DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

DLR Deutsches Zentrum für Luft- und Raumfahrt e.V.
in der Helmholtz-Gemeinschaft

Evaluation und Verbesserung eines Task-Schedulers für eine HPC Anwendung

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

des Studiengangs Informationstechnik

der Dualen Hochschule Baden-Württemberg Mannheim

von

Marius Messerschmidt

Abgabe am 11. September 2020

Bearbeitungszeitraum:	22.06.20 – 13.09.20
Matrikelnummer, Kurs:	2774756, TINF17ITIN
Ausbildungsbetrieb:	Deutsches Zentrum für Luft- und Raumfahrt e.V.
Betreuer des Ausbildungsbetriebs:	Jan Backhaus
Gutachter der Dualen Hochschule:	Dr. Holger Gerhards

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem

Thema

Evaluation und Verbesserung eines Task-Schedulers für eine HPC Anwendung

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.*

* falls beide Fassungen gefordert sind

Köln, den 11. September 2020

Zusammenfassung

Bei der Auslegung von Turbomaschinen wird immer stärker auf die computergestützte numerische Simulation (CFD) gesetzt. Diese Rechnungen sind im Allgemeinen sehr aufwändig, was dazu führt, dass sie parallel abgearbeitet werden müssen, um Ergebnisse in akzeptablen Zeitspannen zu erhalten. Durch immer leistungsfähigere Rechenknoten wird dabei verstärkt auf Shared-Memory Parallelisierung gesetzt. Eine Möglichkeit dies zu implementieren ist die sogenannte Task-Parallelisierung, welche die zu bearbeitende Aufgaben in einzelne kleine Tasks zerlegt welche dann teilweise parallel ausgeführt werden können. Um zu koordinieren, wann welcher Task auf welchem Rechenkern ausgeführt wird, benötigt man einen Scheduler. In dieser Arbeit wird der Task-Scheduler des DLR-Strömungslösers TRACE analysiert und seine Performance evaluiert. Dabei werden vor allem zwei mögliche Probleme aufgezeigt: Die ineffiziente Verwaltung der Tasks, sowie eine möglicherweise nicht-optimale Priorisierung der Tasks. Danach werden einige alternative Datenstrukturen vorgestellt, welche zwar in der Lage sind Skalierungsprobleme zu lösen, jedoch in der aktuellen Konfiguration eine schlechtere Performance aufweisen. Darüber hinaus werden einzelne Sortierungskriterien analysiert und einige alternative Sortierungskriterien evaluiert. Durch diese neue Sortierung wird ein Performancegewinn von $2 - 3\%$ erzielt.

Abstract

The numerical simulation plays an increasingly important role in the aerodynamic design of turbomachines. These calculations are generally very complex which leads to the requirement of parallel computations to be able to get results in an acceptable amount of time. Due to increasingly more powerful computation nodes the shared-memory parallelization is getting more and more into the focus. One method to realize this form of parallelism is the so called task-parallelism. This methods divides the computation into many smaller tasks which could then be partially executed in parallel. A scheduler is then used to control which task is run when and on which core of the system. In this work the task-scheduler of the DLR flow solver TRACE is analyzed and its performance is evaluated. During this analysis two potential performance issues are identified: The inefficient handling of the tasks as well as a possible not optimal prioritization of the tasks. After this analysis, a couple of alternative data structures are presented which are able to solve scaling issues. However they all show a worse performance in the current configuration. Furthermore

the different prioritization criteria are analyzed and evaluated with various alternative metrics. The new task ordering leads to a performance improvement of around $2-3\%$.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

TRACE Turbomachinery Research Aerodynamic Computational Environment

CFD Computational Fluid Dynamics

HPC High Performance Computing

UMA Uniform Memory Access

NUMA Non-Uniform Memory Access

FIFO First-In, First-Out

LIFO Last-In, First-Out

DAG Directed Acyclic Graph

LRU Least-Recently Used

MPI Message Passing Interface

RWS Randomized Work Stealing

JSON Javascript Object Notation

IQR Innerquartile Range

1. Einleitung

Die vorliegende Bachelorarbeit wurde im Deutschen Zentrum für Luft- und Raumfahrt e.V. am Institut für Antriebstechnik in der Abteilung Numerische Methoden in Köln verfasst und behandelt die Analyse und Optimierung eines Task-Schedulers. Um größere Berechnungsprobleme effizient abzuarbeiten wird häufig versucht das Problem in kleinere Unterprobleme zu zerlegen, welche möglicherweise parallel ausgeführt werden können. Traditionell wird dies durch mehrere Prozesse auf mehreren vernetzen Rechenknoten (Cluster) realisiert. Die einzelnen Prozesse tauschen die benötigten Informationen mittels Nachrichten aus. Da heutige Hardware jedoch immer mehr Rechenkerne pro Prozessor bietet, verwendet man hierfür immer häufiger eine Shared-Memory Parallelisierung, also mehrere Threads innerhalb eines Prozesses auf einem Rechenknoten. Die einzelnen Threads teilen sich dabei einen gemeinsamen Speicher. Um zu koordinieren, wann welcher Task auf welchem Rechenkern ausgeführt wird, benötigen solche Anwendungen häufig einen Scheduler. Dieser stellt zum einen sicher, dass bestimmte Programmteile erst dann ausgeführt werden, wenn alle seine Abhängigkeiten erfüllt sind. Außerdem steuert er die Ausführungsreihenfolge, wenn mehrere Teile gleichzeitig ausgeführt werden können. Diese Reihenfolge ist wichtig, da sie einen starken Einfluss auf die Performance der Anwendung besitzt. Im Kontext dieser Arbeit wurde der Scheduler des Strömungslösers TRACE untersucht. Bei TRACE handelt es sich um einen CFD-Löser, welcher auf die Berechnung von Strömungen in Turbomaschinen, beispielsweise Kraftwerksgasturbinen oder Flugzeugtriebwerken, spezialisiert ist. Die Abkürzung CFD steht dabei für Computational Fluid Dynamics. Diese computergestützte Simulation ist erforderlich, da keine analytische Lösung für die zugrunde liegenden Navier-Stokes Gleichungen existiert. Die Lösung muss daher numerisch angenähert werden. Um die Strömung in einem gegebenen Volumen zu berechnen, wird dieses in ein Netz aus einzelnen Zellen eingeteilt. Über viele

Iterationen kann nun die Strömung innerhalb der einzelnen Zellen berechnet werden. Um detaillierte Strömungsphänomene auflösen zu können, muss das Netz oft sehr fein gewählt werden. Die daraus resultierende hohe Anzahl an Zellen führt zu einem hohen Rechenaufwand. Um diesen bewältigen zu können, muss die Berechnung parallelisiert werden. Hierfür wird das Probleme in einzelne kleinere Tasks zerlegt, welche von dem Scheduler nun teilweise Parallel ausgeführt werden können.

1.1. Motivation

Da der Rechenaufwand für einen realistischen Testfall möglicherweise sehr groß werden kann, ist die Optimierung der Performance eines Strömungslösers essentiell. Eine Rechnung kann ohne weiteres mehrere Tage oder sogar Wochen in Anspruch nehmen. Dadurch besitzen bereits kleine Optimierungen ein gewaltiges Potential zur Einsparung von Rechenzeit: Nimmt man beispielsweise an, dass eine Rechnung in 10 Tagen durchgeführt werden kann, so führt bereits eine 1%-Performanceoptimierung zu einer Verkürzung der Laufzeit um

$$1\% \cdot 10 \cdot 24\text{h} = 2.4\text{h}.$$

Besonders für Industriepartner ist diese Zeitersparnis auch eine erhebliche Kosteneinsparung, sowohl durch reduzierten Rechenbedarf, als auch durch eine effizientere Arbeitsweise der Anwender.

1.2. Problemstellung

Ziel dieser Arbeit ist es, die bestehende Prototypen-Implementierung des Task-Schedulers in TRACE zu analysieren. Dabei soll neben der formalen Analyse ein besonderer Schwerpunkt auf der Performance des Schedulers liegen. Darauf aufbauend sollen mögliche Performance-Probleme identifiziert werden. Außerdem sollen Alternativen für die Methoden des Schedulings entworfen werden und deren Performance gegen die Prototypen-Implementierung verglichen werden. Abschließend soll beurteilt

werden inwiefern sich die vorgeschlagenen Alternativen für den Einsatz in TRACE eignen. Hierbei muss gegebenenfalls neben der reinen Performance auch noch auf andere Kriterien, wie beispielsweise den Implementierungs– und Wartungsaufwand, eingegangen werden.

1.3. Stand der Technik

Die Task-basierte Parallelisierung ist ein gängiges Verfahren zur Implementierung von Parallelität im High Performance Computing (HPC)[25]. Einfach ausgedrückt kann man einen Task als eine Liste von Anweisungen, welche, unter Beachtung von Abhängigkeiten, parallel zu anderen Tasks ausgeführt werden können, definieren[25]. Für die Realisierung existieren eine ganze Reihe von Standardimplementierungen. Diese können sowohl als eigene Programmiersprache, Spracherweiterung oder als Bibliothek verfügbar sein[25].

Ein früher Ansatz ist die `Cilk` Erweiterung der Programmiersprache C[3]. Sie erweitert die Sprache primär um `thread` und `spawn` Schlüsselworte, welche das dynamische Erzeugen von Parallelität zur Laufzeit erlauben. Die einzelnen Aufrufe können dabei als Task angesehen werden, welche somit parallel ausgeführt werden können. Die einzelnen Arbeiter-Threads teilen die Arbeit dabei durch das so genante **work stealing** Verfahren zur Laufzeit untereinander dynamisch auf[2]. Hierbei versucht ein Arbeiter-Thread, welcher aktuell keine Aufgabe zum bearbeiten besitzt, einem anderen Thread Arbeit zu stehlen. Bei `Cilk` erfolgt die Auswahl des Threads, von welchen Arbeit gestohlen wird, zufällig. Man spricht daher von Randomized Work Stealing (RWS)[2].

Neben der `Cilk`-Weiterentwicklung `Cilk-Plus` stellt Intel auch eine C++ Bibliothek für die Realisierung von Task-basierter Parallelität zur Verfügung: Threading Building Blocks (TBB)[12]. Auch `TBB` verwendet eine Form des Work-Stealings für die Ausführung der Tasks[12, 22].

Auch die weit verbreitete Parallelisierungserweiterung OpenMP erlaubt bereits seit Version 3.0 die Verwendung von Tasks[1]. Dabei wurden die bestehenden Konstrukte

verändert, so dass diese intern ebenfalls Tasks einsetzen, anstatt Arbeitspakete fest an einen Thread zu koppeln. Darüber hinaus wurde auch ein neues `task` Schlüsselwort geschaffen, welches die explizite Definition von Tasks ermöglicht[1]. OpenMP spezifiziert keine verpflichtende Scheduling-Strategie, sondern überlässt dies zum Großteil der jeweiligen Implementierung[18].

Für den Strömungslöser TRACE wurde eine eigenständige Task-Scheduling Implementierung entworfen. Diese wird in späteren Kapiteln näher beschrieben, orientiert sich aber ebenfalls an dem Konzept des Work Stealings.

2. Grundlagen

Im folgenden Kapitel werden wichtige theoretische Grundlagen für die Arbeit behandelt.

2.1. Scheduling

Um auf einer CPU mehrere Aufgaben parallel ausführen zu können, benötigt man eine Software, welche kontrolliert, wann welche Aufgabe abgearbeitet wird. Diese Komponente bezeichnet man als Scheduler. Er findet sich vor allem in Betriebssystemen, wo er die Ausführung von Prozessen, beziehungsweise Threads, steuert. Auch Anwendungen, welche mehrere Aufgaben parallel auf einem Rechner abarbeiten, benötigen hierfür meist einen Scheduler. Man bezeichnet dies als Shared-Memory Parallelisierung, da sich die parallel ablaufenden Threads einen gemeinsamen Adressraum teilen.

Ein Scheduler generiert aus einem Abhängigkeitsgraphen der auszuführenden Aufgaben einen Ausführungsplan[21] (engl. schedule), welcher die Reihenfolge vorgibt, mit welcher die Aufgaben abgearbeitet werden. Hierbei berücksichtigt der Scheduler möglicherweise die Hardware-Eigenschaften des jeweiligen Systems, um einen möglichst optimalen Plan zu erstellen. Diese Eigenschaften können zum Beispiel grundlegend die Anzahl der verfügbaren Prozessoren sein, aber auch erweiterte Funktionen wie die Topologie des Systems, um eine bestmögliche Lokalität der Aufgaben bei der Ausführung zu gewährleisten.

