vector2d delta = p.pos.subtract(oCircle.pos);
5
6               int radiiCombined = oCircle.radius + p.radius;
7               double distanceSquared = delta.x * delta.x + delta.y *
    delta.y;
8
9               if( distanceSquared <= radiiCombined * radiiCombined) {
10                  collisionPairs.add(new CollisionPair(p, oCircle));
11              }
12          }
13      }
17  });
```

Quelltextbeispiel 25 (Java): Stream – Kollisionserkennung

Die Kollisionsauflösung wird, wie in der Implementierung der *async*-Lösung sequentiell durchgeführt.

```
resolveCircleCollision();

alreadyDone.clear();
collisionPairs.clear();
```

Quelltextbeispiel 26 (Java): Stream – Kollisionsbehandlung und Bereinigung

4.2.2.4 Goroutinen

Die Kollisionserkennung wird, wie die Aktualisierung der Kreise, asynchron durchgeführt. Zusätzlich zu einem Synchronisationskanal kommuniziert die Kollisionserkennung über einen zweiten Kanal mit der aufrufenden Funktion.

Der Synchronisationskanal wird beschrieben, wenn die Funktion beendet (Z. 20).

Wird eine Kollision erkannt, wird das Kollisionspaar nicht wie bei den anderen Lösungen in eine Liste gespeichert, sondern in den zusätzlichen Kanal geschrieben (Z.

15). Dies ermöglicht der aufrufenden Funktion, auf diesem Kanal zu lauschen und darüber gesendete Kollisionspaare sofort zu behandeln. Dadurch kann eine Synchronisierungsbarriere eingespart werden, was die Ausführung des Programmes beschleunigen sollte.

```
1   func (p* Physic) CircleCollisionDetection(circle* Circle, cPChannel
    chan *CollisionPair, done chan bool) {
2
3       for index := 0; index < len(p.World.Circles); index++ {
4           oCircle := &p.World.Circles[index];
5
6           if oCircle != circle {
7               delta := circle.Pos.Subtract(oCircle.Pos)
8
9               radiiCombined := circle.Radius + oCircle.Radius;
10              distanceSquared := delta.X*delta.X + delta.Y*delta.Y
11
12              if (distanceSquared <= radiiCombined * radiiCombined) &&
    (distanceSquared != 0) {
13
14                  cp := NewCollisionPair(circle, oCircle)
15                  cPChannel <- cp
16              }
17          }
18      }
19
20      done <- true
21
22      return
23  }
```

Quelltextbeispiel 27 (Go): Goroutinen – Kollisionserkennung

Nachdem alle asynchronen Funktionsaufrufe für die Kollisionserkennung gestartet wurden, lauscht die aufrufende Funktion auf dem Synchronisationskanal und dem Kanal, über den gefundene Paare transportiert werden. Dies wird über den Operator *select* durchgeführt. Dieser lauscht gleichzeitig auf allen ihm genannten Kanälen (Z. 4). Wird auf einem Kanal etwas empfangen, wird der entsprechende Codestrang ausgeführt.

Im Falle einer erkannten Kollision wird das gelesene Kollisionspaar an die Kollisionsbehebung übergeben (Z. 9). Diese behandelt die Kollision wie die Implementierung der anderen Lösungen.

Wird auf dem Synchronisierungskanal ein Signal empfangen, ist die Kollisionserkennung abgeschlossen. Über einen Zähler wird geprüft, ob alle asynchron gestarteten Aufrufe beendet wurden und somit das Programm wieder synchron ist.

```
1   countDone := 0
2
3   for countDone < len(p.World.Circles) {
```

```
4       select {
5           case <- done:
6               countDone++
7               continue
8           case cp := <- cPChannel:
9               p.resolveCircleCollision( cp )
10      }
11  }
```

Quelltextbeispiel 28 (Go): Goroutinen – Kollisionsbehandlung mithilfe von Abfragen auf mehrere Kanäle

5 Planung und Durchführung der Datenerhebung

Im folgenden Kapitel wird herausgearbeitet, welche Merkmale der Implementierungen für einen Vergleich in den Bereichen Leistung und Bedienbarkeit herangezogen werden können. Anschließend wird dargelegt, mit welchen Techniken und unter welcher Umgebung die Daten erhoben werden.

5.1 Vergleichsmerkmale

Die Merkmale, die für diesen Vergleich zwischen den Implementierungen erhoben werden sollen, werden im Folgenden vorgestellt. Diese lassen sich in Merkmale zum Vergleich der Leistungsfähigkeit und in Merkmale zum Vergleich der Bedienbarkeit einer Parallelisierungstechnik untergliedern.

5.1.1 Merkmale zum Leistungsvergleich

Die Leistung von Parallelisierungstechniken kann mittels verschiedener Merkmale verglichen werden. Dazu gehören z.B. die Ausführungsgeschwindigkeit des parallelisierten Programmes, der Beschleunigungsfaktor zwischen sequentieller und paralleler Implementierung und die CPU-Auslastung der Parallelisierungstechnik.

5.1.1.1 Ausführungszeiten und Beschleunigungsfaktor

Die reine Ausführungszeit der Simulationen kann nur zwischen sequentieller und paralleler Implementierung verglichen werden, und nicht zwischen den Implementierungen mit den verschiedenen Parallelisierungstechniken. Grund hierfür sind die verschiedenen Eigenschaften der genutzten Programmiersprachen, wie z.B. die Art der Ausführung des Programmes, die nicht in den Vergleich zwischen den Parallelisierungstechniken einfließen sollen.

Daher wird auf einen prozentualen Beschleunigungsfaktor zwischen den sequentiellen und parallelen Implementierungen zurückgegriffen. Dieser ist unabhängig von der jeweiligen Sprache und Messtechnik. Er gibt an, wie stark die jeweilige Parallelisierungstechnik die Berechnung der Simulation beschleunigen kann. Ein positiver Wert bedeutet eine Verbesserung der Ausführungsgeschwindigkeit der parallelen Implementierung im Vergleich zur sequentiellen, negative Werte zeigen an, dass die Parallelisierung der Anwendung Geschwindigkeitseinbußen bei der Ausführung der Simulation hervorrufen.

Für die Berechnung des Beschleunigungsfaktors (Formel 7) werden in der sequentiellen (sv) und parallelen (pv) Variante der Simulation die Ausführungszeiten für jeden Zyklus erfasst. Über die Ausführungszeiten jedes Zyklus wird dann der Median gebildet, um zu vermeiden, dass Grenzwerte, wie extrem niedrige oder hohe Ausführungszeiten in die Berechnung einfließen. Der Beschleunigungsfaktor wird nur zwischen sequentiellen und parallelen Implementierungen gebildet.

$$B_a = 100 - \frac{100 * t_{pv}}{t_{sv}}$$

Formel 7: Berechnung des Beschleunigungsfaktors

Der Median der Ausführungszeiten wird zusätzlich genutzt, um die Implementierungen einer Parallelisierungstechnik genauer zu vergleichen. Dazu wird die Differenz der Ausführungszeiten berechnet. Für diese gilt:

$$t_a = \tilde{t_{sv}} - \tilde{t_{pv}}$$

Formel 8: Berechnung der Ausführungszeit

5.1.1.2 CPU-Auslastung

Um zu erfassen, ob die parallele Version der Simulation den Prozessor möglichst optimal ausnutzt, soll dessen Auslastung gemessen werden. Für eine möglichst detaillierte Betrachtung soll sowohl dessen Gesamtauslastung, als auch die Auslastung jedes einzelnen Kernes gemessen werden.

5.1.2 Merkmale zum Vergleich des Bedienaufwandes

Der Mehraufwand, der durch Parallelisierung einer Anwendung entsteht, kann mittels der Komplexität[6] des Quelltextes der parallelisierten Implementierungen gemessen werden, die dann mit der Komplexität der Referenzimplementierung verglichen wird.

5.1.2.1 Code-Komplexität

Um die Komplexität des Quellcodes der parallelisierten Implementierung zu messen, und einen Vergleich zur sequentiellen Referenzimplementierung herzustellen, wurden folgende, statische, analytische Möglichkeiten in Erwägung gezogen (Microsoft Corporation o.A. c):

a) Die Anzahl an Instruktionen, die für die Simulation notwendig sind: ausgehend von der sequentiellen Version der Simulation soll ermittelt werden, wie viele zusätzliche Instruktionen nötig sind, um die Parallelisierung vorzunehmen. Die

[6] *Komplexität* bezeichnet die "Gesamtheit aller voneinander abhängigen Merkmale und Elemente, die in einem vielfältigen aber ganzheitlichen Beziehungsgefüge (System) stehen" (Feess o.A.). Der Begriff kann auch als Intransparenz aus Sicht des Betrachters angesehen werden.

Komplexität eines Programmes wird also durch die Anzahl der verwendeten Anweisungen definiert. Dieser Wert liefert einen sehr genauen Indikator für die Zunahme der Instruktionen und damit der Komplexität, er ist jedoch sehr aufwendig zu erfassen. Zudem muss definiert werden, welche Ausdrücke als Instruktion gewertet werden, und welche nur Teil einer solchen sind.

b) Die Anzahl der Code-Zeilen des Programmes: Dieser Messwert nutzt die Programmierkonvention, nur eine Anweisung pro Zeile zu verwenden, um die Anzahl an genutzten Anweisungen zu ermitteln. Wie auch in a) wird die Komplexität über die Anzahl von verwendeten Instruktionen definiert. Ein Vorteil dieser Methode ist der, im Vergleich zu a) geringe Durchführungsaufwand einer Messung. Nachteilig ist jedoch die relativ ungenaue Messung, wenn z.B. die genannte Konvention missachtet, oder wenn mehrere Anweisungen in einer Zeile zusammengefasst werden müssen.

c) Die zyklische Komplexität: Dieser Wert erfasst die Komplexität des Programmes anhand der logischen Verzweigungen. Je mehr logische Verzweigungen ein Programm enthält, desto komplexer wird das Programm angesehen (McCabe 1976).

Die Synchronisierung des Programmes ist ein Schlüsselproblem bei der Parallelisierung und bestimmt daher deren Komplexität maßgeblich. Dafür sind jedoch keine, oder wenige, komplexen logischen Verzweigungen innerhalb des Programmes nötig. Die zyklische Komplexität liefert daher keinen verlässlichen Wert für die Komplexität.

Da die Anzahl an genutzten Instruktionen, wegen fehlender Werkzeuge nicht bei allen Programmiersprachen ermittelt werden konnte, wurde die Anzahl der Codezeilen als Metrik für die Komplexität gewählt. Hierbei werden jedoch Leerzeilen und Kommentare ignoriert, da deren Anzahl zwischen den verschiedenen Implementierungen sehr stark variiert.

Alle Parameter, die während der Ausführung der Simulation erfasst werden müssen, werden in Log-Dateien archiviert, um sie später auswerten zu können.

5.2 Eingesetzte Messtechniken

Nachfolgend werden die Techniken vorgestellt, mit denen die Ausführungszeiten und Prozessorauslastung gemessen werden. Zusätzlich werden Kriterien für die Wiederholbarkeit der Messungen aufgestellt.

5.2.1 Techniken zur Zeitmessungen

Die Zeitmessungen werden mit den, in der jeweiligen Sprache zur Verfügung stehenden Werkzeugen durchgeführt. Diese erlauben nanosekundengenaue Zeitmessungen. Am abgebildeten Quelltextbeispiel 29 aus der Implementierung in Go wird exemplarisch erläutert, wie diese erfolgen.

Für eine Zeitmessung wird jeweils ein Zeitpunkt vor Ausführung des aktuellen Simulationszyklus, sowie ein Zeitpunkt nach dessen Ausführung gemessen (Z. 1 und 5). Durch Subtraktion der beiden Zeitpunkte kann die vergangene Zeit ermittelt werden (Z. 6).

```
1  start := time.Now().UnixNano()
2
3  physic.PhysicLoop()
4
5  end := time.Now().UnixNano()
6  ellapsedTime := end - start
```

Quelltextbeispiel 29 (Go): Berechnung der Dauer eines Simulationszyklus

Die genutzten Werkzeuge sind:

- std::chrono::high_resolution_clock (C++) (cppreference.com o.A. a)
- System.nanoTime() (Java) (Oracle o.A. m)
- time.Now().UnixNano() (Go) (Google Inc. o.A. f)

5.2.2 Technik zur Messung der Prozessorbelastung

Die Auslastung des Prozessors wird mit Hilfe des Linux-Kommandozeilenprogrammes *mpstat* gemessen (Godard 2015). Dieses erlaubt, die Auslastung jedes einzelnen Prozessorkerns, sowie die Gesamtauslastung des Prozessors zu ermitteln. Die Messungen werden sekündlich, dem geringstmöglichen Intervall, durchgeführt.

5.2.3 Wiederholbarkeit und Vergleichbarkeit der Messungen

Um eine korrekte Wiederholbarkeit der Messungen zu ermöglichen, wird auf den Gebrauch von Zufallszahlen im Emitter verzichtet. Diese könnten bei jedem Programmdurchlauf variieren, wodurch die Physikberechnung mit unterschiedlichen Parametern ausgeführt würde, was die Anzahl an zu berechnenden Kreisen, und damit die Anzahl der Berechnungen, verändern könnte.
Die Vergleichbarkeit der verschiedenen Varianten wird durch einige Faktoren beeinflusst:

a) Umwelteinflüsse auf den Versuchsrechner: Da die Kühlung des Prozessors wesentlichen Einfluss auf dessen Leistung hat (Intel Corporation 2015), sollte für den Versuch konstante Raumtemperatur herrschen. Außerdem sollte der Messvorgang immer mit derselben Anfangstemperatur des Versuchsrechners gestartet werden.
b) Softwarestand des Versuchsrechners: Die Version der verwendeten Programme kann Einfluss auf die Leistung der Implementierung nehmen. Daher werden in dieser Arbeit die Versionen der wesentlichen Programme (JavaVM, Go Runtime, GCC-Compiler) angegeben (s. 5.3).
c) Ungenauigkeit der Zeitmessung: *System.nanoTime()* garantiert keine nanosekundengenaue Genauigkeit der Zeitstempel. Die berechnete Ausführungszeit der Simulation kann daher Ungenauigkeiten im Nanosekundenbereich aufweisen. Da die Ausführungszeiten jedoch nur für die Berechnung des prozentualen Beschleunigungswertes zwischen den Java-basierten Implementierungen genutzt wird, können die Werte genutzt werden (Oracle o.A. m).
d) Rundungsfehler bei Operationen mit Fließkommazahlen: die genutzten Programmiersprachen zeigen teilweise unterschiedliches Verhalten bei Rechenoperationen mit Fließkommazahlen. Dies beeinflusst jedoch nur die Genauigkeit der Physiksimulation und hat keinen Einfluss auf die Zeitmessungen, da diese mit Ganzzahlen berechnet werden.

5.3 Messumgebung und Simulationsparameter

Alle Messungen wurden auf einem PC mit folgender Konfiguration durchgeführt:

CPU	Intel Core i5-2400 (Intel Corporation o.A. a) • 4 physikalische Kerne • max. 4 parallele Threads • 3,1 – 3,4 GHz Taktfrequenz
Arbeitsspeicher	7.7 GB DDR3
Betriebssystem	Ubuntu 14.04.2 LTS 64 Bit (Canonical Ltd o.A.), Kernel 3.13.0-61-generic
Softwarestand	Java VM 1.8.0_45, GCC 4.8.4, Go Runtime 1.2.1

Tabelle 4: Messumgebung

Änderungen an dieser Umgebung sollten vor allem Einfluss auf den Beschleunigungsfaktor nehmen. Je höher die Anzahl der Prozessorkerne, desto höher sollte dieser Wert liegen.

5.3.1 Ausgangswerte der Simulation

Für die Simulation werden folgende konstante Ausgangsparameter gewählt:

Größe der Zeitschritte	0.016
Gravitation	9.81 m/s²
Reibungskoeffizient der Weltgrenzen	0.1
Position der Emitter	An der oberen Weltgrenze. Der Abstand zu den vertikalen Weltgrenzen und zum nächsten Emitter beträgt jeweils 200 Pixel.
Startgeschwindigkeit der Kreise	100 m/s
Startwinkel der Kreise	Zwischen 70° und 110°. Wird nach jedem Austoßvorgang um 5° erhöht. Wird der Winkel inklusive 110° erreicht, wird er wieder auf 70° gesetzt.
Masse der Kreise	10 kg
Radius der Kreise	10 Pixel
Reibungskoeffizient der Kreise	0.1
Lebensdauer der Kreise	3.5000 Zyklen

Tabelle 5: konstante Simulationsparameter

Für einzelne Messungen werden folgende Parameter in der jeweiligen Simulation angepasst.

Die Berechnungsdauer eines einzelnen Simulationsschrittes ist abhängig von der Anzahl der Kreise innerhalb der Simulation ist. Daher die Anzahl der Emitter und die Lebenszeit der Kreise so angepasst, dass die Anzahl der Kreise während der Simulation möglichst konstant ist. Diese steigt während der Simulation zunächst linear an, um dann eine Sättigung zu erreichen. Die Anzahl der Kreise schwankt dann um maximal 7 Kreise. Für die verschiedenen Messungen wurden folgende Parameter eingesetzt:

5.3.1.1 Messung der Prozessorauslastung

Die Messung der Prozessorauslastung wurde mit zwei verschiedenen Konfigurationen durchgeführt, die verschiedene Kreiszahlen erzeugen.

a) ~ 490 Kreise

Anzahl der Simulationszyklen	30.000
Größe der Welt (Breite x Höhe)	2000 x 2000 Pixel
Anzahl der Emitter	7
Anzahl der Kreise	~ 490

Tabelle 6: variable Simulationsparameter: Messung der Prozessorauslastung ~ 490 Kreise

b) ~ 4130 Kreise

Anzahl der Simulationszyklen	30.000
Größe der Welt (Breite x Höhe)	12000 x 12000 Pixel
Anzahl der Emitter	59
Anzahl der Kreise	~ 4130

Tabelle 7: variable Simulationsparameter: Messung der Prozessorauslastung ~4130 Kreise

5.3.1.2 Zeitmessungen

Für die Durchführung der Zeitmessungen wurden drei verschiedene Konfigurationen getestet, die eine unterschiedliche Anzahl an Kreisen in der Simulation erzeugen.

a) ~ 490 Kreise

Anzahl der Simulationszyklen	30.000
Größe der Welt (Breite x Höhe)	2000 x 2000 Pixel
Anzahl der Emitter	7
Anzahl der Kreise	~ 490

Tabelle 8: variable Simulationsparameter: Zeitmessung ~ 490 Kreise

b) ~ 2030 Kreise

Anzahl der Simulationszyklen	30.000
Größe der Welt (Breite x Höhe)	6000 x 6000 Pixel
Anzahl der Emitter	29
Anzahl der Kreise	~ 980

Tabelle 9: variable Simulationsparameter: Zeitmessung ~ 2030 Kreise

c) ~ 4130 Kreise

Anzahl der Simulationszyklen	30.000
Größe der Welt (Breite x Höhe)	12000 x 12000 Pixel
Anzahl der Emitter	59
Anzahl der Kreise	~ 4130

Tabelle 10: variable Simulationsparameter: Zeitmessung ~ 4130 Kreise

6 Präsentation und Diskussion der Messergebnisse

Das folgende Kapitel beantwortet die Frage nach den Unterschieden zwischen den verschiedenen Implementierungen. Dazu werden die Daten, die während der Ausführung der Simulationen aufgezeichnet wurden, präsentiert und interpretiert.

6.1 Ressourcennutzung

Nachfolgend wird die Prozessorauslastung der verschiedenen Implementierungen vorgestellt und anschließend gegenseitig in Relation gesetzt.

6.1.1 Thread Pool

Die sequentielle Implementierung des Thread Pools, POOL-R-J, zeigt sehr starke Schwankungen in der Prozessorauslastung, was Abbildung 2 verdeutlicht.

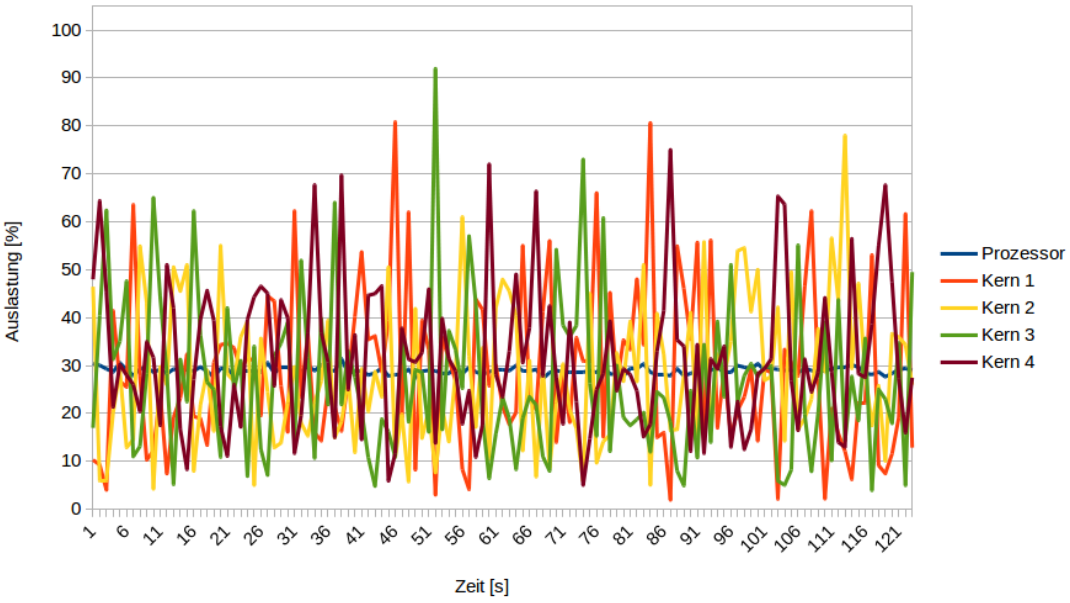

Abbildung 2: Prozessorauslastung POOL-R-J

Während der Ausführung wird meist ein Kern sehr stark ausgelastet. Die Gesamtauslastung des Prozessors liegt durchschnittlich bei ca. 30 Prozent.

Die Schwankungen in der Auslastung werden durch häufiges Scheduling des ausführenden Threads hervorgerufen, was auf eine ineffiziente Ressourcennutzung hindeutet.

Die parallelisierte Implementierung weist, wie in Abbildung 3 ersichtlich, deutlich geringere Schwankungen in der Auslastung auf.

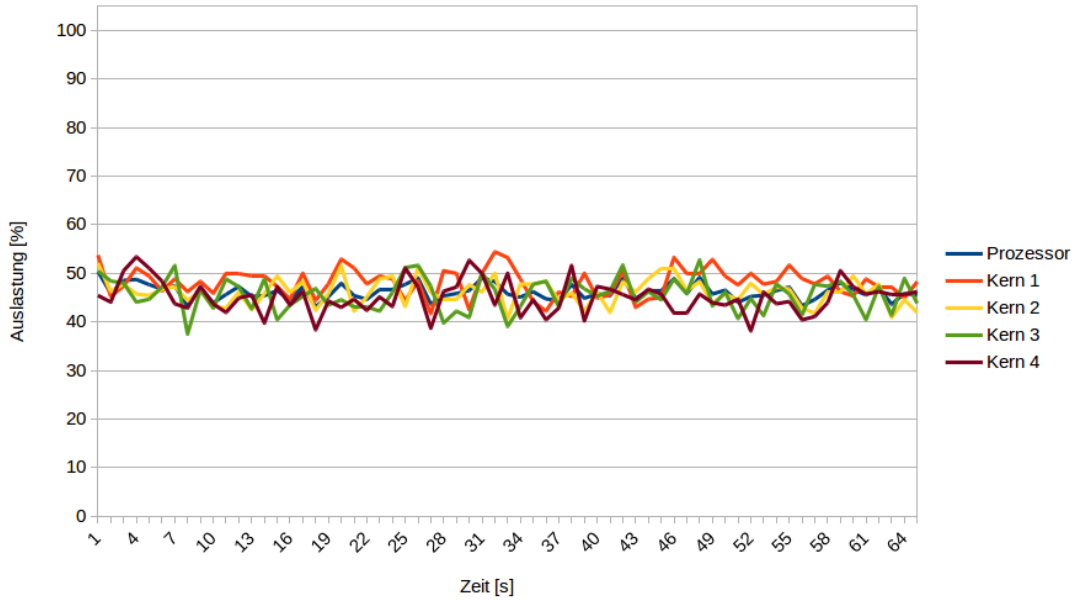

Abbildung 3: Prozessorauslastung POOL-P-J

Ein Grund dafür ist die im Vergleich zu POOL-R-J gesunkene Belastung pro Kern, die durch die Verteilung der Berechnungen auf alle Prozessorkerne entsteht. Die Gesamtauslastung steigt im Vergleich zu POOL-R-J auf ca. 50 Prozent.

6.1.2 C++ 11 – async

Die Prozessorauslastungen von ASYNC-R-C und ASYNC-P-C weisen im wesentlichen keine Unterschiede auf, wie die unten abgebildeten Abbildungen 4 und 5 zeigen.