2.1.1. Task und Taskgraph

Für die Shared-Memory Parallelisierung einer Anwendung gibt es verschiedene Konzepte. Eine Möglichkeit ist beispielsweise die von OpenMP bereitgestellte Schleifenparallelität, bei welcher einzelne Iterationen einer Schleife parallel ausgeführt werden können. Eine andere Möglichkeit ist die so genannte Task-Parallelisierung, welche die Anwendung in einzelne Tasks einteilt. Ein Task bezeichnet dabei ein abgeschlossenes Aufgabenpaket, welches durch den Scheduler ausgeführt werden soll. Ein Task beinhaltet, neben der auszuführenden Funktion und den dafür nötigen Parametern, auch Abhängigkeiten zu anderen Tasks. Ein Task kann erst dann vom Scheduler ausgeführt werden, wenn alle seine Abhängigkeiten erfüllt sind. Die Abhängigkeiten zwischen den einzelnen Tasks lassen sich hierzu in Form eines Task-Graphens darstellen. Hierbei handelt es sich typischerweise um einen azyklischen, gerichteten Graphen (DAG). Details zu der Datenstruktur werden in Abschnitt 2.1.2 näher erläutert.

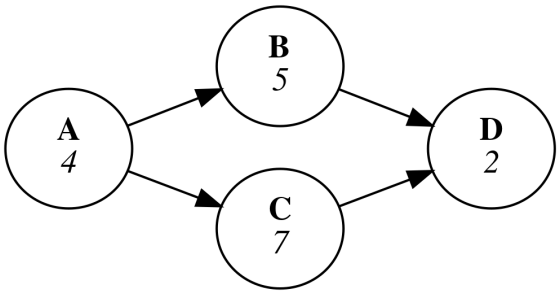


Abbildung 2.1.: Beispielhafter Task DAG mit vier Tasks

Ein beispielhafter Task-Graph ist in Abbildung 2.1 dargestellt. Die Buchstaben stellen den Namen der Tasks dar, während die Zahl darunter den Aufwand der Tasks in Zeiteinheiten angibt. Man erkennt, dass zu Beginn nur Task `A` ausgeführt werden kann, da alle anderen Tasks direkt oder indirekt von ihm abhängig sind. Nach der Ausführung von `A` können nun `B` und `C` gleichzeitig ausgeführt werden, da sie nicht voneinander abhängen. Zuletzt kann `D` ausgeführt werden, sobald `B` und `C` fertig sind.

Eine mögliche Ausführung des Taks-Graphens ist in Abbildung 2.2 dargestellt. Man erkennt, dass die Ausführung auch durch das Hinzufügen weiterer Threads nicht

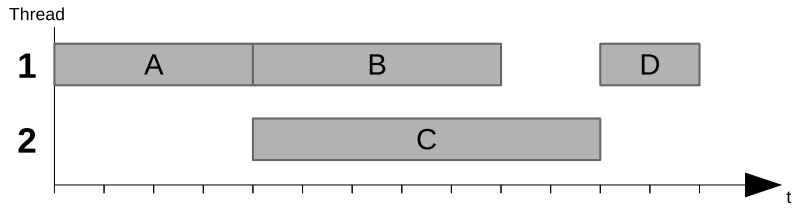


Abbildung 2.2.: Mögliches Gantt-Diagramm zu dem Beispiel-Taskgraph

weiter parallelisiert werden kann. Für die Ausführungszeit eines parallelisierten Programms gilt daher stets

$$T_p \geq T_\infty \tag{2.1}$$

wobei T_p die Ausführungszeit auf p Prozessoren und T_∞ die Zeit auf beliebig vielen Prozessoren bezeichnet[14]. Letzteres wird auch als kritischer Pfad bezeichnet und ist die längste serielle Abfolge von Tasks[14]. In dem Beispiel aus Abbildung 2.2 ist der kritische Pfad `A->C->D`. Der Pfad `A->B->D` erfüllt diese Kriterium nicht, da er zwar auch aus drei Knoten besteht, jedoch die Gesamtsumme der Laufzeit kleiner ist. Daraus folgt, dass das Beispielproblem unabhängig von Scheduling-Strategie und Ressourcen nicht in unter 13 Zeiteinheiten bewältigt werden kann. Diese Eigenschaft eines Taskgraphens kann als Parallelität angegeben werden:

$$P = \frac{T_1}{T_\infty} \tag{2.2}$$

Bei der Gleichung bezeichnet T_1 die Ausführzeit auf einem Prozessor bzw. die Gesamtsumme aller Tasks[14]. Für das Beispiel ergibt sich daher eine Parallelität von $P = \frac{18}{13} \approx 1.38$. Es stehen somit durchschnittlich 1.38 Tasks gleichzeitig zur Ausführung bereit. Damit eine Anwendung effizient auf vielen Threads ausgeführt werden kann, müssen genug Tasks vorhanden sein, um die vorhandenen Ressourcen auszulasten.

2.1.2. Datenstrukturen

Wie bereits in Abschnitt 2.1.1 erläutert ist ein Graph eine essentielle Datenstruktur für das Scheduling von Tasks mit Abhängigkeiten. Neben einem Graphen benötigen die meisten Scheduling-Algorithmen noch eine weitere Datenstruktur, die Queue beziehungsweise Deque. Diese Datenstrukturen sollen in den nachfolgenden Unterkapiteln vorgestellt werden.

Graph

Anders als in einer Liste oder einem Array besitzen die Elemente in einem Graph keine lineare Ordnung. Stattdessen sind die einzelnen Elemente als Knoten abgespeichert, welche beliebig viele Verbindungen mit anderen Knoten besitzen können. Diese Verbindungen werden als Kanten bezeichnet [10]. Die Kanten geben an, welche Knoten von einem gegebenen Knoten erreicht werden können. Man unterscheidet zwischen gerichteten Bäumen, in welchen die Kanten nur in eine Richtung traversiert werden können, und ungerichteten Bäumen, in welchen Kanten in beide Richtungen gültig sind [10].

Für die Speicherung eines Graphen verwendet man typischerweise entweder eine Verkettung zwischen den Knoten analog zu einer verketteten Liste oder eine Adjazenzmatrix [10]. In Letzterer geben die Elemente einer $N \times N$ Matrix an, welche Knoten miteinander verbunden sind. N bezeichnet dabei die Gesamtzahl der Knoten. Ein Wert > 0 an der Stelle $M_{i,j}$ gibt an, dass eine Verbindung von Knoten i nach j existiert. Für den in Abbildung 2.1 gezeigten Graphen könnte die Adjazenzmatrix wie folgt aussehen:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.3}$$

Die vier Kanten in dem Graphen sind hier durch die vier Einsen wiedergegeben. Eine weitere wichtige Eigenschaft von Graphen ist das mögliche Vorhandensein von Zyklen. Als Zyklus bezeichnet man eine Abfolge von Knoten welche traversiert werden können, wobei der letzte Knoten wieder dem Startknoten entspricht. Diese Eigenschaft ist dementsprechend ausschließlich für gerichtete Graphen relevant, da im ungerichteten Fall zwei verbundene Knoten automatisch einen Zyklus bilden.

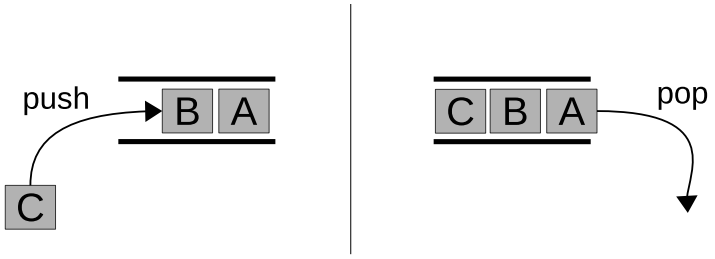
Im Kontext eines Schedulers werden vor allem gerichtete, azyklische Graphen (Directed Acyclic Graph (DAG)) eingesetzt. Die Richtung gibt dabei meist die Abhängigkeit an: Eine Kante $A \rightarrow B$ impliziert das B abhängig von A ist. Der Graph darf keine Zyklen enthalten, weil die Abhängigkeiten ansonsten nicht erfüllt werden können, da sich die Knoten des Zyklus gegenseitig blockieren.

Queue und Stack

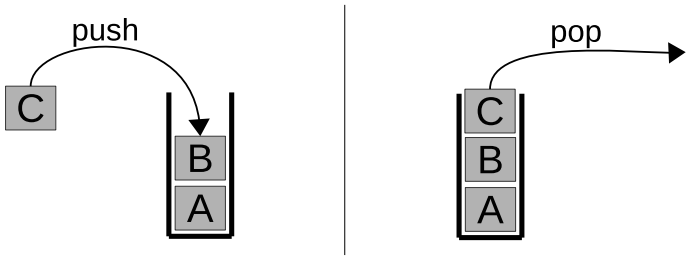
Eine Queue ist eine Datenstruktur, welche eine Warteschlange implementiert. Sie besitzt dabei einen Anfang und ein Ende. Wie in Abbildung 2.3a dargestellt können neue Einträge lediglich am Ende eingefügt werden, während bestehende Einträge nur am Anfang entnommen werden können. Auf die Elemente dazwischen kann nicht zugegriffen werden [10]. Man bezeichnet diese Art der Datenspeicherung als First-In, First-Out (FIFO), da die Einträge in der gleichen Reihenfolge entnommen werden, in welcher sie eingefügt wurden.

Das Gegenstück der Queue ist der Stack. Bei diesem werden die Einfüge- und Entnahmeoperationen an der selben Seite durchgeführt (siehe Abbildung 2.3b). Da dies dazu führt, dass die zuletzt eingefügten Elemente als erstes wieder entnommen werden, spricht man hier von einer Last-In, First-Out (LIFO) Speicherung [10].

Beide Datenstrukturen können in Form eines Arrays implementiert werden. Dafür werden ein (Stack), beziehungsweise zwei (Queue), Zeiger benötigt, welche auf die Elemente des Arrays zeigen und dabei die Einfüge- und die Entnahmeposition markieren [10]. Hierbei ist zu beachten, dass das Array entweder eine feste Größe besitzt und damit die Anzahl der Elemente die zur gleichen Zeit in der Datenstruktur



(a) Queue Operationen



(b) Stack Operationen

Abbildung 2.3.: Einfüge- und Entnahmeoperatoren auf Queues/Stacks

vorhanden sein können begrenzt ist, oder beim Einfügen eines neuen Eintrages das Array gegebenenfalls erweitert werden muss.

Deque

Bei der Deque (engl. für Double Ended Queue) handelt es sich um eine Fusion aus Stack und Queue[10]. Das bedeutet, dass an beiden Enden der Warteschlange Einträge sowohl eingefügt als auch entnommen werden können. Dies ist in Abbildung 2.4 dargestellt. Diese Eigenschaft ermöglicht den gleichzeitigen Zugriff auf das älteste, beziehungsweise das jüngste, Element in einer Warteschlange. Wie schon bei den zugrundeliegenden Strukturen kann auch hier nicht auf die Elemente in der Mitte zugegriffen werden[10].

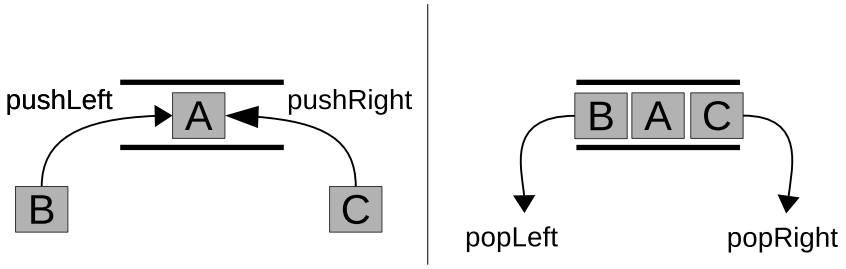


Abbildung 2.4.: Einfüge- und Entnahmeoperatoren auf einer Deque

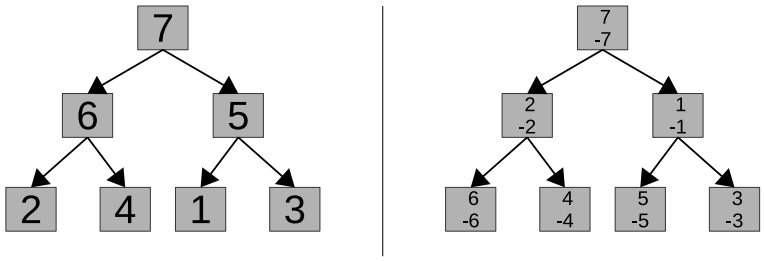
Heap und Prioritätsqueue

Besonders bei Schedulern kann es wünschenswert sein, nicht das älteste Element einer Queue zu entnehmen, sondern das mit der höchsten Priorität. Daher verwendet man statt einer regulären Queue möglicherweise eine Prioritätsqueue. Diese kann in Form eines Heaps implementiert werden[10].