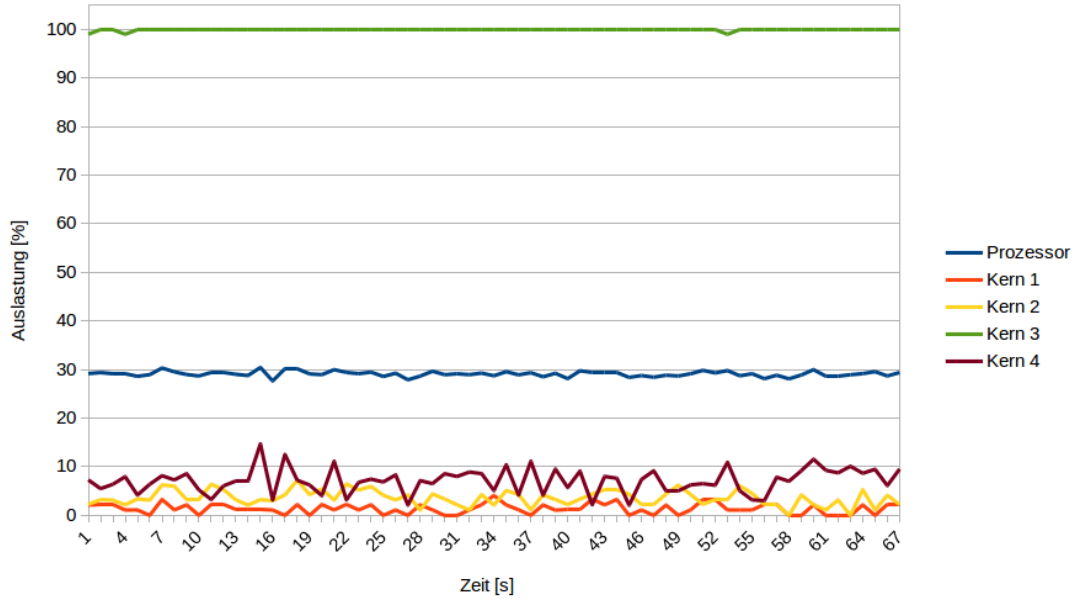

Abbildung 4: Prozessorauslastung ASYNC-R-C

Sowohl die sequentielle, als auch die parallele Implementierung erzeugen nahezu 100 Prozent Prozessorlast auf einem Kern, sowie eine durchschnittliche Gesamtauslastung des Prozessors von unter 30 Prozent.
Im Vergleich zu ASYNC-R-C schwankt die Prozessorauslastung unter ASYNC-P-C jedoch stärker und wird einmal von einem Kern 4 auf Kern 3 verschoben.

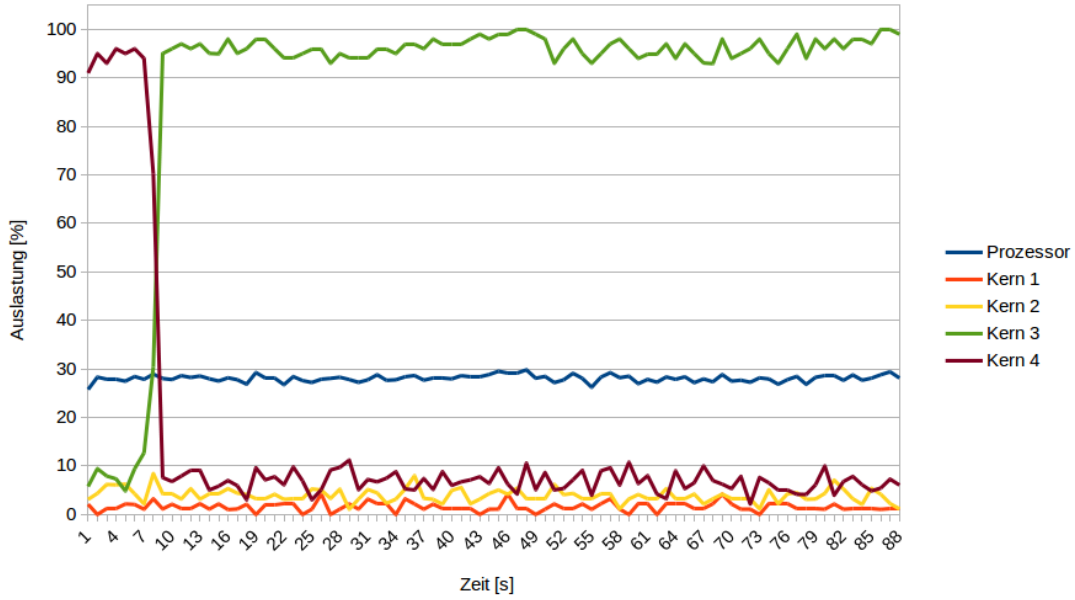

Abbildung 5: Prozessorauslastung ASYNC-P-C

Die schlechte Prozessorauslastung der parallelen Implementierung lässt sich mit dem Verhalten des GCC-Compilers unter Linux erklären. Die eingesetzte Version 4.8.4 setzt als voreingestelltes Startverhalten für async den Parameter *std::launch::deferred*. Dieser besagt, dass ein asynchroner Aufruf nicht in einem separaten Thread, sondern im Thread des Aufrufers ausgeführt wird (cppreference.com 2013), was die Nutzung mehrerer Prozessorkerne, und damit Parallelisierung, verhindert.

6.1.3 Java Parallel Streams

Beim Vergleich zwischen sequentieller Referenzimplementierung (Abbildung 2) und paralleler Implementierung (Abbildung 7) zeigt sich, dass die Parallelisierungstechnik eine deutliche Verbesserung der Prozessorauslastung bewirkt.
Zusätzlich zum Vergleich mit der Referenzimplementierung wird auch die nebenläufig Implementierung STREAM-N-J betrachtet, deren Prozessorauslastung in Abbildung 6 dargestellt ist, um zu analysieren, ob eine nebenläufige Implementierung ähnliche Ergebnisse erzielt, wie die parallele Implementierung.

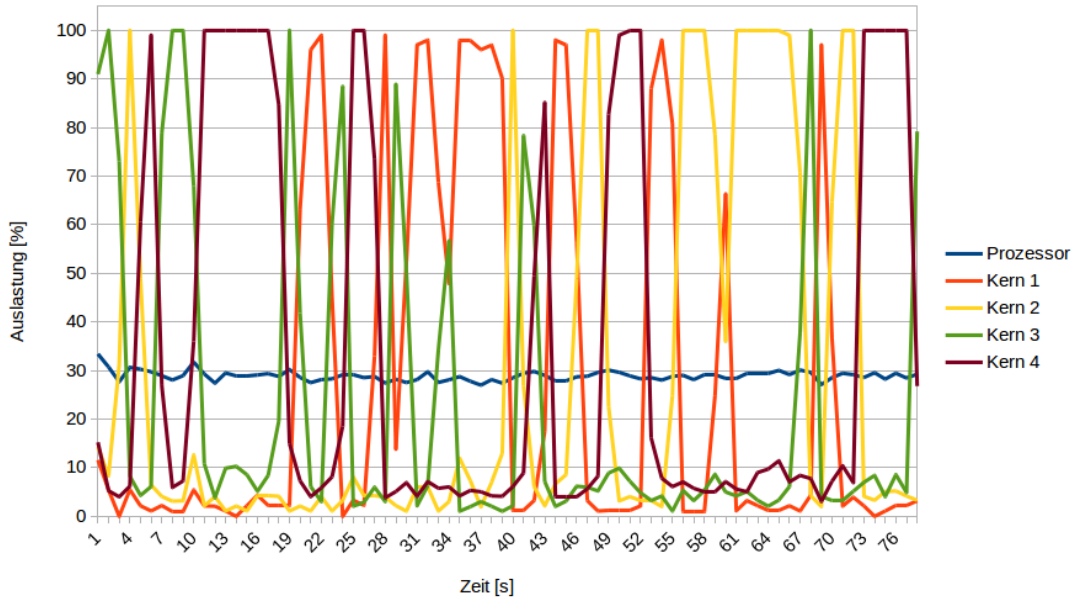

Abbildung 6: Prozessorauslastung STREAM-N-J

In Abbildung 6 zeigt sich, dass die Prozessorauslastung jeweils eines Kernes sprunghaft auf bis zu 100 Prozent ansteigt, um dann wieder abzufallen. Anschließend wiederholt sich das Muster meist bei einem anderen Kern. Dies deutet darauf hin, dass der Prozess in dem die Java Virtual Maschine, und damit die Berechnung von STREAM-N-J, läuft, häufig zwischen verschiedenen Prozessoren verschoben wird. Dies beeinflusst die Prozessorauslastung aufgrund des entstehenden hohen Verwaltungsaufwandes negativ. Die Gesamtbelastung des Prozessors liegt, wie auch bei der sequentiellen Referenzimplementierung durchschnittlich bei knapp 30 Prozent.

Aus Abbildung 7 wird ersichtlich, dass STREAM-P-J dagegen eine konstant hohe Auslastung aller Prozessorkerne erzeugt. Die Gesamtauslastung des Prozessors liegt bei über 80 Prozent und damit weit über dem Wert der sequentiellen und nebenläufigen Implementierung.

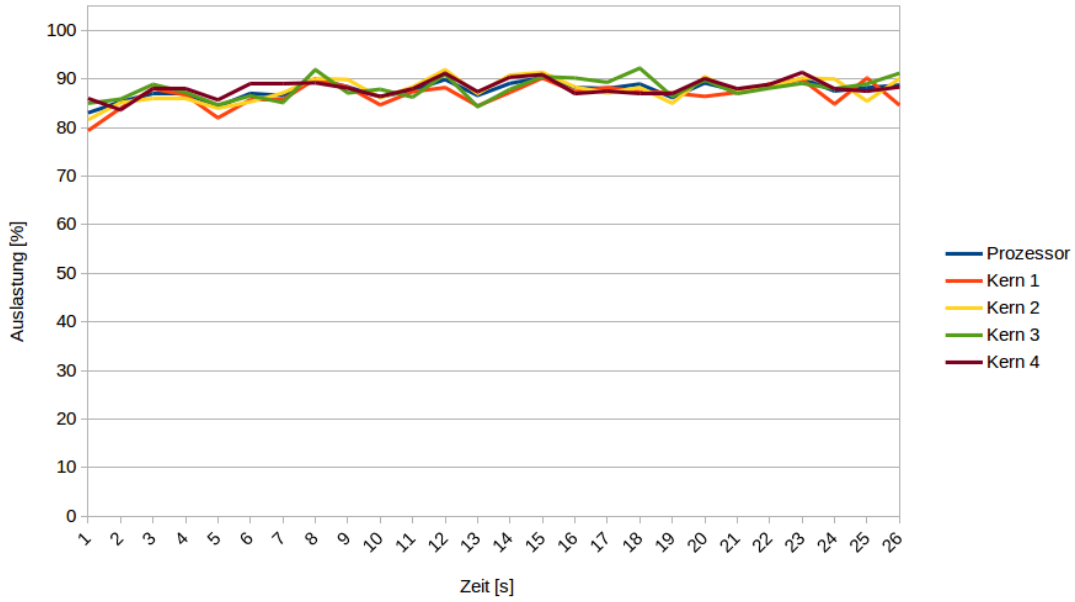

Abbildung 7: Prozessorauslastung STREAM-P-J

Die konstante Auslastung der jeweiligen Kerne deutet darauf hin, dass im Gegensatz zu STREAM-N-J die Berechnung nicht zwischen verschiedenen Prozessorkernen verschoben wird.

6.1.4 Goroutinen

Die Prozessorauslastung von GO-R-G zeigt die typischen Charakteristika aller sequentiellen Implementierungen der Simulation, wie in Abbildung 8 zu sehen ist.