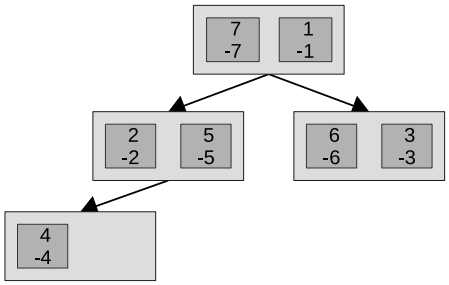
Ein Heap ist ein vollständiger binärer Baum für den die Heap-Bedingung an jedem Knoten erfüllt ist. Die Heap-Bedingung besagt im Falle eines Max-Heaps, dass der Wert eines Knotens größer oder gleich allen Elementen unter ihm sein muss[10]. Daraus folgt, dass das Element mit der höchsten Priorität stets in der Wurzel des Baums zu finden ist. Im Gegensatz zu einer regulären Queue bedeutet dies aber auch, dass bei jedem einfügen bzw. entfernen von Knoten der Baum umstrukturiert werden muss, damit die Heap-Bedingung erfüllt bleibt[10]. Dies ist beispielsweise im linken Baum in Abbildung 2.5a dargestellt: Der Wurzelknoten besitzt mit 7 den größten Wert des gesamten Baumes. Ebenso besitzt beispielsweise Knoten 6 den größten Wert des Unterbaumes dessen Wurzel er bildet.

k-D-Heap

Im vorherigen Abschnitt wurden die Eigenschaften eines klassischen Heaps dargestellt. Die einzelnen Elemente werden anhand eines Schlüssels sortiert. Eine Modifikation



(a) Heap und k-D Heap



(b) Optimierung des k-D Heap für große k

Abbildung 2.5.: Übersicht über verschiedene Heap Varianten

dieses Konzepts ist der k-D-Heap[7], welche eine multidimensionale Sortierung innerhalb eines Heaps erlaubt. Das bedeutet, dass jedem Eintrag k Schlüssel zugeordnet sind. Beim Entfernen des höchsten Elementes muss nun zusätzlich der Schlüsselindex, dessen Maximum entfernt werden soll, angegeben werden.

Die wesentliche Änderung des k-D-Heaps ist es, dass die Heap Bedingung nun abhängig von der Ebene des Eintrags im binären Baum ist. Für jeden Knoten der Ebene i gilt, dass der Schlüssel $i \mod k$ größer als die Schlüssel $i \mod k$ aller untergeordneten Einträge ist[7]. In der zweiten Abbildung in Abbildung 2.5a ist ein solcher k-D Heap dargestellt. Genauer handelt es sich um einen 2-D Heap, da jeder Knoten zwei Schlüssel enthält. Dadurch, dass der zweite Schlüssel in diesem Fall der negierte erste Schlüssel ist, wird hier ein Min-Max-Heap implementiert, welcher erlaubt, den größten und den kleinsten Eintrag zu lokalisieren[7]. Man sieht, dass der Wurzelknoten das Schlüsselpaar $\{7, -7\}$ besitzt. Da der Knoten auf Ebene $i = 0$ ist, muss der erste Schlüssel größer sein als alle anderen ersten Schlüssel des Heaps. Dies

ist erfüllt, da kein Eintrag größer sieben existiert. Die Knoten in der zweiten Ebene ($i = 1$) müssen die Heap-Bedingung nun für den zweiten Schlüssel erfüllen. Da -2 und -1 größer sind als die jeweils untergeordneten Schlüssel ($-6, -4$ und $-5, -3$) ist auch hier die Bedingung erfüllt.

Die modifizierte Heap-Bedingung führt dazu, dass der optimale Eintrag für den Schlüssel i stets unter den ersten $i + 1$ Ebenen zu finden ist. Durch die Eigenschaft des binären Baums müssen daher

$$2^i + 1 \tag{2.4}$$

Einträge des Heaps durchsucht werden um das Optimum für einen Schlüssel i zu lokalisieren. Dies führt dazu, dass für eine große Anzahl an Schlüsseln sehr viele Einträge durchsucht werden müssen. Für den Fall, dass beispielsweise das Optimum des 20. Schlüssels entfernt werden soll, müssen nach Gleichung (2.4) $2^{19} + 1 = 524.289$ Einträge durchsucht werden. Dies führt dazu, dass oft der ganze Heap durchsucht werden muss und somit der Vorteil des Heaps, das Lokalisieren und entfernen des Optimums von $\mathcal{O}(\log(n))$[10], immer weiter in Richtung $\mathcal{O}(n)$ verschoben wird.

Um dieses Problem zu lösen, stellen die Autoren des k-D-Heaps eine weitere Möglichkeit vor, diesen zu modifizeren um ihn effizient für große Schlüsselanzahlen k verwenden zu können[7]. Dazu werden in jedem Knoten des Baums k Einträge gespeichert, wobei jeweils das Element mit Index i die Heap-Bedingung für den Schlüssel i erfüllt. Um nun das Optimum für einen Schlüssel i zu finden, muss lediglich der erste Knoten des Baumes durchsucht werden. Damit werden maximal k Einträge überprüft, was im vorherigen Beispiel von $k = 20$ einer Verbesserung um mehrere Größenordnungen entspricht. Allgemein gilt hier, dass das Entfernen eines Optimums in $\mathcal{O}(k^2 \log(n))$ möglich ist[7]. Der k-D-Heap erlaubt es somit eine Prioritätsqueue für mehrere unterschiedliche Prioritäten gleichzeitig effizient zu implementieren. Für einen Scheduler bedeutet dies, dass die Tasks gleichzeitig nach mehreren Kriterien sortiert gespeichert werden können. In Abbildung 2.5b ist ein beispielhafter 2-D-Heap mit der Optimierung für große Schlüsselanzahlen dargestellt. Er definiert den gleichen Heap wie der in der zweiten Abbildung in Abbildung 2.5a. Die beiden Heap-Bedingungen, welche im vorherigen Abschnitt überprüft wurden, gelten nun auch hier für die beiden Einträge im Wurzelknoten.

2.1.3. Kriterien für einen Task-Scheduler

Obwohl ein Scheduler sowohl in einem Betriebssystem, als auch in einer Task-Parallelen Shared-Memory Anwendung verwendet wird, unterscheiden sich die Anforderungen der beiden Scheduler teilweise deutlich voneinander. Für den Scheduler in einem Betriebssystem ist zum Beispiel die Reaktionszeit des Schedulers essentiell. Diese gibt an, wie lange ein zur Ausführung fertiger Task durchschnittlich warten muss, bis er ausgeführt wird. Für ein interaktives System ist es wichtig, dass Eingaben des Benutzers möglichst schnell verarbeitet werden[24]. Für ein HPC System ist es jedoch weniger relevant wie lange ein einzelner Task auf seine Ausführung wartet, da primär die Gesamtlaufzeit aller Tasks relevant ist.

Task-Parallele HPC-Systeme sind am ehesten mit klassischen Stapelverarbeitungssystemen vergleichbar. Für einen solchen Scheduler im HPC-Kontext sind daher vor allem die folgenden Kriterien ausschlaggebend[24]:

Auslastung Als Auslassung wird bezeichnet, zu welchem Prozentsatz eine CPU aktiv ist. Ein idealer Scheduler lastet eine CPU durchgehend zu 100% aus.

Durchsatz Der Durchsatz gibt an, wie viele Tasks in einer bestimmten Zeiteinheit verarbeitet werden können.

Overhead Als Overhead wird der prozentuale Anteil der Laufzeit bezeichnet, welcher durch Scheduling-Routinen verursacht wird.

Es ist wichtig zu beachten, dass die Kriterien stets gemeinsam betrachtet werden müssen, da die Aussagekraft einzelner Kriterien alleine stark eingeschränkt ist. Eine CPU-Auslastung von 100% gibt zwar an, dass die CPU vollständig ausgelastet wird, jedoch gibt es keine Aussage darüber mit welcher Art von Arbeit die CPU ausgelastet wird. Verbringt der Scheduler den Großteil der Zeit in Busy-Wait Routinen, so führt dies ebenfalls zu einer hohen CPU-Auslastung. Es ist daher nicht möglich aus der CPU-Auslastung Rückschlüsse auf das effiziente Abarbeiten von Tasks zu ziehen[24].

Wichtig ist jedoch, dass primär die Gesamtlaufzeit des Schedulers als Optimierungsgröße betrachtet wird, da diese der ausschlaggebende Faktor ist. Die anderen Metriken leisten dabei Unterstützung bei der Detektion möglicher Performance-Probleme. Die

Gesamtlaufzeit steht für klassische Stapelverarbeitungssysteme nicht zur Verfügung, da es keine begrenzte Menge an Tasks gibt. Daher wird hier oft die Durchlaufzeit (wie lange benötigt ein Task im Durchschnitt) betrachtet[24].

Die gesonderte Betrachtung des Durchsatzes ist für den Scheduler hingegen weniger wichtig, da die Anzahl der Tasks fix ist und daher der Durchsatz stets durch den Quotienten aus Tasks und Laufzeit gegeben ist.

2.1.4. Work-Stealing

Eine wichtige Funktion eines Task-Schedulers ist es, die Ausführung der Tasks so zu koordinieren, dass alle Threads stets ausgelastet sind, sofern dies durch die Anzahl an Tasks möglich ist. Das Verteilen der Tasks auf die einzelne Threads bezeichnet man dabei als Load Balancing. Für dieses gibt es eine Reihe von verschiedenen Strategien, welche sich dabei durch zwei Parameter beschreiben lassen: synchron/asynchron und reaktiv/proaktiv[27]. Proaktive Strategien versuchen das Load-Balancing durch eine effiziente Verteilung neuer Tasks zu realisieren während reaktive Strategien versuchen auf eine potentielle Imbalance zu reagieren. Ein Scheduler kann dabei eine Kombination aus reaktiven und proaktiven Strategien verwenden[27]. Die Synchronizität gibt an, wie die Threads während des Load-Balancings miteinander interagieren.

Das sogenannte Work-Stealing bezeichnet eine asynchrone, reaktive Strategie[27]. Neue Tasks werden dem ausführenden Thread zugeordnet. Sollte ein Thread feststellen, dass er keine Tasks mehr ausführen kann, so versucht er Tasks aus den Queues von anderen Threads zu stehlen. Auch diese Mechanik dient dem Ausgleich der Last der einzelnen Threads, um ein gutes Load Balancing zu erreichen [2]. Der Vorteil von diesem Ansatz ist, dass das eigentliche Scheduling nur von den Threads durchgeführt werden muss, welche aktuell keine andere Arbeit besitzen während Threads mit Tasks diese ungestört ausführen können[27]. In einem vollständig ausgelasteten System ist daher kein Austausch zwischen den Threads nötig [2]. Für die Implementierung eines solchen Work-Stealing Schedulers kann beispielsweise statt einer Task-Queue eine Deque für jeden Worker verwendet werden: Neue Tasks werden vorne eingefügt und der Worker entnimmt aus seiner eigenen Deque stets den vordersten (damit

neuesten) Task und führt diesen aus. Versucht er hingegen bei anderen Workern Arbeit zu stehlen, so entnimmt er das älteste Element am hinteren Ende [2].

2.2. Speicher

Die Berechnung von Strömungsphänomenen mit TRACE benötigt viel Arbeitsspeicher. Dabei können einzelne Rechnungen ohne weiteres viele Gigabyte Arbeitsspeicher belegen. Aus der Sicht des Schedulers ist dies wichtig zu beachten, damit die Tasks möglichst effizient ausgeführt werden können. Die Ziele hierbei sind vor allem lokale Speicherzugriffe auf NUMA Systemen[23] (siehe Abschnitt 2.2.1), aber auch eine möglichst effiziente Ausnutzung der Caches des Systems[5] (siehe Abschnitt 2.2.2), um so die Performance der Speicherzugriffe zu optimieren und daher die Ausführungszeit so gering wie möglich zu halten. Lokale Speicherzugriffe sind durch die hohen Datenmengen besonders relevant, da es ansonsten schnell zu einer Überlastung des Non-Uniform Memory Access (NUMA)-Verbindungsnetzes kommt, welche die Performance reduziert.

2.2.1. Vergleich NUMA und UMA

Im der klassischen Rechnerarchitektur nach von Neumann ist einem Prozessor ein uniformer Speicher über einen Bus zugeordnet. Durch den steigenden Bedarf an paralleler Verarbeitung werden jedoch meistens mehr als ein Rechenkern installiert. Dies führt dazu, dass alle Prozessoren auf den gleichen Speicherbus zugreifen. Hier spricht man von Uniform Memory Access (UMA), da, unabhängig von Prozessor und Adresse, jeder Speicherzugriff die gleichen Laufzeitkosten besitzt.