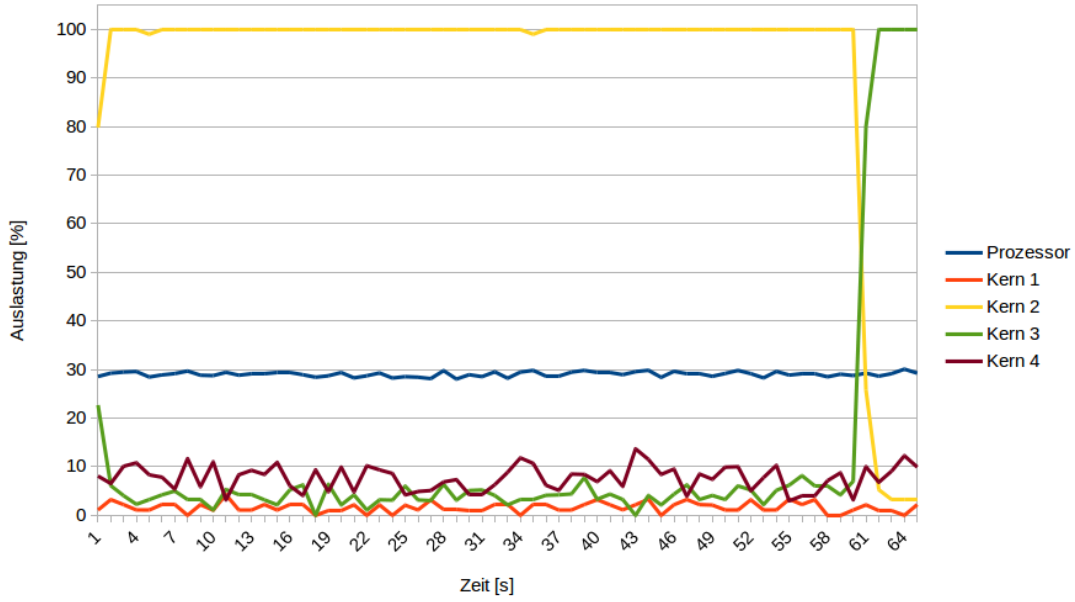

Abbildung 8: Prozessorauslastung GO-R-G

Ein Prozessorkern wird mit 100 Prozent stark ausgelastet, die durchschnittliche Gesamtauslastung des Prozessors liegt bei ca. 30 Prozent. Während der gesamten Ausführung der Simulation wird der ausführende Thread nur einmal zwischen verschiedenen Prozessorkernen verschoben, was an dem starken Absacken der Auslastung von Kern 2, direkt gefolgt von dem starken Anstieg der Auslastung von Kern 3 ersichtlich ist.

Die Prozessorauslastung von GO-N-G ähnelt der von GO-R-G, wie in Abbildung 9 zu sehen ist.

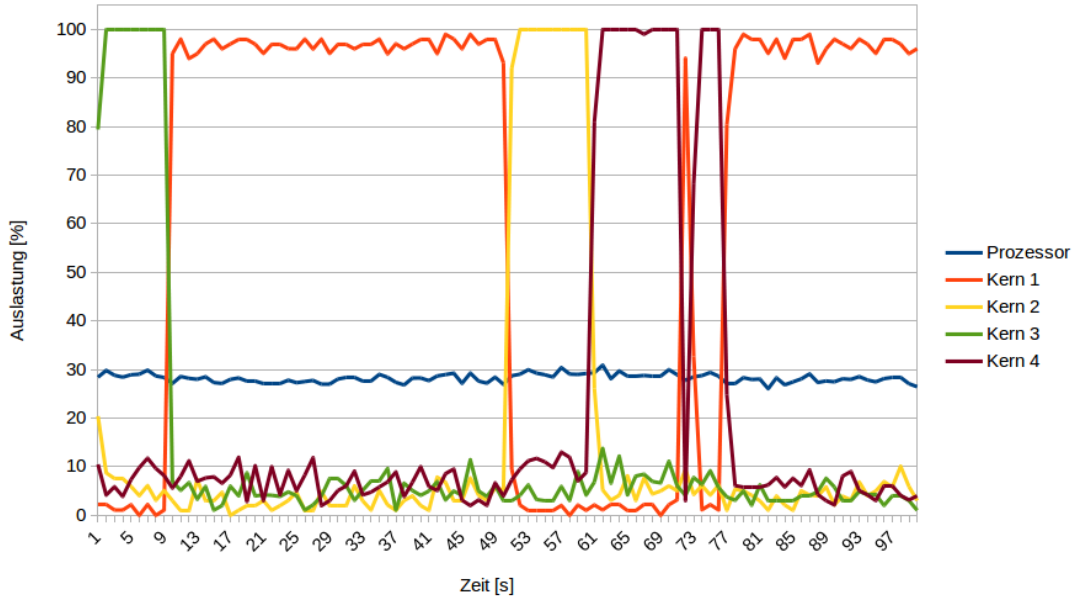

Abbildung 9: Prozessorauslastung GO-N-G

Im Unterschied zur sequentiellen Implementierung wird der berechnende Thread von GO-N-G mehrmals zwischen verschiedenen Prozessorkernen verschoben. Die Gesamtauslastung des Prozessors liegt, ebenfalls wie bei der sequentiellen Implementierung, bei ca. 30 Prozent.

Abbildung 10 zeigt die durchschnittliche Prozessorauslastung von GO-P-G.

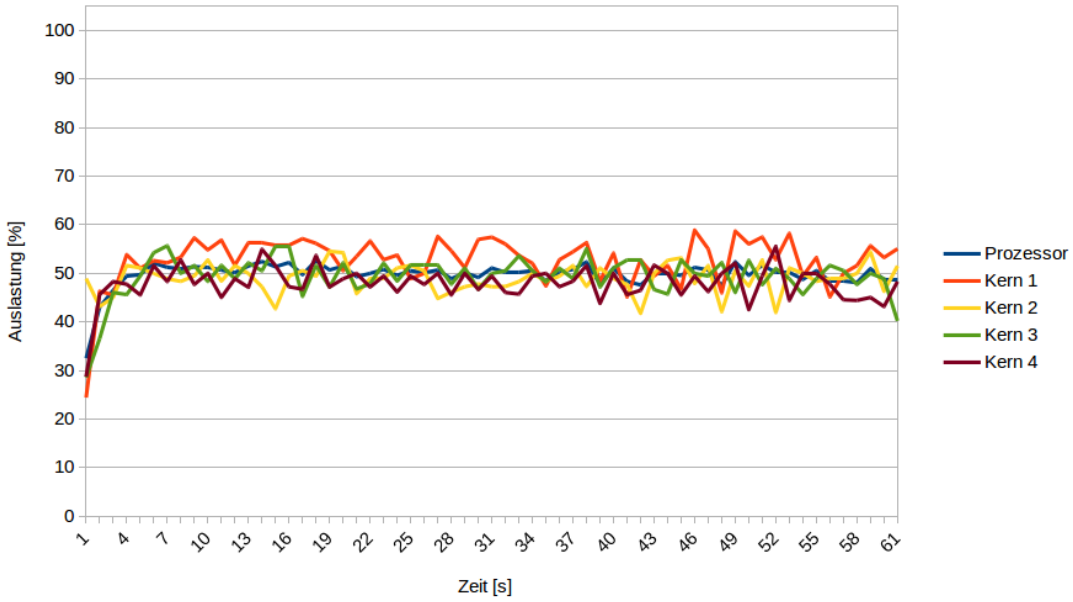

Abbildung 10: Prozessorauslastung GO-P-G

Die Gesamtauslastung des Prozessors ist, im Vergleich zu der von GO-R-G mit ca. 20 Prozent nur geringfügig höher. Da die jeweiligen Kerne nur mit unter 60 Prozent ausgelastet werden, ist die Gesamtauslastung des Prozessors im Vergleich zu derer von STREAM-P-J geringer.

6.1.5 Vergleich der Parallelisierungstechniken

Alle sequentiellen Implementierungen der Simulation weisen eine durchschnittliche Prozessorauslastung von ca. 30 Prozent auf. Trotz der Verwendung von unterschiedlichen Programmiertechniken und Programmiersprachen ändert sich dieser Wert nicht. Aus Abbildung 11 und 12 geht hervor, dass bei den parallelisierten Implementierungen die verwendeten Techniken stärker Einfluss auf die Prozessorauslastung nehmen.
Grafik 11 zeigt die Auslastung des Prozessors während der Berechnung der Simulation mit ca. 490 Kreisen.

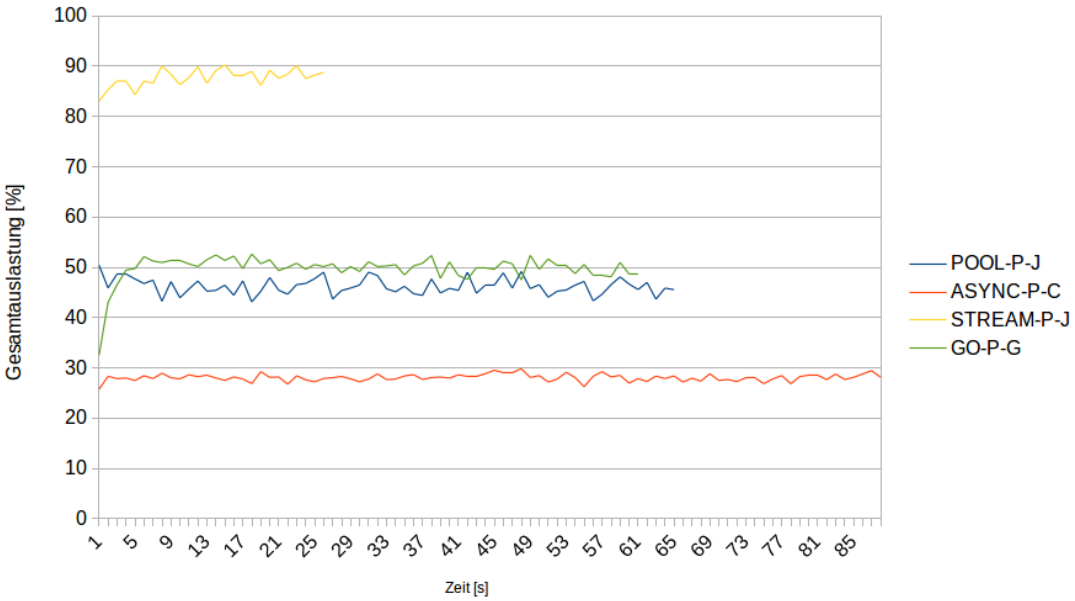

Abbildung 11: Prozessorauslastung: Vergleich aller parallelisierten Implementierungen (~490 Kreise)

Dabei fällt ein Zusammenhang zwischen Auslastung des Prozessors und Ausführungsgeschwindigkeit auf. STREAM-P-J weist mit über 80 Prozent die höchste Auslastung und gleichzeitig geringste Ausführungsdauer auf, ASYNC-P-C mit unter 30 Prozent die geringste Auslastung und größte Ausführungsdauer.

Abbildung 12 zeigt die Auslastung des Prozessors bei Berechnung der Simulation mit ca. 4130 Kreisen.

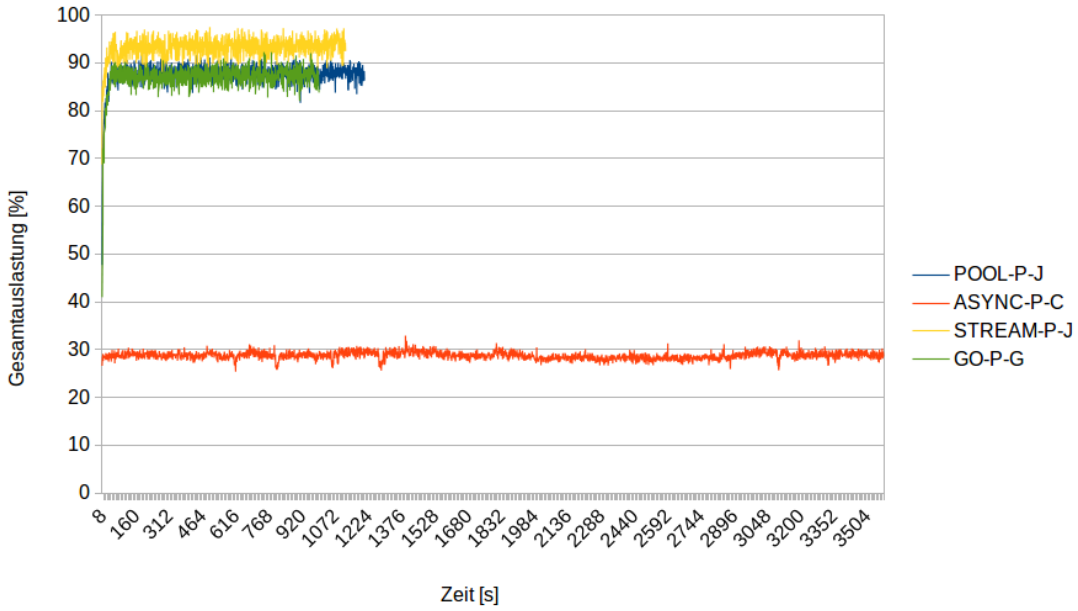

Abbildung 12: Prozessorauslastung: Vergleich aller parallelisierten Implementierungen (~4130 Kreise)

Der Zusammenhang zwischen Prozessorauslastung und Ausführungsgeschwindigkeit ist bei dieser hohen Kreisanzahl nicht mehr erkennbar. ASYNC-P-C benötigt mit unter 30 Prozent Prozessorauslastung zwar am längsten für die Berechnung der Simulation. STREAM-P-J, die, wie bei geringen Kreiszahlen, mit über 90 Prozent die höchste Auslastung erzeugt, benötigt jedoch für die Berechnung länger als GO-P-G.

Im Vergleich zur Ausführungsdauer der Simulation mit ca. 490 Kreisen fällt die Ausführungsdauer von STREAM-P-J hier höher aus. Die Ursache hierfür könnte in der Ausführung der Simulation liegen. Die in STREAM-P-J verwendete Programmiersprache erzeugt Bytecode, der von der Java Virtual Maschine zur Laufzeit interpretiert und in Maschinencode übersetzt wird (Flanagan 2005, S. 8) Im Gegensatz dazu wird der Quellcode von GO-P-G beim Compilieren direkt in Maschinenbefehle übersetzt (Google Inc. 2015a). Dieser fehlende Zwischenschritt könnte zur kürzeren Berechnungsdauer von GO-P-G führen.

6.2 Ausführungszeiten

Im Folgenden werden die Ausführungszeiten der jeweiligen Implementierungen vorgestellt und diskutiert. Die sequentiellen Implementierungen sollten, aufgrund ihrer geringeren CPU-Auslastung, höhere Ausführungszeiten aufweisen, als die parallelen Implementierungen.

6.2.1 Thread Pool

Aus Abbildung 13 werden die Ausführungszeiten von POOL-R-J und POOL-P-J ersichtlich.

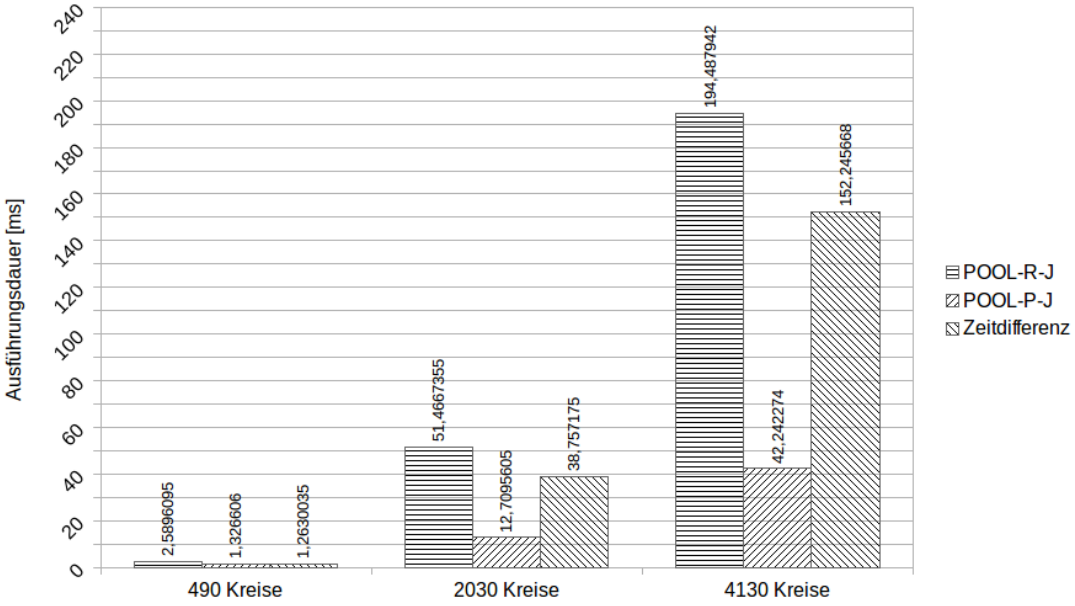

Abbildung 13: Ausführungsdauer von POOL-R-J und POOL-P-J

Diese liegen bei der parallelen Implementierung POOL-P-J stets unter den Ausführungszeiten der sequentiellen Implementierung POOL-R-J, unabhängig von der Anzahl der zu berechnenden Kreise.

6.2.2 C++ 11 – async

Bei den Ausführungszeiten von ASYNC-R-C und ASYNC-P-C, die in Abbildung 14 zu sehen sind, fällt auf, dass die Ausführungsdauer der parallelen Implementierung ASYNC-P-C bei geringen Kreiszahlen höher ist als die von ASYNC-R-C.

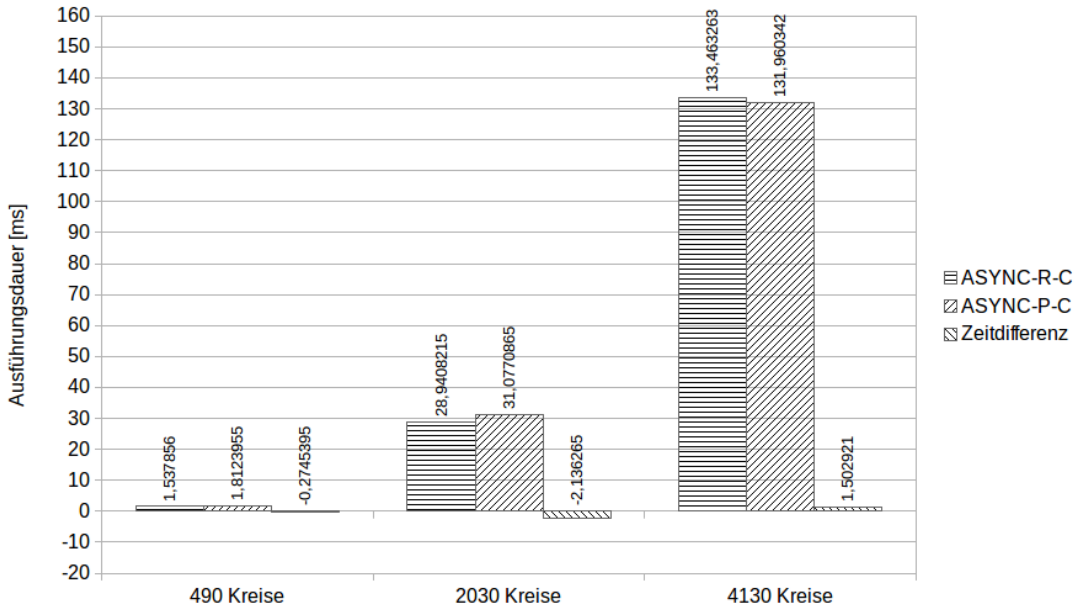

Abbildung 14: Ausführungsdauer von ASYNC-R-C und ASYNC-P-C

Die Parallelisierungstechnik kann die Ausführungsgeschwindigkeit nicht erhöhen, durch den Verwaltungsaufwand der Threads wird diese reduziert. Einzig bei der Simulation mit ca. 4130 Kreisen führt ASYNC-P-C die Berechnung der Simulation um ca. 1,126ms schneller aus als ASYNC-R-C.

6.2.3 Java Parallel Streams

Abbildung 15 zeigt die Ausführungszeiten der Implementierungen STREAM-R-J, STREAM-N-J und STREAM-P-J.

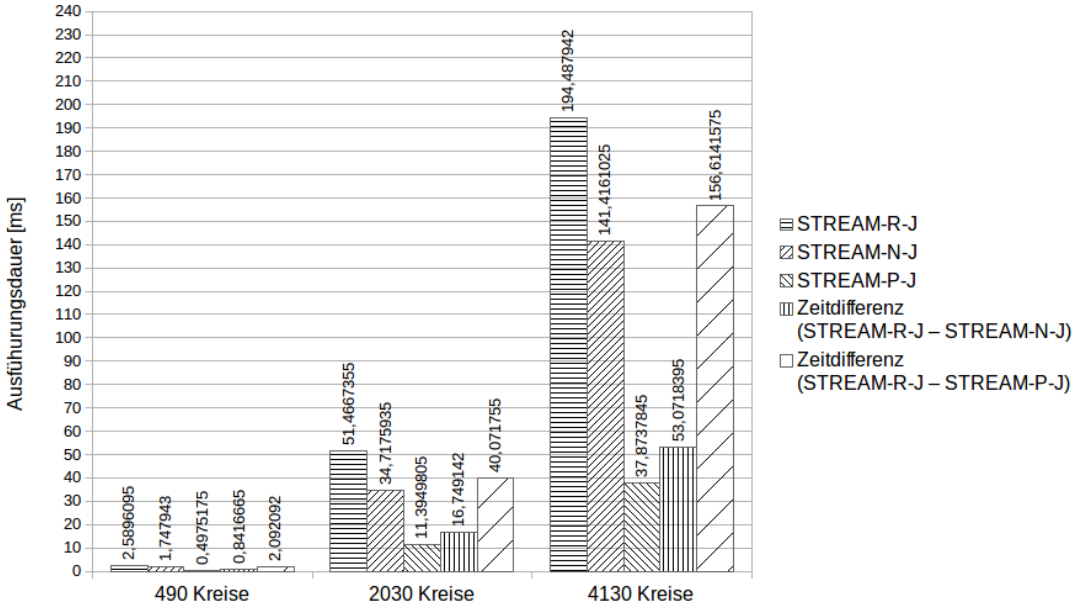

Abbildung 15: Ausführungsdauer von STREAM-R-J, STREAM-N-J und STREAM-P-J

Aus der Grafik geht hervor, dass die Ausführungszeiten von STREAM-P-J sowohl bei geringen, als auch hohen Kreiszahlen unter den Ausführungszeiten von STREAM-R-J liegen.

Die Ausführungszeiten der nebenläufige Implementierung STREAM-N-J liegen ebenfalls unter denen von STREAM-R-J, die Differenz zwischen den Implementierungen fällt mit ca. 53,071ms jedoch deutlich geringer aus, als die zwischen STREAM-R-J und STREAM-P-J.

6.2.4 Goroutinen

Aus Abbildung 16 gehen die Ausführungszeiten der Implementierungen GO-R-G, GO-N-G und GO-P-G hervor.

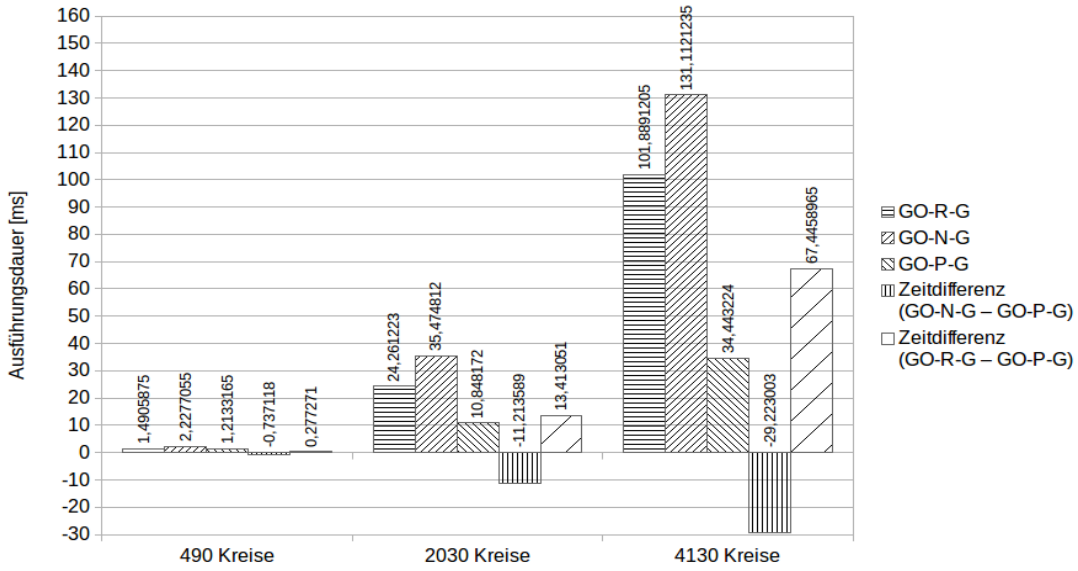

Abbildung 16: Ausführungsdauer von GO-R-G, GO-N-G und GO-P-G

Dabei fällt auf, dass die Ausführungszeiten von GO-P-G stets geringer sind als die von GO-R-G. Die Parallelisierung der Simulation führt also zu einer Beschleunigung deren Berechnung.

Im Gegensatz dazu zeigt sich, dass die nebenläufige Implementierung GO-N-G keinen Geschwindigkeitsvorteil gegenüber GO-R-G bringt. Unabhängig von der Kreisanzahl liegt die Ausführungszeit von GO-N-G stets über der von GO-R-G.

6.2.5 Darstellung der Ausführungszeiten aller parallelen Implementierungen

Um die Vollständigkeit zu waren, werden die Ausführungszeiten aller parallelisierten Simulation in Grafik 17 dargestellt. Ein Vergleich erfolgt jedoch interpretationsoffen und wird, aus Gründen, die in 5.1.1.1 erläutert wurden, nicht in die Auswertung dieser Arbeit miteinbezogen.