Konkurrieren nun viele Prozessoren auf dem gleichen Bus, so kann es vorkommen, dass die Bandbreite so stark ausgenutzt wird, dass die einzelnen Prozessoren nicht mehr ausreichend mit Daten versorgt werden können [17]. Eine Möglichkeit dieses Problem zu umgehen ist es den Speicher aufzuteilen. Das bedeutet, dass jeder Prozessor nur noch auf einen Teil des Speichers zugreifen kann. Er muss dafür über einen eigenen Datenbus mit diesem Speicher verbunden werden. Diesen Speicher bezeichnet man als

lokal. Da nur noch ein Teil aller Prozessoren an diesen Bus angebunden sind, ist die Zahl der konkurrierenden Speicherzugriffe geringer. Jedoch kann es erforderlich sein, dass ein Prozessor auf Daten im Speicher der anderen Prozessoren zugreifen muss. Hierfür gibt es die Möglichkeit über den Speichercontroller auf den so genannten Remote-Speicher zuzugreifen [17]. Der Zugriff auf Remote-Speicher ist jedoch deutlich aufwändiger, da die Kommunikation über ein Verbindungsnetzwerk durchgeführt werden muss [13]. Dies führt zu einer deutlich höheren Zugriffszeit. Darüber hinaus führt jeder Zugriff über das Verbindungsnetzwerk wieder zu Konkurrenz um den Zugriff auf diesem geteilten Bus [13].

Da hier kein uniformer Speicherzugriff mehr möglich ist und die Speicherperformance sowohl von Prozessor als auch von der gewählten Speicherstelle abhängt, spricht man hier von Non-Uniform Memory Access (NUMA) Systemen.

Ein Task-Scheduler kann dazu beitragen Remote-Speicherzugriffe zu reduzieren, indem er die Tasks den Threads zuweist, welche auf einem Prozessor ausgeführt werden, der mit den entsprechenden Daten lokal verbunden ist. Hierfür benötigt der Scheduler jedoch detaillierte Informationen über die Speicherarchitektur von der Laufzeitumgebung sowie der Affinitäten der einzelnen Tasks.

2.2.2. Cache

Viele CPU-Instruktionen müssen während der Ausführung auf Daten im Speicher zugreifen. Da der Zugriff auf den Hauptspeicher im Vergleich zu heutigen CPU-Taktraten langsam ist, werden bei modernen Prozessoren zusätzlich Caches verwendet. Bevor Daten aus dem Hauptspeicher geladen werden, wird geprüft, ob die Daten im Cache vorhanden sind. Erst wenn die Daten nicht im Cache verfügbar sind wird ein Zugriff auf den Hauptspeicher durchgeführt. Sind die Daten beim Zugriff im Cache verfügbar so spricht man von einem Cache-Hit, ansonsten von einem Cache-Miss[8].

Da die Größe des Caches stark begrenzt ist, werden nicht mehr benötigte Einträge wieder aus dem Cache entfernt sobald neue Daten in einen vollen Cache geladen werden sollen. Man spricht in dem Fall von Verdrängung[8]. Hierbei sollten idealerweise die Einträge entfernt werden, die nicht mehr benötigt werden. Da es jedoch

unmöglich ist dies vorherzusagen, ohne das gesamte Programm und alle verarbeiteten Daten zu analysieren, muss eine Heuristik verwendet werden. Es ist ebenfalls zu beachten, dass die Verdrängung nicht aufwändig sein darf, da sie gegebenenfalls bei jedem Speicherzugriff angewandt werden muss.

Für die Verdrängung gibt es verschiedene Strategien, die gebräuchlichsten dabei sind[8]:

Last-Recently-Used Beim Laden von Daten in den Cache wird der Eintrag aus dem Cache verdrängt, welcher am längsten nicht mehr benutzt wurde.

Random Beim Laden von Daten in den Cache wird ein zufälliger Eintrag verdrängt.

Der verbreitete Standardalgorithmus ist dabei jedoch Least-Recently Used (LRU)[20]. Ein wiederholter Datenzugriff ist daher zu bevorzugen, um die maximale Cache-Performance ausnutzen zu können.

Für einen Task-Scheduler bedeutet dies, dass bevorzugt die Tasks hintereinander auf dem gleichen Prozessorkern ausgeführt werden sollten, welche auf die gleichen Daten zugreifen, da so die Wahrscheinlichkeit steigt, dass die vom nachfolgenden Task benötigten Daten noch zum Teil im Cache liegen und so die Laufzeit durch reduzierte Speicheranfragen weiter reduziert werden kann[5].

Um die Cache-Effizienz anzugeben wird die Cache-Miss-Rate[8] als gebräuchliche Metrik verwendet. Sie wird durch folgenden Gleichung bestimmt:

$$\text{miss}_{\%} = 100 \cdot \frac{n_{\text{miss}}}{n} \tag{2.5}$$

wobei n die Gesamtsumme der Cache-Zugriffe und n_{miss} die Anzahl der Cache-Misses während einer Programmausführung darstellt. Je näher die Cache-Miss-Rate gegen 0% läuft desto besser ist die Cache-Performance der untersuchten Anwendung.

3. Methodik

In diesem Kapitel werden die verwendeten Testfälle, Analysewerkzeuge und Auswertungsmethoden vorgestellt werden, welche verwendet werden um die aktuelle Scheduler-Implementierung, sowie Modifikationen zu evaluieren.

3.1. Testfälle

Für die Beurteilung des Task-Schedulers in TRACE werden insgesamt drei verschiedene Testfälle verwendet. Diese umfassen ein Minimalbeispiel eines Stömungslösers und zwei prozedural generierte synthetische Testfälle. Task-Graphen der vorgestellten Testfälle sind in Abbildung A.1 beispielhaft dargestellt.

3.1.1. TRACE-Minimalbeispiel

Das TRACE Minimalbeispiel wird im Rahmen eines laufenden Optimierungsprojektes als Grundlage für verschiedene Experimente zur Evaluation von möglichen Performanceverbesserungen verwendet. Es handelt sich dabei um einen Poisson-Löser, welcher die in TRACE vorhanden Algorithmen und Datenstrukturen verwendet um ein vereinfachtes Testproblem zu lösen. Dieses lässt sich beliebig groß generieren und dabei in beliebig viele Teile zerlegen, um eine parallele Abarbeitung zu ermöglichen. Ein wichtiger Unterschied zur realen TRACE Version ist, dass die einzelnen Tasks relativ feingranular und ausgewogen sind.

Die Ausführung wird über zwei Parameter $\{l, p\}$ konfiguriert. Der erste Parameter l gibt die Kantenlänge eines zu berechnenden Würfels an, der zweite (p) gibt die

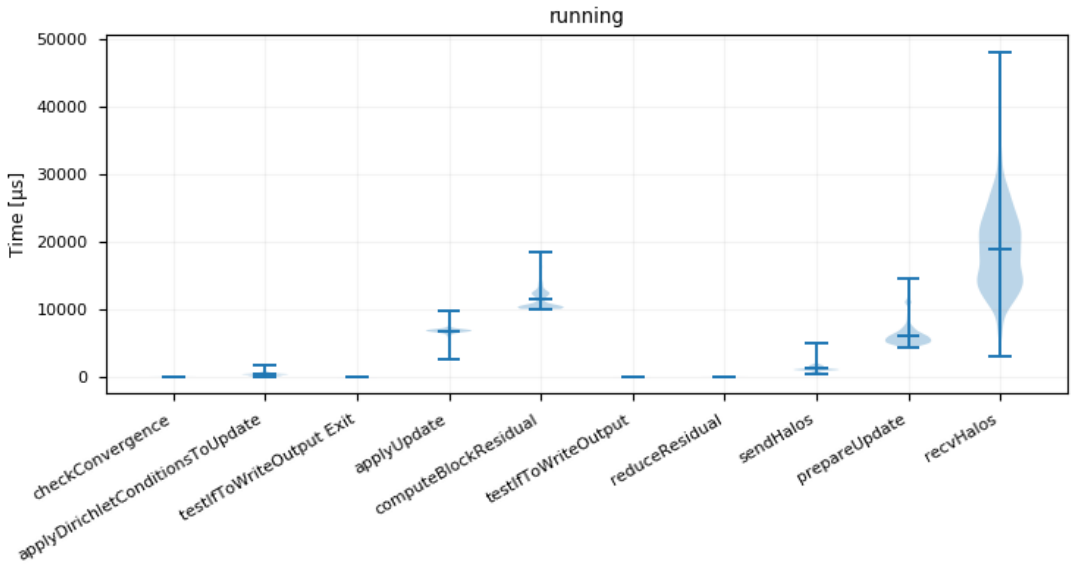


Abbildung 3.1.: Task Laufzeiten im TRACE-Minimalbeispiel

Anzahl an zu verwendende Threads an. Da es sich um die Kantenlänge eine Würfels handelt, muss beachtet werden, dass die tatsächliche Problemgröße mit $\mathcal{O}(l^3)$ wächst. In Abbildung 3.1 ist die Verteilung der Laufzeiten und der Durchschnitt der Tasks aus dem Minimalbeispiel mit den Parametern $\{400, 20\}$ dargestellt. Auffällig ist dabei vor allem die starke Varianz des `recvHalos` Tasks, was darauf zurückzuführen ist, dass dieser Task die von anderen Threads berechneten Daten empfängt. Darüber hinaus ist dieser Task auch der mit der mit Abstand längsten Laufzeit, da Tasks, welche Daten empfangen, möglichst früh gestartet werden, um auf die Daten zu warten. Zusätzlich gibt es noch einzelne weitere Tasks mit einer erhöhten Laufzeit und eine Reihe an Tasks mit einer im Vergleich verschwindend geringen Laufzeit.

Der Vorteil dieses Testfalls ist die sehr übersichtliche Länge des Quelltextes. Gleichzeitig ist das Laufzeitverhalten (vor allem durch die vorhandene MPI-Kompatibilität) mit dem von TRACE vergleichbar. Ein weiterer großer Vorteil ist, dass die Ausführzeit über die Parameter gut kontrollierbar ist und eine Ausführung so auf wenige Sekunden beschränkt werden kann. Dies erleichtert es, eine große Stichprobenanzahl bei Messungen zu generieren.

3.1.2. Parallele-Regionen

Der zweite Testfall generiert mit zwei Parameter n_l und n_p einen Task-Graphen welcher aus n_l Regionen besteht, die sequentiell aufeinander folgen. Dafür ist nach jeder Region ein Synchronisierungs-Task eingefügt, welcher von allen Tasks der vorherigen Region abhängt und eine Abhängigkeit für alle Tasks der nächsten Region bildet. Jede dieser Regionen enthält n_p vollständig parallele Tasks. Durch die Gleichung

$$n = n_l(n_p + 1) \tag{3.1}$$

ergibt sich die gesamte Anzahl an Tasks n des generierten Graphen. Der Testfall eignet sich gut um die Skalierung des Schedulers zu testen, da mit Hilfe des Parameters n_p die Anzahl an gleichzeitig aktiven Tasks direkt kontrollierbar ist. Die einzelnen Tasks führen dabei keinerlei Berechnung aus. sondern warten für eine einstellbare Zeit. So wird verhindert, dass die Laufzeit durch andere Faktoren wie den Speicher beeinflusst wird.

3.1.3. G(n,p)-Methode

Der dritte Testfall generiert einen Abhängigkeitsgraphen durch die $G(n, p)$ Methode von Erdös-Rényi[6]: Durch die beiden Parameter n und p wird zunächst eine Adjazenzmatrix der Größe $n \times n$ erzeugt und alle Felder mit einer Null belegt. Anschließend wird für jedes Element der unteren Dreiecksmatrix, welches nicht Teil der Hauptdiagonale ist, eine annähernd gleichverteilte Zufallszahl $r \in [0, 1]$ generiert. Gilt $r < p$ so wird an dieser Stelle der Dreiecksmatrix eine Eins eingetragen. Somit steuert der Parameter n die Anzahl an Tasks und p gibt an wie viele Abhängigkeiten durchschnittlich generiert werden.

Wenn p gegen Eins läuft, so konvergiert der Abhängigkeitsgraph mehr und mehr zu einem Reihe aus n sequentiellen Tasks. Läuft p gegen Null, so verschwinden alle Abhängigkeiten aus dem Graphen. Die Steuerung des Parameters erlaubt eine Evaluation des Schedulers für verschiedene Level an Parallelität des Taskgraphen.

Der Zufallsgenerator, welcher für die Kantengenerierung verwendet wird, wird vorher über einen weiteren Seed Parameter initialisiert, um eine reproduzierbare Graphenerzeugung zu gewährleisten.

3.2. Messungen

Um die Performance der jeweiligen Scheduler-Implementierungen zu bewerten werden eine Reihe von Messungen durchgeführt. Dieser Abschnitt bietet einen Überblick über die verwendeten Werkzeuge, sowie die Methodik der Messverfahren die dabei zum Einsatz kamen.