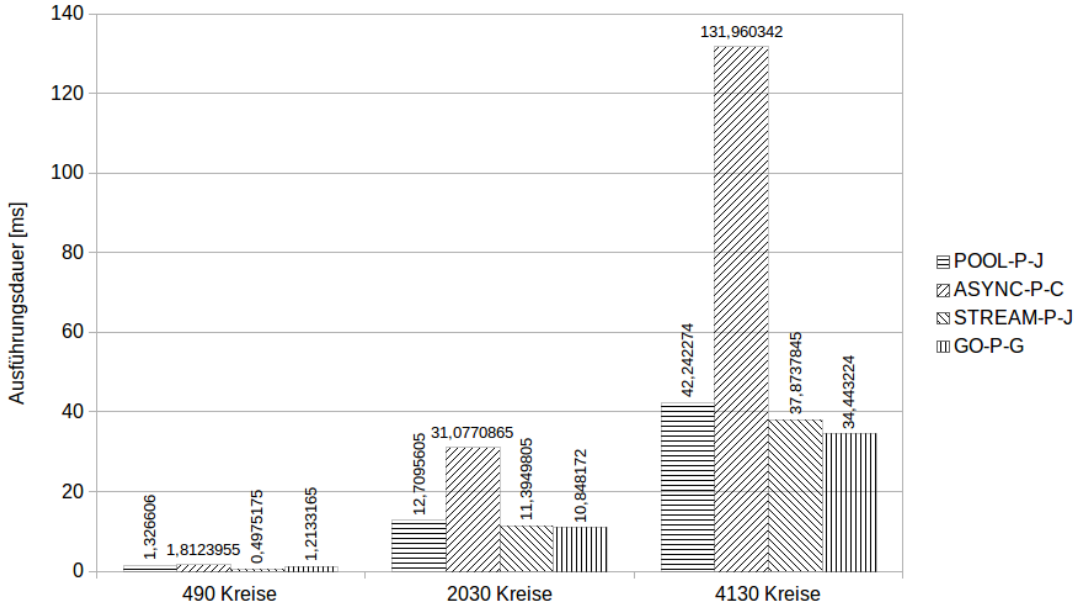

Abbildung 17: Darstellung der Ausführungsdauer aller parallelen Implementierungen

Die Grafik bestätigt die Beobachtung in 6.1.5, bei der deutlich wird, dass ASYNC-P-C die höchste Ausführungsdauer aufweist.

6.3 Beschleunigungsfaktor

Aus den Ausführungszeiten der Simulationen werden die Beschleunigungsfaktoren der jeweiligen Parallelisierungstechnik errechnet. Diese ergeben sich aus der Beschleunigung der entsprechenden parallelen Implementierung im Vergleich zu ihren Referenzimplementierung, wie in Grafik 18 dargestellt. Der Beschleunigungsfaktor sollte möglichst hoch sein.

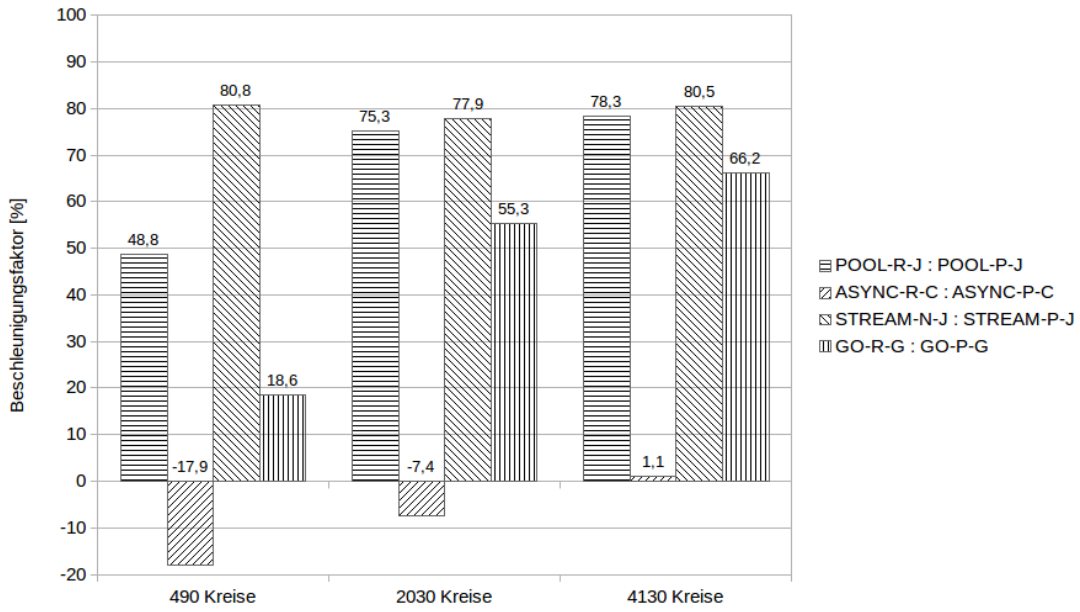

Abbildung 18: Beschleunigungsfaktor aller Implementierungen

Die Grafik zeigt, dass die für POOL-P-J verwendete Parallelisierungstechnik bei geringer Anzahl zu berechnender Kreise weniger hohe Beschleunigungswerte erzeugt, als bei hoher Kreisanzahl. Bei hoher Anzahl schwankt der Wert nur noch gering um drei Prozent. POOL-P-J erzeugt mit 78,3 Prozent den, im Vergleich zu den anderen Implementierungen, zweit höchsten Beschleunigungsfaktor bei der Berechnung der Simulation.

Im Vergleich mit den anderen Parallelisierungstechniken zeigt die für ASYNC-P-C verwendete, den geringsten Beschleunigungsfaktor. Während bei einer Kreisanzahl von ~490 Kreisen, bzw. ~ 2030 Kreisen die Beschleunigung negativ ausfällt, die Simulation im Vergleich zur sequentiellen Variante also langsamer berechnet wird, weißt die Simulation bei einer Kreisanzahl von ~ 4130 eine leichte Beschleunigung von 1,1 Prozent auf. Der Beschleunigungsfaktor von Goroutinen (GO-P-G) wird am stärksten durch die Anzahl der zu berechnenden Kreise beeinflusst, steigt jedoch ähnlich zu dieser an.

Aus der Grafik wird ersichtlich, dass die Anzahl der zu berechnenden Kreise kaum Einfluss auf den Beschleunigungsfaktor der Parallelisierungstechnik von STREAM-P-J hat. Dieser schwankt bei allen Simulationen jeweils um maximal einen Prozent. Der Beschleunigungsfaktor von STREAM-P-J ist mit 80,8 Prozent zudem der höchste im Vergleich zu anderen Implementierungen.

Der Beschleunigungsfaktor von Goroutinen (GO-P-G) wird am stärksten durch die Anzahl der zu berechnenden Kreise beeinflusst, steigt jedoch proportional zu dieser an. Mit 66,2 Prozent fällt der Faktor im Vergleich mit den anderen Implementierungen jedoch deutlich geringer aus.

6.4 Komplexität

Nachfolgend werden Daten präsentiert, die Aufschluss über die Komplexität der jeweiligen Implementierungen geben. Die Komplexität der parallelen Implementierung sollte, im Vergleich zur sequentiellen Referenzimplementierung möglichst nicht steigen.

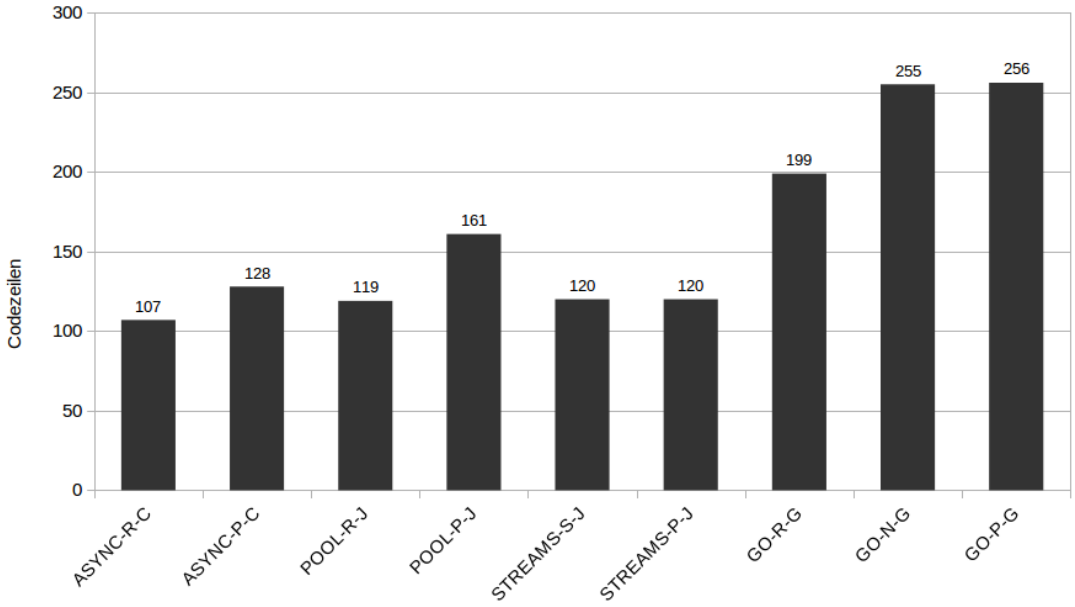

Abbildung 19: Anzahl der Codezeilen der verschiedenen Implementierungen

Aus Abbildung 19 wird ersichtlich, dass GO-P-G mit über 250 Zeilen Code die umfangreichste Implementierung ist. Dies resultiert aus dem, im Vergleich zu anderen Varianten geänderten Algorithmus zur Parallelisierung. Dieser nutzt Kommunikationskanäle, um die Synchronisierung des Programmes zu Optimieren. Des Weiteren ist die Umsetzung von Objektorientierung in Go im Vergleich zu anderen Sprachen relativ aufwendig, was zu deutlich längerem Code führt (Google Inc. 2015a).

Die folgende Abbildung 20 hebt den Unterschied der Implementierungen weiter hervor. Sie stellt die Differenz der Anzahl der Codezeilen zwischen den verschiedenen Implementierungen dar.

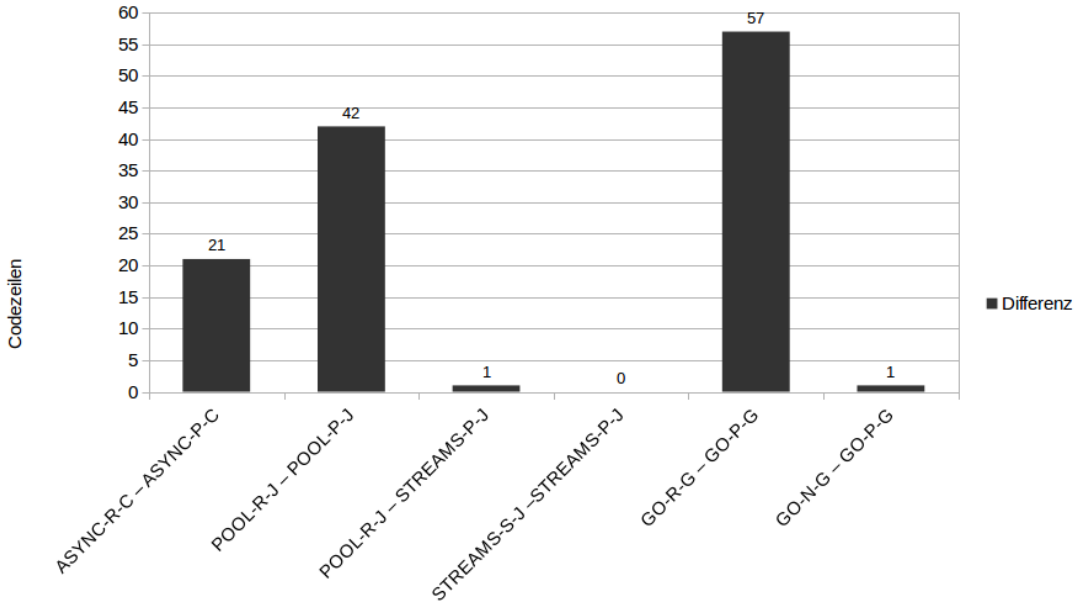

Abbildung 20: Differenz der Anzahl an Codezeilen zwischen den verschiedenen Implementierungen

Die Unterschiede der Differenz zwischen ASYNC-P-C und POOL-P-J sind, trotz der Verwendung des gleichen Parallelisierungsalgorithmus, relativ hoch. Grund hierfür ist die zwingend erforderliche Exception-Behandlung von Java in der parallelisierten Implementierung (14 Zeilen). Werden die dafür nötigen Codezeilen von der Differenz der Anzahl an Codezeilen zwischen POOL-R-J und POOL-P-J abgezogen, zeigt, sich, dass beide Varianten einen sehr ähnlichen Parallelisierungsaufwand aufweisen.

Trotz unterschiedlicher verwendeter Programmiertechniken entstehen zwischen POOL-R-J und STREAM-P-J geringen Differenzen. In letzterer Variante werden Lambda-Ausdrücke innerhalb der Stream-Operatoren eingesetzt, während erstere mit traditionellen Funktionen arbeitet. Ein Grund für die ähnliche Komplexität ist die fehlende Synchronisierung der Parallelisierung von STREAM-P-J. Da Streams implizit synchronisiert werden, ist hierfür kein programmatischer Aufwand nötig, was die Komplexität der Implementierung deutlich minimiert.