3.2.1. Statistik

Damit die Ergebnisse der Untersuchung sinnvoll verglichen werden können, müssen die Ergebnisse statistisch signifikant sein. Der wichtigste Aspekt dabei ist, eine ausreichende Menge an Stichproben zu erfassen. Da nicht davon ausgegangen werden kann, dass die gemessenen Laufzeiten und andere Performance-Metriken Normalverteilt sind, ist es nicht möglich die benötigte Anzahl an Stichproben vorher zu berechnen[11]. Stattdessen wird die in[11] beschriebene Gleichung verwendet, um nach k Messungen zu Überprüfen, ob das 95% Konfidenzintervall um den Median kleiner als die vorgegebene Toleranz ist. Die Intervallgrenzen ergebenen sich dabei wie folgt:

$$l = \left\lfloor \frac{n - z(0.5\alpha)\sqrt{n}}{2} \right\rfloor \tag{3.2}$$

$$u = \left\lceil 1 + \frac{n + z(0.5\alpha)\sqrt{n}}{2} \right\rceil \tag{3.3}$$

Hierbei ist n die Anzahl der bisher aufgenommenen Stichproben, α der Anteil der nicht im Konfidenzintervall enthaltenen Daten und $z(x)$ die Normalverteilung. Für

das 95% Konfidenzintervall gilt daher $z(0.025) = 1.96$[11]. Dabei ergibt sich das Konfidenzintervall für ein Array A aus n sortierten Messwerten durch:

$$[A[l]; A[u]] \tag{3.4}$$

wobei l und u durch die Gleichungen (3.2) beziehungsweise (3.3) bestimmt werden.

Um zu überprüfen, ob die aktuelle Anzahl an Stichproben ausreichend ist, muss lediglich überprüft werden, ob die Länge des Intervalls kleiner ist als ein vorher eingestellter Grenzwert. Es müssen dementsprechend so lange neue Stichproben aufgenommen werden, bis folgende (Un-)Gleichung erfüllt ist:

$$G >= A[u] - A[l] \tag{3.5}$$

Dabei bezeichnet G den vorher definierten Grenzwert. Grundsätzlich gilt, je kleiner G ist, desto höher ist die Genauigkeit der Messung, jedoch werden auch deutlich mehr Messungen benötigt. Bei den jeweiligen Messungen in dieser Arbeit wird stets die Maximalgröße des 95% Konfidenzintervalls angegeben. Aus dem angewendeten Verfahren für die Bestimmung der Anzahl der zu erfassenden Stichproben folgt, dass für die einzelnen Testfälle innerhalb einer Messung unterschiedlich viele Stichproben aufgenommen werden. Bevor die Abbruchbedingung überprüft wird, müssen vorher mindestens 15 Stichproben genommen werden. Nach maximal 1000 Stichproben wird die Erfassung ebenfalls beendet, auch wenn die benötigte Genauigkeit noch nicht erreicht wurde.

Für die Auswertung der Messung wird ein möglicher Unterschied als statistisch signifikant betrachtet, wenn sich die Konfidenzintervalle der zu vergleichenden Messungen nicht überlappen[11]. Zur Visualisierung werden die Daten in dieser Arbeit häufig mit einem Boxplot dargestellt. Ein Beispiel für einen solchen Boxplot ist in Abbildung 3.2 abgebildet.

Dieser stellt neben dem Median zusätzlich die Verteilung der Daten graphisch da. Dabei besteht er aus einer Box und zwei Antennen. Die sortierten Daten werde für die Box in vier gleiche Teile eingeteilt[19]. Das untere Ende der Box entspricht

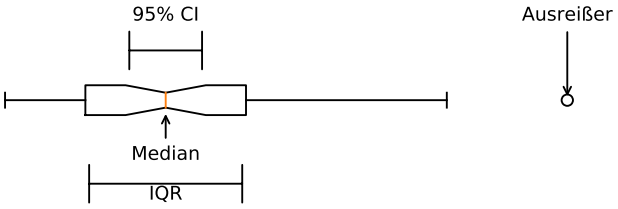


Abbildung 3.2.: Beispielhafter Boxplot

dabei dem Wert nach 25% (erstes Perzentil) aller Messwerte und das obere nach 75% (drittes Perzentil). Die Linie innerhalb der Box markiert den Median, also den Wert nach 50% aller Messwerten. Innerhalb der Box sind dementsprechend die mittleren 50% der Daten enthalten. Die Länge der Box bezeichnet man als Innerquartile Range (IQR). Die beiden Enden der Antennen befinden sich jeweils beim letzten Datenelement, welches maximal $1.5 \times$ IQR vom den Rändern der Box entfernt ist. Werte außerhalb dieser Antennen werden als Ausreißer bezeichnet und als einzelne Messpunkte eingetragen. Den Median umgibt eine Einbuchtung an der Box, welche das 95% Konfidenzintervall um den Median markiert[19].

3.2.2. Laufzeit

Die einfachste Metrik zur Bewertung eines Schedulers ist die Laufzeit. Sie wird durch den jeweiligen Testfall selbst gemessen. Die Laufzeit wird durch die Zeitdifferenz zwischen Beginn und Ende der Scheduler-Ausführung definiert. Das bedeutet, dass die Scheduler-Initialisierung nicht Teil der Laufzeit ist, da in dieser Arbeit primär der Scheduling-Algorithmus an sich betrachtet werden soll. Aus der Laufzeit und der Anzahl der ausgeführten Tasks ergibt sich der Durchsatz des Schedulers. Diese Metrik kann verwendet werden, um unterschiedliche Task-Anzahlen miteinander zu vergleichen.

3.2.3. Skalierung

Neben der Laufzeit muss auch die Skalierbarkeit betrachtet werden. Diese gibt an, wie gut das untersuchte Programm auf größere Probleme bzw. Ressourcen übertragbar ist. Beispielsweise würde ein perfekt skalierendes Programm mit den doppelten Ressourcen das gleiche Problem in der Hälfte der Zeit lösen. Um die Skalierbarkeit zu messen wird die so genannte Paralleleffizienz angegeben. Diese wird durch folgende Gleichung bestimmt:

$$\eta = \frac{t_1}{t_n \cdot n} \tag{3.6}$$

Dabei bezeichnet t_1 die Laufzeit des Programms unter Verwendung eines Rechenkerns und t_n die Laufzeit auf n Rechenkernen. Der Wert liegt typischerweise im Intervall $\eta \in [0, 1]$, jedoch ist es durch Nebeneffekte, wie größere Caches durch die Verwendung von mehr Rechenkernen, auch möglich eine Effizienz von über 100% zu erreichen. In diesem Fall spricht man von einer superlinaren Skalierung. Eine Paralleleffizienz von 100% bedeutet somit, dass eine Erhöhung der Ressourcen um den Faktor x die Rechenzeit ebenfalls um den Faktor x verkürzt. Je weiter η gegen Null läuft, desto weniger rentabel ist das Hinzufügen weiterer Ressourcen.

3.2.4. Cache-Performance

Der Linux Kernel stellt ein Profiler-Werkzeug namens `perf` zur Verfügung. Mit diesem ist möglich, ohne großen Overhead, die Werte verschieder Hardware Performance Counter zu überwachen. Hierfür wird das in `perf` integrierte `stat` Werkzeug verwendet[16]. Damit ist es möglich, sowohl die Gesamtzahl an Cache-Zugriffen, als auch die Anzahl an Cache Misses zu überwachen. Es wurde ein Python-Skript entworfen, welches die von `perf` generierten Messdaten automatisch weiterverarbeitet und visualisiert.

3.2.5. Scheduler-Instrumentierung

Für die Untersuchung von Scheduling-Details wurde das in der vorherigen Praxisphase geschaffene Werkzeug zur Instrumentierung von Schedulern an den Scheduling-Algorithmus angebunden[15]. Damit ist es möglich, detaillierte Statistiken über die Ausführung der einzelnen Tasks zu erhalten, unter Anderem die CPU-Auslastung, die Laufzeiten der Tasks sowie die Wartezeiten der Worker. Um dies zu ermöglichen, wurden einige der Scheduler-Funktionen mittels einer Tracing-API annotiert um die Task-Ausführung extern zu überwachen.

3.3. Benchmark-Architektur

Die Auswertung der Daten erfolgt getrennt von der Sammlung der Daten. Dies ermöglicht es die Auswertungsmethoden und die Darstellung der Daten im Nachhinein zu verändern ohne die Daten neu Aufnehmen zu müssen. Darüber hinaus ist es möglich auf Systemen, ohne die benötigten Python-Bibliotheken für die graphische Darstellung, Daten aufzunehmen. Um dies zu ermöglichen, werden die Daten der Messungen bei allen Auswertungen als Javascript Object Notation (JSON) Datei gespeichert und in einem zweiten Schritt von einem Analyseskript ausgewertet.

Um die Auswertungen so automatisiert wie möglich zu gestalten, wurde eine Python-Benchmark Umgebung implementiert. Diese bietet die Möglichkeit leicht einen Benchmark zu definieren. Jeder Benchmark besteht dabei aus zwei Schritten, der Datenerhebung und der Datenauswertung. Ersterer wird implementiert durch die Spezifikation eines Executors und eines Begrenzers. Der Executor enthält eine Vorlage wie die Kommandozeile bei jedem Aufruf aussehen soll. Zusätzlich werden die Anzahl an Skalierungsschritten sowie verschiedene Konfigurationen angegeben. Der Begrenzer spezifiziert ein Entscheidungskriterium, welches überprüft, ob noch weitere Messungen für die aktuelle Konfiguration durchgeführt werden müssen. Derzeit sind zwei verschiedene Begrenzer implementiert: Abbrechen nach einer fixen Anzahl von Messungen oder abbrechen nach dem Median-Konfidenzintervall gemäß Abschnitt 3.2.1.

```
1  class MyBenchmark(Benchmark):
2      def getCollector(self):
3          limit = FixedIterations(4)
4          template = ["::exe", "20", ":*iteration=200"]
5          executor = Executor(template, limit,
6                              step=2, maxIteration=20)
7
8          executor.addConfig("foo", {"exe": "../tests/foo"})
9          executor.addConfig("bar", {"exe": "../tests/bar"})
10         return executor
11
12     def plot(self, path):
13         with open(path, "r") as inp:
14             data = json.load(inp)
15         boxplot_2d(data["runtimes"])
```

Abbildung 3.3.: Beispiel-Benchmarkdefinition

In Abbildung 3.3 ist ein exemplarisches Benchmark, welches zwei Programme mit dem beispielhaften Parametersatz ausführt, dargestellt. In der Vorlage wird `::exe` durch das `exe` Attribut aus der Konfiguration ersetzt. Der Parameter `:*iteration=200` wird in jeder Skalierungsiteration mit der jeweiligen Iterationsnummer multipliziert. Da für jeden Skalierungsschritt vier Stichproben für jede Konfiguration erhoben werden, werden insgesamt

$$n = 4 \cdot 2 \cdot 10 = 80$$

Messungen durchgeführt.

In der `plot` Routine werden die Daten eingelesen und beliebig viele Auswertungen und Grafiken erzeugt. Im Beispiel oben wird beispielhaft ein zweidimensionales Boxplot-Diagramm (z.B. Abbildung 6.1) erzeugt.

Die einzelnen Benchmarks werden von einem Wrapper-Skript automatisch geladen und können über die Kommandozeile ausgeführt und ausgewertet werden. Obriges Beispiel kann anschließend mit folgender Kommandozeile ausgeführt und visualisiert werden:

```
./BenchmarkTool MyBenchmark collect
./BenchmarkTool MyBenchmark plot path/to/json/file
```

Der Vorteil an dieser Architektur ist, dass es sehr einfach ist, ein neues Benchmark anzulegen, da die meisten Code-Teile bereits existieren und einfach wiederverwendet werden können.

3.4. Testumgebung

Alle in dieser Arbeit durchgeführten Messungen wurden auf einer Workstation ausgeführt. Bei dieser handelt es sich um ein NUMA System mit zwei Sockeln, wobei je ein Sockel eine NUMA-Domäne bildet. In jedem Sockel befindet sich ein Intel Xeon E5-2650 v3 Prozessor. Jeder dieser Prozessoren verfügt über 10 Rechenkerne mit jeweils zwei Threads. Jedem Kern sind dabei zwei eigene Cache-Level zugeordnet: 32kB L1 sowie 256kB L2 Cache. Der letze Cache Level wird innerhalb aller Kerne eines Prozessor geteilt und umfasst 25MB. Das System verfügt über insgesamt 128GB Arbeitsspeicher, welcher gleichmäßig auf beide NUMA-Domänen verteilt ist.

4. Ist-Analyse des Schedulers

Das folgende Kapitel zeigt eine Ist-Analyse des aktuell in TRACE implementierten Task-Schedulers. Dabei soll zuerst der Ablauf einer typischen Scheduler-Ausführung beschrieben werden. Im Anschluss soll dieser in eine Kategorie der Scheduler-Literatur eingeordnet werden.

Nach der theoretischen Untersuchung wird die Performance des Schedulers erfasst, damit diese als Basis für spätere Modifikationen dienen kann.

4.1. Scheduling Algorithmus

4.1.1. Ablauf

In diesem Abschnitt wird der Ablauf einer typischen Scheduler Ausführung beschrieben. Nach einer kurzen High-Level Übersicht werden detailliert die wichtigsten Komponenten des Schedulings erläutert.

High-Level Überblick

Von einer hohen Abstraktionsebene aus betrachtet, werden vor der Ausführung des Schedulers alle Tasks und Abhängigkeiten spezifiziert. Anschließend beginnt der Scheduler den so definierten DAG zu optimieren, sofern die Optimierungen aktiviert sind. Danach wird für jeden zu verwendenden Prozessorkern ein Array angelegt und mit Zeigern auf alle Tasks gefüllt. Dieses wird nun für jeden Kern individuell sortiert. Nach dem Sortieren wird für jeden Kern ein Arbeiter-Thread gestartet, welcher die

Liste immer wieder nach verfügbaren Tasks absucht und diese ausführt oder an andere Worker delegiert. Diese Vorgehen wird so lange wiederholt bis der Scheduler von einem Thread beendet wird.

Taskgraphen

Eine Besonderheit des TRACE-Schedulers ist es, dass der Taskgraph möglicherweise kein DAG ist. Der Scheduler bietet die Möglichkeit den Taskgraphen automatisch zu wiederholen. Das bedeutet, dass die Tasks immer wieder hintereinander ausgeführt werden. Dafür wird zwischen zwei verschiedenen Abhängigkeiten unterschiedenen: Vorwärts und rückwärts. Bei einer Vorwärtsabhängigkeit von Task A zu Task B ist die Ausführung von Task B in Iteration n (B_n) abhängig von der Ausführung von A_n. Dies kann als $A \rightarrow B$ spezifiziert werden. Ist gleichzeitig A rückwärts von B abhängig $A \leftarrow B$ so kann A_n erst ausgeführt werden, wenn B_{n-1} ausgeführt wurde. Für den Fall $n < 0$ wird die Abhängigkeit als erfüllt betrachtet.

Taskgraph Optimierungen

Wie bereits erwähnt werden vor der eigentlichen Ausführung des TRACE-Schedulers Optimierungen auf dem Task-Graphen ausgeführt. Diese dienen dazu den Scheduling-Overhead zu reduzieren, indem Abhängigkeiten und Tasks zusammengefasst werden.

Dabei werden vor allem zwei Optimierungen durchgeführt:

1. Entfernen von nicht notwendigen Abhängigkeiten mithilfe einer transitiven Reduktion[26].
2. Zusammenfassen von aufeinanderfolgenden Tasks ohne weitere Abhängigkeiten.

Erstere führt eine transitive Reduktion durch. Dabei wird ein neuer Graph gesucht welcher mit der geringsten Anzahl an Kanten die gleiche transitive Hülle besitzt[26]. Das bedeutet, dass von jedem Knoten in beiden Graphen die gleichen anderen Knoten erreicht werden können, jedoch auf einem möglicherweise anderen Weg.

Die zweite Optimierung sucht Ketten von sequentiell aufeinanderfolgenden Tasks. Diese zeichnen sich dadurch aus, dass der Kindtask B ausschließlich vom Elterntask A abhängt ($A \rightarrow B$) und B der einzige Nachfolger von A ist[23]. Der Vorteil dieser Optimierung hier ist, dass der Scheduler nach der Ausführung von A nicht erst einen neuen Task suchen muss, sondern direkt B mit ausführt. Da B ausschließlich nach A ausgeführt werden kann, reduziert diese Optimierung nicht die Parallelität des Schedulers.

Task Datenstruktur

Jeder Task beinhaltet eine ganze Reihe an Attributen und Einstellungen. Diese sind jedoch nicht alle relevant für das Scheduling. Beispielsweise ist jedem Tasks ein Name zugeordnet, um das Debugging zu erleichtern. Wichtig ist, das jedem Task eine Kern-Affinität zugeordnet ist. Diese gibt entweder an, auf welchem Rechenkern der Task bevorzugt ausgeführt werden soll, oder definiert das dieser Task keine besondere Affinität besitzt und überall ausgeführt werden kann. Letzteres ist beispielsweise der Fall bei einfachen Tasks, die kaum auf Daten im Speicher zugreifen.

Darüber hinaus speichert jeder Task die Zahl der noch nicht erfüllten Abhängigkeiten und einen Verweis auf alle Eltern und Kindtasks. Wurde ein Task ausgeführt, so wird die Zahl der Abhängigkeiten aller Kindtasks atomar um eins reduziert. Zu Beginn einer Task-Ausführung wird der Zähler wieder auf die Anzahl der Elterntasks zurückgesetzt.

Eine wichtige Konfigurationseigenschaft von Tasks ist die Entscheidung ob ein bestimmter Task als Oversubscribed markiert ist. Unter Oversubscription versteht man den Zustand, dass mehrere Threads einem Kern zugeordnet sind. Dieser kann selbstverständlich immer nur einen dieser Threads gleichzeitig ausführen und muss daher zwischen ihnen wechseln. Ein Anwendungsfall in TRACE ist beispielsweise das Schreiben von Zwischenergebnissen in eine Datei.

Außerdem besitzt jeder Task eine atomare Flagge welche angibt, ob der Task gerade ausgeführt wird. Um zu prüfen ob ein Task ausgeführt werden kann, muss

geprüft werden, ob die Zahl der unerfüllten Abhängigkeiten gleich Null ist und die Ausführungsflagge nicht bereits gesetzt ist.

Arbeiter Threads

Ein Worker-Thread wird innerhalb des Schedulers als Pool Worker bezeichnet. Jeder Thread ist dabei einem Rechnerkern zugeordnet. Mittels der Open-Source `hwloc` Bibliothek[4] wird der Thread explizit auf diesen Kern gebunden solange er ausgeführt wird.

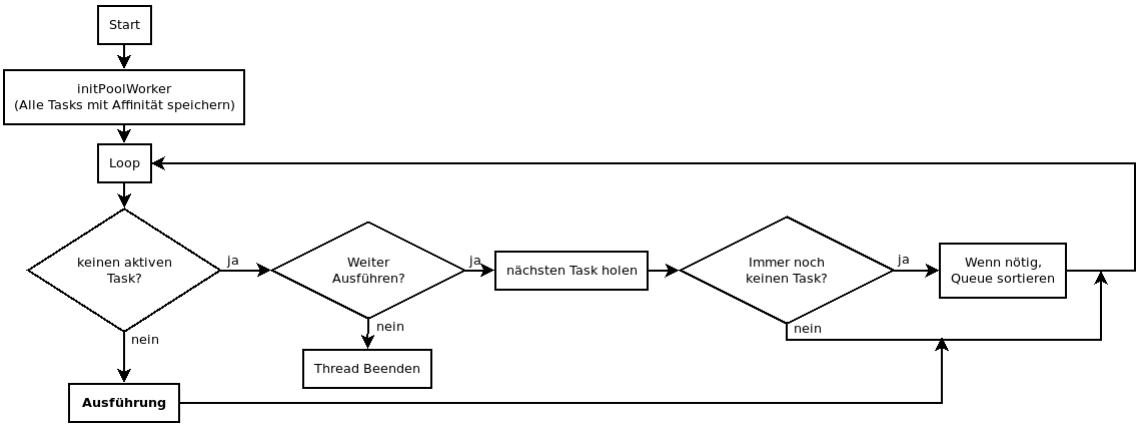


Abbildung 4.1.: Ablaufdiagramm eines Pool Workers

In Abbildung 4.1 ist die Hauptschleife eines Pool-Workers dargestellt: Zuerst wird eine Initialisierungsroutine aufgerufen. Diese prüft, ob dem Kern des Workers bereits ein Task-Array zugeordnet ist. Ist dies nicht der Fall, so werden Zeiger auf die Tasks in das Array eingefügt. Jeder Task enthällt dabei eine Markierung ob der Task für den aktuellen Worker affin ist. Ist er dies nicht, so wird er nicht von Workern auf diesem Kern ausgeführt. Die Affinität ist abhängig von der work stealing Konfiguration des Schedulers. Die implementierten Optionen sind `PROCESS`, `NUMA_DOMAIN`, `NONE`. Dabei werden entweder alle Tasks, Tasks der gleichen NUMA Domain oder nur Tasks des aktuellen Kerns als affin betrachtet.

Innerhalb der Hauptschleife (`Loop`) prüft der Worker zunächst, ob er einen Task zur Ausführung ausgewählt hat. Ist dies der Fall, so beginnt er mit dessen Ausführung. Besitzt er keinen Task, so prüft er zunächst ob der Scheduler beendet wurde.

Ist dies der Fall, so beendet sich der Thread. Ansonsten durchsucht er das Task Array nach einem zur Ausführung bereiten Task. Ist dies erfolgreich, so markiert er ihn für die nächste Iteration. Alternativ investiert er einen Teil seiner Laufzeit in die Neusortierung des Task-Arrays. Dies ermöglicht die Berücksichtigung von dynamischen Eigenschaften wie die Task-Laufzeit bei der Sortierung. Das Array wird jedoch nur sortiert, wenn dies seit der letzten Ausführung eines Tasks noch nicht geschehen ist. Die Kriterien für die Sortierung des Arrays werden in einem späteren Analyseabschnitt erläutert.

Task Ausführung

Wie bereits kurz im vorherigen Pool-Worker Abschnitt angerissen, ist die Ausführung eines Tasks durch den Pool-Worker komplexer als die bloße Ausführung der Taskfunktion.

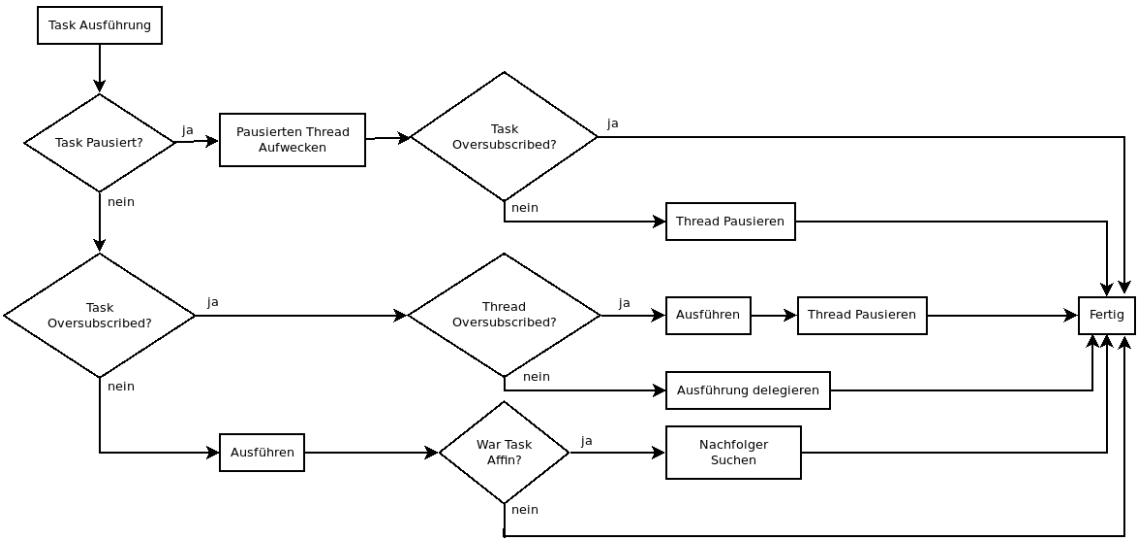


Abbildung 4.2.: Ablaufdiagramm der Task Ausführung

In Abbildung 4.2 ist der Ablauf der Ausführung eines Tasks dargestellt. Zunächst überprüft der ausführende Task, ob der Task bereits zum Teil ausgeführt wurde und die Ausführung pausiert hat, da er auf eine ausstehende Kommunikation wartet. In diesem Fall wird der pausierte Thread aufgeweckt. Der geweckte Thread prüft ob

seine Daten nun vorliegen. Ist dies nicht der Fall so pausiert er erneut. Der weckende Thread pausiert anschließend, sofern der aufgeweckte Task nicht oversubscribed ist. Ansonsten ist die Ausführung beendet und der Thread beginnt die nächste Iteration.