Die Komplexität der Implementierung eines sequentiellen Streams unterscheidet sich kaum von deren eines parallelen Streams, was die geringe Differenz zwischen STREAM-N-J und STREAM-P-J zeigt. Werden Anwendungen mit Hilfe von Streams entwickelt, können diese ohne nennenswerten Mehraufwand parallelisiert werden. Dies ermöglicht eine sehr einfache Anpassung an das Zielsystem.

Ebenso wie die Parallelisierung von Streams erfordert die Parallelisierung von Goroutinen kaum Mehraufwand, wenn diese bereits auf Nebenläufigkeit ausgelegt sind.

Dieser Aufwand ist, mit über 55 Zeilen Code im Vergleich zur sequentiellen Version, jedoch relativ hoch. Im Vergleich mit dem Mehraufwand anderer parallelisierten Implementierungen ist diese Variante die aufwendigste.

6.5 Abschließende Bewertung

Die Parallelisierung mit Hilfe von async bietet, unter Linux, Nachteile im Vergleich zur sequentiellen Implementierung. Der Mehraufwand für die Umsetzung der Parallelisierung wird weder durch eine bessere Prozessorauslastung, noch durch eine Beschleunigung der Simulationsberechnung gerechtfertigt. Um die Vorteile der Parallelisierungstechnik zu nutzen, müsste weiterer Mehraufwand in die Implementierung der Parallelisierung fließen, um das Startverhalten der asynchronen Aufrufe zu ändern. Die an die parallelisierte Implementierung gestellten Ziele einer gleichmäßig hohen Prozessorauslastung auf allen Kernen, geringer Ausführungsdauer im Vergleich zur sequentiellen Implementierung, sowie geringerer Komplexität wurden nicht erreicht.

Die Parallelisierungstechnik Java Parallel Streams zeig die besten Eigenschaften der in dieser Arbeit verglichenen Techniken. Der Synchronisierungswand ist dank der impliziten Synchronisierung sehr gering, was die Bedienbarkeit der Technik stark vereinfacht. Der Beschleunigungsfaktor ist, im Vergleich mit dem der anderen Parallelisierungstechniken am höchsten. Die Ausführungsgeschwindigkeit ist, bei geringen Kreiszahlen die höchste im Vergleich, Auch lastet die Technik den Prozessor mit über 80 Prozent am besten aus. Damit erfüllt die Technik die Ziele der gleichmäßig hohen Prozessorauslastung und des hohen Beschleunigungsfaktors am besten.

Der Mehraufwand für den komplexeren Parallelisierungsalgorithmus, den GO-P-G nutzt, ist, bei Berechnung von mittleren und hohen Kreiszahlen, gerechtfertigt. Die Ausführungsdauer ist, im Vergleich zu den anderen Implementierungen, am geringsten, obwohl die Prozessorauslastung geringer ausfällt, als die von STREAM-P-J. Die Komplexität der Implementierung wird jedoch mit Go-Version 1.5 sinken, in der die manuelle Deklaration der zu nutzenden Prozessorkerne entfällt.

Die Parallelisierung mit Hilfe von Thread Pool erzeugt eine vergleichsweise hohe Komplexität. Dies wird weder durch eine hohe und gleichmäßige Prozessorauslastung, noch durch sehr schnelle Ausführung im Vergleich zur sequentiellen Implementierung gerechtfertigt.

7 Zusammenfassung und Beantwortung der Forschungsfragen

Im folgenden Kapitel werden die Eingangs gestellten Forschungsfragen beantwortet. Danach wird die Thesis abschließend zusammengefasst und ein Ausblick über anschließende Forschungsthemen gegeben.

7.1 Beantwortung der Forschungsfragen

Die Forschungsfragen, die in dieser Thesis bearbeitet werden, lassen sich nun wie folgt beantworten:

a) Welche Merkmale der Implementierungen können genutzt werden, um einen Vergleich zwischen parallelen und sequentiellen Referenzimplementierungen sowie zwischen unterschiedlichen parallelen Implementierungen hinsichtlich ihrer Stärken und Schwächen zu ermöglichen?
   - Um die Implementierungen auf dem Bereich der Leistung vergleichen zu können, bieten sich folgende Merkmale an:
     - Um zwischen sequentieller Referenzimplementierung und paralleler Implementierung zu vergleichen, können die Ausführungszeiten der Simulation verglichen werden.
     - Um verschiedene Parallelisierungstechniken zu vergleichen, kann der Beschleunigungsfaktor zwischen Referenzimplementierung und paralleler Implementierung herangezogen werden. Dieser erlaubt es, die Techniken unabhängig von der genutzten Programmiersprache zu vergleichen.
     - Die Prozessorauslastung liefert einen aussagekräftigen Wert zur Ressourcennutzung der jeweiligen Parallelisierungstechnik.
   - Um die Parallelisierungstechnik hinsichtlich ihrer Bedienbarkeit zu vergleichen, bietet ein Vergleich der Anzahl der Quelltextzeilen der Implementierungen ein aussagekräftiges und quantifizierbares Instrument an. Dieses gibt an, wie viele Anweisungen im Vergleich zur Referenzimplementierung zusätzlich nötig sind, um die Implementierung zu parallelisieren.

b) Wie unterscheiden sich die untersuchten Implementierungen gemessen an den definierten Merkmalen?

- Hinsichtlich der Ressourcennutzung weißt die Parallelisierungstechnik Java Parallel Streams die höchste Prozessorauslastung an. C++11-async lastet den Prozessor am geringsten aus. Die Auslastung von Thread Pool und Goroutinen unterscheidet sich nur geringfügig.
- Java Parallel Streams bieten den, vergleichsweise, höchsten Beschleunigungsfaktor, der sich jedoch nur geringfügig von dem von Thread Pools unterscheidet. Der Beschleunigungsfaktor von Goroutinen ist stark von der Anzahl der zu berechnenden Objekte abhängig. C++11 – async erzeugt negative Beschleunigungswerte, die Parallelisierungstechnik verlangsamt die Berechnung also.
- Die Parallelisierung mittels Goroutinen erzeugt die meisten zusätzlichen Zeilen Quelltext und damit den, vergleichsweise, höchsten Mehraufwand. Die parallele Implementierung mittels Thread Pools ist ebenfalls wesentlich aufwendiger als deren Referenzimplementierung. C++11 – async erzeugt einen vergleichsweise geringen Mehraufwand. Java Parallel Streams erzeugen, im Vergleich zur Referenzimplementierung, nahezu keinen Mehraufwand. Des Weiteren zeigt sich, dass die Parallelisierung mithilfe von Goroutinen oder Java Parallel Streams ausgehend von einer nebenläufigen Implementierung kaum Mehraufwand bedeutet – die Menge des Quelltextes unterscheidet sich nur um 2 Zeilen.

7.2 Zusammenfassung

In dieser Arbeit wurden verschiedene Parallelisierungstechniken anhand einer Beispielanwendung evaluiert. Dazu wurde mit einer Physiksimulation eine Anwendung gewählt, die, aufgrund ihrer häufigen Anwendung und übertragbaren Problemstellungen, eine hohe Alltagsrelevanz hat.

Für die Implementierungen der Beispielanwendung wurden die Parallelisierungstechniken Thread Pools und Parallel Streams in Java, C++ async-Funktionen sowie die in Go verfügbaren Goroutinen verwendet. Zusätzlich wurde für eine sequentielle Referenzimplementierung in der jeweiligen Programmiersprache entwickelt.

Für den Vergleich der Parallelisierungstechniken wurden verschieden Merkmale der Implementierungen untersucht und daraus eine Metrik gebildet. Diese umfasst die Komplexität der parallelen Implementierung, den Geschwindigkeitsgewinn bei Aus-

führung der parallelen Implementierung im Vergleich zur sequentiellen Referenzimplementierung, sowie deren Prozessorauslastungen.

Anhand dieser Metriken wurden die verschiedenen Implementierungen verglichen. Dabei stellte sich heraus, dass die Parallelisierungstechnik Java Parallel Streams, im Vergleich mit den anderen Techniken die besten Eigenschaften aufweist. Die damit implementierte Parallelisierung erforderte wenig Mehraufwand, insbesondere durch die implizite Synchronisierung, lastete den Prozessor am besten aus und konnte bei geringen Kreiszahlen die Simulation am schnellsten berechnen. Bei mittleren und hohen Kreiszahlen lieferten Goroutinen die höchste Berechnungsgeschwindigkeit, trotz einer geringeren Prozessorauslastung im Vergleich zur Stream-Implementierung. Jedoch ist der Mehraufwand für die Parallelisierung, verglichen mit anderen Techniken, am höchsten.

Die in dieser Thesis gewonnen Ergebnisse zu Ausführungszeiten der jeweiligen Implementierungen lassen sich kaum auf andere Beispiele übertragen. Die Aussagen zur Beschleunigung der Parallelisierungstechniken und zu deren Komplexität haben jedoch allgemeinere Bedeutung und können auf andere Anwendungsfälle übertragen werden.

7.3 Ausblick

Während der Bearbeitung dieser Thesis stellten sich weiterführende interessante Möglichkeiten zum Vergleich von Parallelisierungstechniken heraus.

Ein noch weitgehend offenes Feld ist die Bedienbarkeit der verschiedenen Parallelisierungstechniken. Hier wurde die Anzahl der Quelltextzeilen als Maß für die Komplexität der Parallelisierungstechnik gewählt. Interessant wäre, das tatsächliche Nutzerverhalten bei Bedienung der Techniken zu beobachten und auszuwerten. Dazu bietet sich eine qualitative Studie an, bei der erfasst wird, wie verständlich, schnell erlernbar und fehleranfällig die verschiedenen Techniken sind. Dies könnte ebenfalls mit Hilfe einer Beispielanwendung durchgeführt werden, die den Teilnehmerinnen entweder vorgelegt wird, und diese sie interpretieren, oder die die Teilnehmerinnen selbst implementieren. Um die Teilnehmerinnen nicht durch eine eventuell komplexe Anwendungslogik abzulenken, sollte eine allgemein bekannte Anwendung, wie z.B. eine Dateisuche im Verzeichnisbaum verwendet werden.

Parallelisierte Programme werden entweder in Maschinenbefehle kompiliert, oder durch eine Laufzeitumgebung interpretiert. Dabei werden sie entweder durch den

Compiler, oder durch *Just-in-time* Kompilierung auf höhere Ausführungsgeschwindigkeit optimiert. Hier könnte durch einen Vergleich zwischen optimierten und unoptimierten Programmen untersucht werden, wie stark der Einfluss der Optimierungen auf die Ausführungsgeschwindigkeit der Programme ist.

Des Weiteren sollte untersucht werden, wie die verschiedenen Parallelisierungstechniken mit unterschiedlicher Anzahl an nutzbaren Prozessorkernen skalieren. Dazu könnte ein Vergleich durchgeführt werden, bei dem die verschiedenen Implementierungen der Simulation auf unterschiedlichen Anzahlen Prozessorkernen aufgeführt werden.

Parallelisierungstechniken werden in dieser Arbeit für die Beschleunigung von Berechnungsschritten eingesetzt. Ein weiterer Anwendungsfall liegt in der Parallelisierung von Benutzeroberflächen. Dabei wird versucht, durch asynchrone Ausführung von sehr zeitaufwendigen Programmteilen eine durchgehende Bedienbarkeit der Oberfläche zu garantieren. Hier könnte ebenfalls ein Vergleich zwischen verschiedenen Parallelisierungstechniken durchgeführt werden, der dann mit dieser Arbeit korreliert wird, um zu ermitteln, ob eine Parallelisierungstechnik sowohl Berechnungsschritte performant parallelisieren, als auch eine Bedienoberfläche mit anderen Programmkomponenten asynchron verknüpfen kann.

Die Weiterentwicklung von Parallelisierungstechniken wird sicher fortgeführt werden, mit dem Ziel, das Leistungspotential moderner Prozessoren besser auszuschöpfen.
Zum einen wird der Aufwand für die Parallelisierung einer Anwendung durch automatische Parallelisierung, z.B. durch den Compiler, gesenkt werden.
Zum anderen wird die Nutzung moderner Prozessorarchitekturen verbessert, in dem z.B. Berechnungen automatisch auf integrierte Grafikprozessoren verlagert werden, was zu stärkerer Beschleunigung von parallelisierten Anwendungen führen wird.

8 Quellen

Belikov, Evgenij, Panazis Deligiannis, Prabhat Totoo, Malak Aljabri, und Hans-Wolfgang Loidl. 2013. „A Survey of High-Level Parallel Programming Models.“ http://www.macs.hw.ac.uk/~eb96/pdf/high-level-parallelism-survey-2013.pdf.

Ben-Ari, Mordechai. 2006. *Principles of Concurrent and Distributed Programming*. 2. Auflage. Harlow: Pearson Education Limited.

Canonical Ltd (Hg.). o.A. *Ubuntu 14.04.2 LTS (Trusty Tahr)*. Zugegriffen am 3.8.2015. http://releases.ubuntu.com/14.04/.

Church, Alonzo. 1936. „A Note on the Entscheidungsproblem.“ In *The Journal of Symbolic Logic* 1 (1): 40–41.

Conrod, Jay. 16.7.2015. *Stackoverflow - Ball to Ball Collision - Detection and Handling.* Zugegriffen am 3.8.2015. http://stackoverflow.com/questions/345838/ball-to-ball-collision-detection-and-handling.

cppreference.com (Hg.). o.A. s*td::chrono::high_resolution_clock*. Zugegriffen am 3.8.2015. http://www.cplusplus.com/reference/chrono/high_resolution_clock/.

cppreference.com (Hg.). 14.8.2013. s*td::launch*. August 14. Zugegriffen am 3.8.2015. http://en.cppreference.com/w/cpp/thread/launch.

cppreference.com (Hg.). 2.4.2015a. s*td::async - Cppreference.com*. Zugegriffen am 24.7.2015. http://en.cppreference.com/w/cpp/thread/async.

cppreference.com (Hg.). 23.52015b. *C++ Concepts: Callable – Cppreference.com*. Zugegriffen am 29.6.2015. http://en.cppreference.com/w/cpp/concept/Callable.

Feess, Eberhard. o.A. *Komplexität*. Zugegriffen am 24.8.2015. http://wirtschaftslexikon.gabler.de/Archiv/5074/komplexitaet-v8.html.

Feo, J.T., Hrsg. 1992. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. Bd. 6. Special Topics in Supercomputing. Amsterdam: Elsevier Science Publishers B.V.

Flanagan, David. 2005. *Java in a Nutshell - Deutsche Ausgabe Für Java 1.4*. 4. Auflage. Köln: O’Reilly Verlag.

Gerrand, Andrew. 16.1.2013a. *The Go Blog - Concurrency Is Not Parallelism.* Zugegriffen am 3.8.2015. https://blog.golang.org/concurrency-is-not-parallelism.

Gerrand, Andrew. 10.11.2013b. *Four Years of Go - The Go Blog.* Zugegriffen am 10.8.2015. https://blog.golang.org/4years.

Glatz, Eduard. 2015. *Betriebssysteme - Grundlagen, Konzepte, Systemprogrammierung*. 3. überarbeitete und aktualisierte. Heidelberg: dpunkt.verlag.

Gleim, Urs, und Tobias Schüle. 2011. *Multicore-Software, Grundlagen Architektur und Implementierung in C/C++, Java und C#.* Heidelberg: dpunkt.verlag.

Godard, Sebastien. 6.2015. *Mpstat Manual Page*. Zugegriffen am 3.8.2015. http://sebastien.godard.pagesperso-orange.fr/man_mpstat.html.

Google Inc. (Hg.). o.A. a *A Tour of Go - Exercise: Web Crawler*. Zugegriffen am 26.7.2015. https://tour.golang.org/concurrency/9.

Google Inc. (Hg.). o.A. b. *Documentation - The Go Programming Language*. Zugegriffen am 20.8.2015. https://golang.org/doc/.

Google Inc. (Hg.). o.A. c. *Effective Go – Channels*. Zugegriffen am 3.8.2015. https://golang.org/doc/effective_go.html#channels.

Google Inc. (Hg.). o.A. d. *Effective Go – Goroutines*. Zugegriffen am 3.8.2015. https://golang.org/doc/effective_go.html#goroutines.

Google Inc. (Hg.). o.A. e. *Effective Go – Parallelization*. Zugegriffen am 3.8.2015 https://golang.org/doc/effective_go.html#parallel.

Google Inc. (Hg.). o.A. f. *Time - The Go Programming Language – UnixNano*. Zugegriffen am 3.8.2015 http://golang.org/pkg/time/#Time.UnixNano.

Google Inc. (Hg.). 5.8.2015a. *The Go Programming Language Specification - The Go Programming Language - Method Declarations*. Zugegriffen am 10.8.2015. https://golang.org/ref/spec#Method_declarations.

Google Inc. (Hg.). 2015b. *Go 1.5 Release Notes - The Go Programming Language - Runtime*. Zugegriffen am 10.8.2015. https://tip.golang.org/doc/go1.5#runtime.

Grama, Ananth, Anshul Gupta, George Karypis, und Vipin Kumar. 2003. *Introduction to Parallel Computing*. zweite Auflage. Harlow: Pearson Education Limited.

Gustfason, John L. 5.1988. *Speedup under Amdahl's Law*. Zugegriffen am 13.8.2015. http://www.johngustafson.net/pubs/pub13/Fig1.gif.

Hahn, Jens-Uwe. 2000. „Simulation und Photorealismus." Tübingen: Eberhard-Kars-Universität Tübingen. https://publikationen.uni-tuebingen.de/xmlui/bitstream/handle/10900/48309/pdf/diss_hahn_neu.pdf?sequence=1&isAllowed=y.

Halliday, David, und Robert Resnick. 1994. *Physik, Part 2*. New York: Walter de Gruyter.

Havok Group (Hg.). o.A. *Havok Physics*. Zugegriffen am 26.7.2015. http://www.havok.com/physics/.

Herlihy, Maurice, und Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Burlington,: Morgan Kaufmann Publishers.

Inden, Michael. 2014. *Java 8 Die Neuerungen*. Heidelberg: dpunkt.verlag.

Intel Corporation (Hg.). o.A. *ARK | Intel Core i5-2400 Processor (6M Cache, up to 3.40 GHz)*. Zugegriffen am 10.8.2015. http://ark.intel.com/products/52207/Intel-Core-i5-2400-Processor-6M-Cache-up-to-3_40-GHz.

Intel Corporation (Hg.). o.A. b *Atomic Operations | Intel® Developer Zone*. Zugegriffen am 30.5.2015. https://software.intel.com/en-us/node/506090.

Intel Corporation (Hg.). o.A. c. *Legacy Intel® Pentium® Processor*. Zugegriffen am 15.7.2015. http://ark.intel.com/products/family/78132/Legacy-Intel-Pentium-Processor#@Desktop.

Intel Corporation (Hg.). 29.6.2015. *Processors Temperature FAQs*. Zugegriffen am 10.8.2015. http://www.intel.com/support/processors/sb/CS-033342.htm.

Kasim, Henry, Rita Zhang, und Simon See. 2008. „Survey on Parallel Programming Model." In *Network and Parallel Computing*, 5245:266–75. Lecture Notes in Computer Science. Springer Berlin Heidelberg. http://web.cse.ohio-state.edu/~agrawal/788-sp13/Papers/3.models-overview.pdf.

Kurzweil, Peter, Bernhard Frenzel, und Florian Gebhard. 2009. *Physik Formelsammlung*. 2. Auflage. Wiesbaden: Vieweg+Teubner.

Lindholm, Tim, Frank Yellin, Gilad Bracha, und Alex Buckley. 13.2.2015. *The Java Virtual Machine Specification - Java SE 8 Edition*. Zugegriffen am 3.8.2015. https://docs.oracle.com/javase/specs/jvms/se8/html/.

Maurer, Christian. 2012. *Nichtsequentielle Programmierung mit Go 1*. 2. Auflage. Springer Vieweg.

McCabe, Thomas J. 1976. „A Complexity Measure.“ *IEEE Transactions on Software Engineering* SE-2 (4): 308–20. doi:10.1109/TSE.1976.233837.

Merzinger, Gerhard, und Thomas Wirt. 2006. *Repetitorium der höheren Mathematik*. 5. Aufl. Barsinghausen: Binomi Verlag.

Microsoft Corporation (Hg.). o.A. a. *Asynchrone Programmierung Mit Async Und Await (C# Und Visual Basic) - C#*. Zugegriffen am 10.8.2015. https://msdn.microsoft.com/de-de/library/Hh191443(v=vs.120).aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-1.

Microsoft Corporation (Hg.). o.A. b. *Asynchrone Programmierung Mit Async Und Await (C# Und Visual Basic) - Visual Basic*. Zugegriffen am 10.8.2015. https://msdn.microsoft.com/de-de/library/Hh191443(v=vs.120).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2.

Microsoft Corporation (Hg.). o.A. c. *Code Metrics Values*. Zugegriffen am 3.8.2015. https://msdn.microsoft.com/en-us/library/bb385914.aspx.

Microsoft Corporation (Hg.). o.A. d. *Lambda Expressions in C++*. Zugegriffen am 29.6.2015. https://msdn.microsoft.com/en-us/library/dd293608.aspx.

Microsoft Corporation (Hg.). o.A. e. *Minimieren von Deadlocks*. Zugegriffen am 4.5.2015. https://technet.microsoft.com/de-de/library/ms191242(v=sql.105).aspx.

Oracle (Hg.). o.A. a. *Callable (Java Platform SE 8)*. Zugegriffen am 30.7.2015. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html.

Oracle (Hg.). o.A. b. *Catching and Handling Exceptions*. Zugegriffen am 3.8.2015. https://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html.

Oracle (Hg.). o.A. c. *Executors (Java Platform SE 7)*. Zugegriffen am 3.7.2015. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool(int).

Oracle (Hg.). o.A. d. *Interface Collection<E>*. Zugegriffen am 3.8.2015. http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html.

Oracle (Hg.). o.A. e. *Java.util.concurrent (Java 2 Platform SE 5.0)*. Zugegriffen am 10.8.2015. https://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/package-summary.html.

Oracle (Hg.). o.A. f. *Java.util.concurrent (Java Platform SE 8)*. Zugegriffen am 10.8.2015. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html.

Oracle (Hg.). o.A. g. *Lambda Expressions*. Zugegriffen am 30.7.2015. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

Oracle (Hg.). o.A. h. *Lock (Java Platform SE 7)*. Zugegriffen am 24.7.2015. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html#tryLock.

Oracle (Hg.). o.A. i. *Package Java.util.stream*. Zugegriffen am 3.8.2015. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

Oracle (Hg.). o.A. j. *ParallelStream*. Zugegriffen am 3.8.2015. https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#parallelStream--.

Oracle (Hg.). o.A. k. *Runnable (Java Platform SE 8)*. Zugegriffen am 30.7.2015. https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html.

Oracle (Hg.). o.A. l. *Starvation and Livelock*. Zugegriffen am 7.6.2015. https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html.

Oracle (Hg.). o.A. m. *System (Java Platform SE 8) – nanoTime*. Zugegriffen am 3.8.2015. http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--.

Oracle (Hg.). o.A. n. *ThreadPoolExecutor (Java Platform SE 7)*. Zugegriffen am 3.7.2015. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html#execute(java.lang.Runnable.

Oracle (Hg.). o.A. o. *ThreadPoolExecutor (Java Platform SE 7)*. Zugegriffen am 3.7.2015. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html#shutdown.

Rauber, Thomas, und Gundula Rünger. 2010. *Parallel Programming for Multicore and Cluster Systems*. Heidelberg: Springer.

Smith, Richard. 2015. „Working Draft, Standard for Programming Language C++ N4527.“ ISO/IEC. http://open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4527.pdf.

Stroustrup, Bjarne. 2015. *Die C++ Programmiersprache*. München: Carl Hanser Verlag.

Sun Microsystems (Hg.). 2001a. *Semaphores (Multithreaded Programming Guide)*. Zugegriffen am 24.7.2015. http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032s/index.html.

Sun Microsystems (Hg.). 2001b. *Using Barrier Synchronization (Multithreaded Programming Guide)*. Zugegriffen am 4.5.2015. https://docs.oracle.com/cd/E19120-01/open.solaris/816-5137/gfwek/index.html.

Sun Microsystems Inc. (Hg.). 2001c. *Defining Multithreading Terms*. Zugegriffen am 30.5.2015. http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html.

Toit, Stefanus Du. 2012. „Working Draft, Standard for Programming Language C++ - N3337.“ ISO/IEC. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf.

Valles, Antonio. 20.11.2009. *Performance Insights to Intel® Hyper-Threading Technology*. Zugegriffen am 15.7.2015.. https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology.

Wasson, Scott. 21.2.2005. *Intel's Pentium 4 600 Series Processors - Power Consumption.* Zugegriffen am 25.8.2015. http://techreport.com/review/7998/intel-pentium-4-600-series-processors/16.

Will, Torsten. 2012. *C++11 programmieren*. Bonn: Galileo Press.

Winterberg, Benjamin. 31.7.2015. *Java 8 Stream Tutorial*. Zugegriffen am 25.8.2015. http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/.

9 Anhang

Quellcode

Der Quellcode aller Implementierungen ist auf der beliegenden CD zu finden.