Wenn kein pausierter Thread dem Task zugeordnet ist, so wird überprüft ob der Task oversubscribed ist. Ist dies der Fall, so führt der Worker den Task nur aus, wenn er selbst auch oversubscribed ist. In diesem Fall pausiert der Thread nach erfolgter Ausführung wieder. Ist der aktive Thread nicht Oversubscribed, so delegiert er die Ausführung an einen pausierten Thread bzw. startet einen neuen Worker, sofern kein pausierter Thread verfügbar ist.

Ist der Task nicht oversubscribed, so wird er von dem Worker direkt ausgeführt. Handelt es sich um einen affinen Task, so wird nach der Ausführung geprüft ob ein geeigneter Kindtask zur Ausführung bereit ist. Ist dies der Fall so wird dieser als nächster Task festgelegt. Ansonsten startet der Worker Thread regulär in die nächste Iteration und beginnt mit der Suche nach einem neuen Task aus dem Task Array.

Task Sortierung

In vorherigen Abschnitten wurde bereits beschrieben, dass für jeden Rechenkern ein eigenes Array mit individueller Reihenfolge vorhanden ist. Dieses enthält lediglich Zeiger auf die global angelegten Tasks. Die aktuelle Implementierung sortiert die Tasks mittels einem mehrstufigen Ansatzes. Dabei werden der Reihe nach die einzelnen Vergleichsfunktionen angewandt, bis bei einem Vergleich eine Ordnung festgestellt werden kann. Dabei sind aktuell sechs verschiedene Vergleichsfunktionen implementiert:

Kritikalität Wenn einer der beiden Tasks als kritisch makiert ist, so wird er bevorzugt behandelt. Kritisch bedeutet, dass der Task auch dann noch ausgeführt wird, wenn der Scheduler eigentlich beendet wird.

Oversubscription Tasks, welche als oversubscribed markiert sind, werden vorrangig ausgeführt.

Synchronisationspunkte Jedes mal, wenn ein Task auf eine ausstehende Kommunikation wartet und dabei seinen Thread blockiert, wird ein Zähler für einen Synchronisationspunkt erhöht. Jeder Thread speichert die maximale Anzahl an Synchronisationspunkten pro Iteration. Tasks mit mehr Synchronisationspunkten werden bevorzugt ausgeführt.

NUMA Distanz Befindet sich ein Task auf der selben NUMA-Domäne wie der Rechenkern, so wird er bevorzugt behandelt, um die Anzahl an Remote-Speicherzugriffen möglichst gering zu halten.

Kerndistanz Tasks deren Rechenkern topologisch näher an dem aktuellen Rechenkern liegen, werden bevorzugt ausgeführt. Die Nähe wird hier über die Nähe im der Prozessorhierarchie definiert. Benachbarte Rechenkerne, verwenden oft mehr Cache-Level gemeinsam.

Laufzeit Jeder Task speichert die Anzahl der CPU-Zyklen welche mit seiner Ausführung bisher verwendet wurden. Tasks auf der gleichen NUMA-Domäne wie der aktuelle Rechenkern werden bevorzugt, wenn sie möglichst lang sind. Bei Tasks in anderen Domänen werden wiederum kurze Tasks bevorzugt.

Diese werden von oben nach unten auf die zu vergleichenden Tasks angewandt bis ein Unterschied gefunden wird. Als eigentlicher Sortieralgorithmus wird die Quicksort-Implementierung aus der C-Standardbibliothek verwendet.

4.1.2. Komplexität

In den vorherigen Abschnitten wurde der Ablauf des Schedulers beschrieben. Daraus lässt sich ableiten, dass es im wesentlichen drei verschiedene Operationen für den Scheduler gibt:

- Task zum Ausführen suchen
- Task in Warteschlange einfügen
- Taskliste sortieren

Da der aktuelle Scheduler die Liste von Tasks stets komplett durchsucht, handelt es sich hierbei um eine sequentielle Suche. Diese besitzt einen durchschnittlichen Zeitaufwand von $\mathcal{O}(\frac{n}{2})$[10], wobei n die Anzahl der Tasks des Schedulers darstellt.

Da bei der Auswahl des auszuführenden Tasks der Task nicht aus der Liste der Tasks entnommen wird, besteht keine Notwendigkeit den Task zur Laufzeit des Schedulers wieder in die Liste der Tasks einzufügen. Daher ist diese Operation nicht notwendig und benötigt daher auch keine Laufzeit.

Da für das Sortieren der Task Liste die Quicksort-Implementierung verwendet wird, kann hierfür ein durchschnittlicher Zeitaufwand von $\mathcal{O}(n \cdot \log(n))$ angenommen werden[10]. Da das Sortieren nur dann ausgeführt wird, wenn der Worker-Thread keine weitere Arbeit zum Ausführen besitzt, wird hier Idle-Zeit dafür genutzt, die Auswahl des nächsten Tasks zu optimieren.

Ein mögliches Skalierungsproblem, welches aus der Komplexität der Such- bzw. Sortieroperation resultiert, ist, dass jeder Worker stets alle Tasks betrachtet. Für den Fall, dass sehr viele Tasks im Scheduler vorhanden sind, könnte das Suchen nach Tasks und vor allem das Sortieren sehr aufwändig sein.

4.1.3. Theoretische Einordnung

Vergleicht man den Scheduler in TRACE mit den in [25] klassifizierten Schedulern, so stellt man fest, dass der Scheduler der Einzige ist, welcher einen Graphen als Task-Struktur verwendet, der kein DAG ist. Der Grund hierfür ist, dass durch die Abhängigkeiten aus vorherigen Iterationen eine Art Zyklus darstellen.

Die Art des Task-Schedulings ist etwas schwieriger zu definieren: Der TRACE Scheduler ähnelt einem Work-Stealing Scheduler. Indizien dafür sind, dass es vorrangig

kein vorausschauendes Scheduling gibt. Stattdessen suchen Threads, die keinen Task zum Ausführen besitzen, eigenständig nach neuer Arbeit.

Ein wichtiger Unterschied zum klassischen Work-Stealing nach Blumofe[2] ist, dass die einzelnen Threads keine Ready-Deques verwalten. Stattdessen durchsuchen sie immer wieder das gesammte Array von Tasks. Dies hat den Vorteil, dass keine Modifikationen des Arrays während der Laufzeit nötig sind[9]. Jedoch ist es nötig das Task-Array gegebenenfalls neu zu sortieren, um dynamische Eigenschaften (z.B. Task-Länge) beim Scheduling berücksichtigen zu können.

Dadurch, dass alle Threads stets alle Tasks in einer individuellen Reihenfolge durchsuchen, ist es möglich, dass Tasks von anderen Rechenkernen gestohlen werden, obwohl noch ausführbare Tasks für den aktiven Kern verfügbar wären. Um das klassische Work-Stealing Verhalten abzubilden, müsste das erste Sortierungsmethode die Tasks nach eigenen und gestohlenen Tasks sortieren. Wie in Abschnitt 4.1.1 beschrieben ist dies jedoch standardmäßig nicht der Fall.

4.2. Untersuchung des Laufzeitverhaltens

In diesem Abschnitt sollen verschiedene Laufzeiteigenschaften des TRACE-Schedulers untersucht werden. Diese umfassen, neben einer Laufzeitanalyse, auch einen detaillierten Blick auf die Mindestgröße für Tasks des Schedulers.

4.2.1. Thread-Auslastung

In Abschnitt 2.1.3 wurde bereits diskutiert, dass die Ressourcenauslastung durch einen Scheduler essentiell ist. Daher wurde für den TRACE-Scheduler die Auslastung der zur verfügung stehenden Threads untersucht.

In Abbildung 4.3 ist die Messung des Minimalbeispiels mittels des in Abschnitt 3.2.5 vorgestellten Analysewerkzeuges dargestellt. Es gibt an, wie viele Threads zu welchem Zeitpunkt mit welcher Art von Arbeit beschäftigt sind. Da die Parameter $\{400, 20\}$ gewählt wurden, sind im Idealfall konstant 20 Threads mit der Ausführung eines

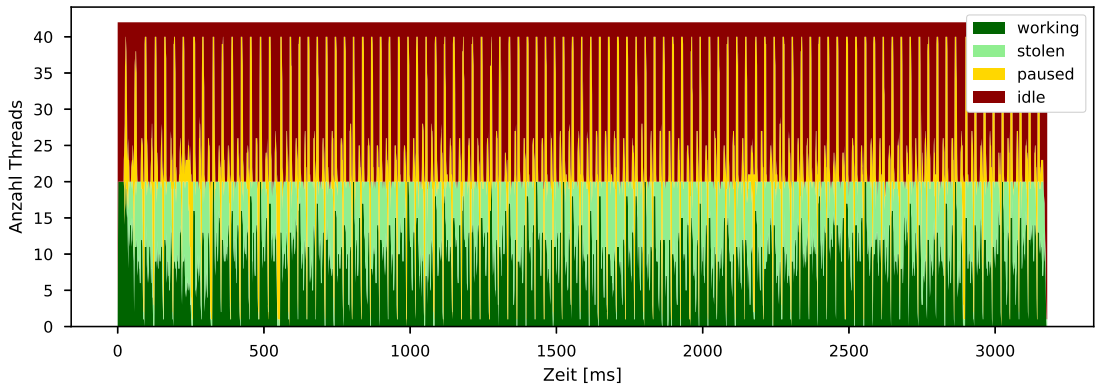


Abbildung 4.3.: Visualisierung der Scheduler-Ausführung

Tasks beschäftigt. Hier wird zwischen `working` und `stolen` unterschieden. Ersteres bezeichnet die Ausführung eines Tasks auf seinem bevorzugen Rechenkern sowie die Ausführung eines nicht-affinen Tasks. Letzteres bezeichnet wiederum die Ausführung eines Tasks ohne Affinität zum aktuellen Rechenkern, zum Beispiel von der gleichen NUMA-Domäne, wenn das entsprechende Task-Stealing aktiviert ist. Threads, die auf die Verfügbarkeit von Daten warten, bevor ihre Ausführung fortgesetzt werden kann, sind in gelb dargestellt.

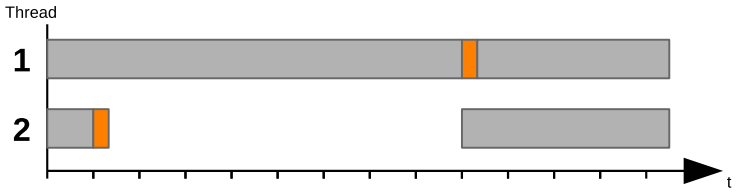
Betrachtet man die Ergebnisse für den Scheduler, so stellt man fest, dass zu beinahe keinem Zeitpunkt weniger als 20 Threads aktiv sind. Jedoch existieren eine Reihe von Spikes durch pausierte Threads. Bei näherer Betrachtung stellt man fest, dass die Anzahl der Pausen-Spikes der Anzahl der Iterationen des Lösers (100) entspricht. Daher existiert in jeder Iteration einen Phase, in der nahezu alle Threads blockieren und auf Daten warten. Die Ursache hierfür können vor allem zwei Dinge sein:

- Fehlende Parallelität innerhalb des Task-Graphen durch Daten-Abhängigkeiten
- Suboptimale Ausführungsreihenfolge der Tasks durch den Scheduler

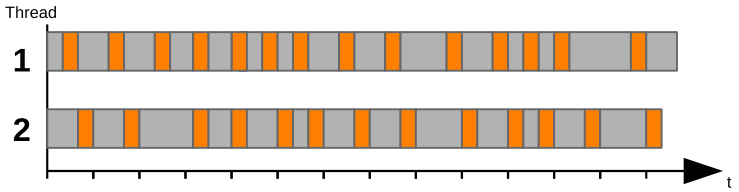
Um den zweiten Punkt näher bewerten zu können, muss die Task-Sortierung in dem Scheduler untersucht und gegebenenfalls verbessert werden. Dies wird in späteren Experimenten in Abschnitt 6.1.2 untersucht.

4.2.2. Optimale Task-Größe

Für die Performance eines Schedulers ist die Größe der einzelnen Tasks von essentieller Bedeutung. Sind die einzelnen Tasks zu groß, so kommt es leicht zu einem Ungleichgewicht zwischen den einzelnen Arbeiter-Threads[23], da oft nicht alle Tasks eine gleich lange Laufzeit haben: Während ein Thread beispielsweise einen sehr aufwändigen Task bearbeitet, müssen die anderen Threads gegebenenfalls sehr lange pausieren, wenn die anderen Tasks alle von diesem Task abhängen. Für die Lastbalancierung ist es daher vorteilhaft, wenn die Tasks feingranularer sind, da sie dynamischer verteilt werden können.



(a) Schlechte Lastverteilung durch zu große Tasks



(b) Sehr hoher Scheduler-Overhead durch zu kleine Tasks

Abbildung 4.4.: Performanceprobleme mit zu großen und zu kleinen Tasks

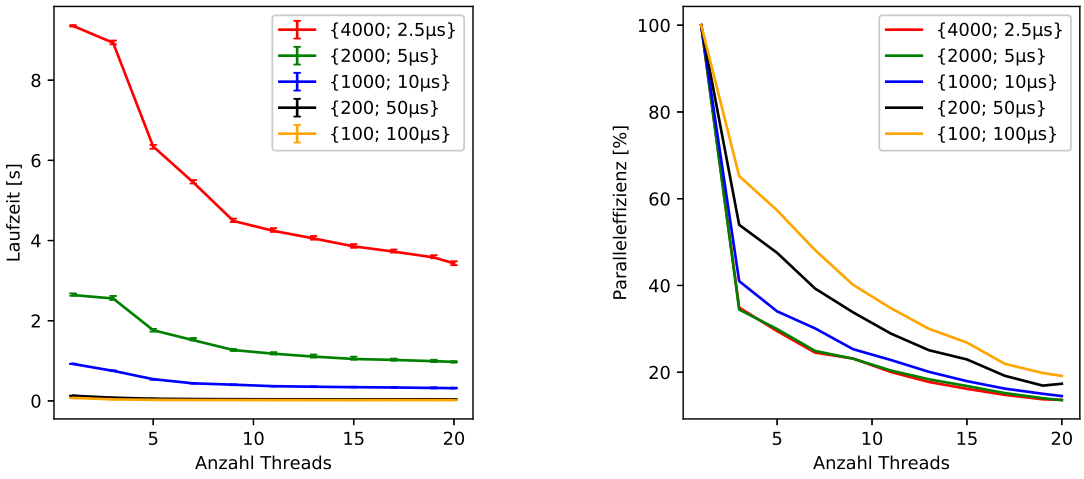
Auf der anderen Seite ist nach der Ausführung eines jeden Tasks ein Aufruf des Schedulers nötig. Ist die durchschnittliche Task-Länge zu kurz, so wird der Scheduler so oft ausgeführt, dass er einen signifikanten Anteil der Laufzeit einnimmt[23]. Als Beispiel soll angenommen werden, dass ein Scheduler 1μs benötigt, um einen ausführbaren Task auszuwählen. Sind die Tasks durchschnittlich 3μs lang, so kann im Optimalfall nur 75% der Laufzeit für die Ausführung von Tasks verwendet werden und die restlichen 25% werden im Scheduler verbraucht. In Abbildung 4.4 sind die beiden Möglichkeiten dargestellt. Dabei sind die Tasks in grau und die Scheduler-Ausführung in orange dargestellt. In Abbildung 4.4a sieht man, dass der zweite Thread

länger blockiert, da nicht genügend Parallelität vorhanden ist. In Abbildung 4.4b sind hingegen beide Threads voll ausgelastet, jedoch wird ein signifikanter Teil der Laufzeit für die Auswahl der nächsten Tasks verwendet, da stets nur sehr kurze Tasks ausgeführt werden.

Zu beachten ist jedoch, dass viele kleine Tasks leicht zu einem großen Task aggregiert werden können. Hierfür gibt es neben dem Zusammenfassen von mehreren aufeinanderfolgenden Tasks (siehe Abschnitt 4.1.1) noch einige weitere Möglichkeiten: Neben sequentiellen Tasks können auch mehrere parallele Tasks zusammengefasst werden, wenn diese die gleichen Abhängigkeiten haben[23]. Außerdem können auch ganze Gruppen von Tasks zu einem großen Task zusammengefasst werden. Dieser neue Task besitzt im Anschluss die Abhängigkeiten aller zusammengefassten Tasks[23]. Da es im Allgemeinen nicht immer möglich ist, große Tasks in kleinere zu zerlegen[23], ist es sinnvoll die Tasks so klein wie möglich zu halten. Das erlaubt es, die Task-Größe zur Laufzeit an das Optimum anzupassen.

Um die optimale Task-Größe für den TRACE Scheduler zu finden, wurde 2019 bereits eine Untersuchung des Fraunhofer-Instituts für Techno- und Wirtschaftsmathematik (ITWM) durchgeführt. Diese kam zu dem Ergebnis, dass die optimale Task-Größe zwischen 10μs und 100μs liegt[9]. Da dieses Ergebnis jedoch bereits über ein Jahr zurückliegt und der Scheduler in der Zwischenzeit weiterentwickelt wurde, soll überprüft werden, ob die angegebene Task-Granularität immer noch eine sinnvolle Empfehlung ist.

Als Testfall wurde der in Abschnitt 3.1.2 vorgestellte parallele Regionen Testfall verwendet. Dabei werden in jeder Ausführung 4 Ebenen von Tasks verwendet. Für die Ausführung des Programms werden zwei Parameter benötigt: Die Anzahl der Tasks pro Ebene und die Länge eines Tasks in Mikrosekunden. Als Basiswert werden 1000 Tasks mit je 10μs Länge verwendet. Um die Gesamtlaufzeit vergleichen zu können ist das Produkt aus Task-Anzahl und Task-Länge stets konstant. Die Ergebnisse sind in Abbildung 4.5 dargestellt. Dabei ist in Teil `(a)` die Gesamtlaufzeit der untersuchten Testfälle dargestellt. In `(b)` wiederum wird die Paralleleffizienz dargestellt. Betrachtet man die Laufzeit der Tasks, so erkennt man, dass die Variante mit 2.5μs langen Tasks mehr als die doppelte Laufzeit besitzt im Vergleich zu der Variante mit 5μs



(a) Laufzeit

(b) Parallelleffizienz

Abbildung 4.5.: Vergleich verschiedener Task-Größen für den Scheduler

Tasks. Der Laufzeitunterschied zwischen den anderen Testfällen ist absolut betrachte deutlich kleiner. Außerdem fällt auf, dass der Laufzeitunterschied zwischen 50μs und 100μs Tasks verschwindend gering ist. Der Abstand zu den nächst-kleineren Tasks (10μs) ist hingegen deutlich größer. Bei der Parallelleffizienz ist der Unterschied zwischen diesen beiden Tests noch deutlicher. Aus diesem Grund ist es möglicherweise sinnvoll, die in [9] vorgeschlagene Untergrenze von 10μs anzuheben, da größere Tasks nicht nur eine deutlich bessere Laufzeit aufweisen sondern darüber hinaus auch besser skalieren. Tasks sollten am unteren Ende mindestens 50μs Laufzeit umfassen, um eine gute Performance und Skalierbarkeit zu gewährleisten.

Um dieses Ergebnis weiter zu überprüfen, wurde eine Messreihe mit den verschiedenen Task-Größen und einem weiter vereinfachten parallele Regionen Testfall durchgeführt. Dieser enthält nun stets 500 Tasks, welche keinerlei Abhängigkeiten untereinander besitzen. Dadurch sollten beliebig viele Threads in der Lage sein, diese abzuarbeiten. Für den Versuch wurden jeweils 20 Threads verwendet. Die aufgenommenen Scheduling-Verläufe finden sich in Abbildung B.1. Man erkennt deutlich, dass bei einer Task-Größe von 10μs lediglich etwa 13% der Threads gleichzeitig ausgelastet sind. Bei steigender Task-Größe steigt somit auch die Anzahl der maximal nutzbaren Threads an. Für eine Task-Größe von 200μs werden bereits knapp ein Drittel

der verfügbaren Threads ausgelastet. Wichtig zu beachten ist, dass die Angaben zur prozentualen Auslastung (η) sich auf die durchschnittliche Auslastung beziehen und daher kurzzeitig mehr Threads verwendet werden. Es wird dennoch deutlich, dass die im vorherigen Absatz erwähnten 50μs nur als unterer Grenzwert dienen können, die Tasks, sofern möglich, jedoch deutlich größer sein sollten. Dies kann jedoch mittels Zusammenfassen von Tasks dynamisch geschehen um so die nötige Parallelität gewährleisten. Zu beachten ist jedoch die sehr kurze Ausführungszeit des Schedulers. Man erkennt, dass die Thread Auslastung gegen Ende ansteigt. Dies ist vermutlich durch eine Startup-Phase zu erklären, in welcher die einzelnen Worker gestartet werden und beginnen einen Task zur Ausführung zu finden. Vergleicht man die Ausführungszeiten der Tasks des Minimalbeispiels aus Abbildung 3.1 mit der empfohlenen Mindestlaufzeit, so stellt man fest, dass diese deutlich größer sind. Daher ist davon auszugehen, dass die Tasks des Strömungslösers im Allgemeinen groß genug sind, um den Scheduler effizient auszulasten. Ob die nötige Parallelität gewährleistet wird, kann erst nach der finalen Implementierung in TRACE realistisch beurteilt werden. Untersucht man jedoch den Task-Graphen des Minimalbeispiels für 20 Threads mittels der Gleichung aus (2.2), so erhält man durchschnittlich etwa 30 parallel verfügbare Tasks. Daher lässt sich folgern, dass zumindest das Minimalbeispiel sowohl die nötige Parallelität als auch die Mindest-Taskgröße erfüllt.

5. Untersuchte Modifikationen

In diesem Kapitel werden eine Reihe von Modifikationen am Scheduler vorgestellt. Dabei wurden insgesamt vier verschiedene Kategorien von Änderungen untersucht: Minimale Abänderungen an der bestehenden Implementierung, alternative Datenstrukturen für die Tasklisten der Worker, eine modifizierte Task-Reihenfolge, sowie Methoden für eine potentiell erhöhte Task-Lokalität.

5.1. Task Separation

Wie bereits in Abschnitt 4.1.1 beschrieben, könnte die Komplexität der Suche ($\mathcal{O}(\frac{n}{2})$) und der Sortierung ($\mathcal{O}(n \cdot \log(n))$) bei vielen Tasks zu einem Skalierungsproblem führen. Je nach eingestellter Work-Stealing Strategie gibt es einen Menge von Tasks, welche auf einem bestimmten Rechenkern nicht ausgeführt werden. Da diese Gruppe von Tasks gegebenenfalls trotzdem bei jeder Iteration durchsucht werden muss, reduziert sie die Skalierung des Schedulers.

Um dem Problem vorzubeugen, werden in einem ersten Schritt nur noch auf einem Kern ausführbare Tasks in das entsprechende Task-Array einsortiert. Dies führt dazu, dass in jedem dieser Arrays nur noch m_i Tasks enthalten sind. Dabei gilt für jedes m_i das $m_i \leq n$. Je mehr Tasks an einen bestimmten Kern gebunden sind, desto besser ist die erwartete Skalierung, da mehr Tasks ausgeschlossen werden und daher weniger Tasks in einer Task-Liste berücksichtigt werden müssen. Es ist jedoch zu beachten, dass eine strengere Zuordnung möglicherweise einen negativen Einfluss auf die Laufzeit hat und eine bessere Skalierung so dennoch zu einer höheren Laufzeit führen kann.

5.2. Datenstruktur

In Abschnitt 5.1 wurde eine Möglichkeit vorgestellt, den Zeitaufwand der Listenoperationen zu reduzieren, indem die Anzahl der Einträge reduziert werden. Eine Alternative ist das verwenden einer Run-Queue (Deque) wie bei klassischen Scheduling-Algorithmen. Das bedeutet, dass ein Task dessen Abhängigkeiten erfüllt wurden, in eine Deque eingefügt wird. Die einzelnen Worker durchsuchen nun nicht mehr die gesamte Gruppe an Tasks, sondern entnehmen lediglich das vorderste Element aus ihrer Deque. Damit das Work-Stealing weiterhin möglich ist, suchen Worker, welche in ihrer Deque keine Tasks haben, in den Deques anderer Worker nach Tasks zur Abarbeitung.

5.2.1. Beschreibung der Strukturen

Im nachfolgenden Abschnitt sollen alle untersuchten alternativen Datenstrukturen für das Task-Scheduling näher beschrieben werden.

Nicht-Blockierende Ready-Deque

Anstelle eines Arrays fester Größe verwendet diese Modifikation des Schedulers für jeden Rechenkern eine eigene Deque. Der Ablauf des Schedulings erfolgt ähnlich zum Randomized Work Stealing (RWS) nach Blumofe[2]. Das heißt, ein Worker ohne Arbeit in seiner lokalen Deque, versucht, aus einer anderen Deque einen Task zu stehlen. Jedoch gibt es gegenüber dem original RWS Algorithmus einige kleinere Änderungen, um eine höhere Lokalität zu gewährleisten.

Diese Änderungen sind in Abbildung 5.1 dargestellt. Es ist erkennbar, dass beide Methoden im Wesentlichen den Work-Stealing Algorithmus implementieren, jedoch wird ein neuer Task nicht in der eigenen Deque abgelegt, sondern, falls vorhanden, seine Affinität beachtet oder zufällig einsortiert. Beim Stehlen eines Tasks wird zusätzlich kein zufälliges Opfer gewählt, stattdessen wird die größte Deque verwendet. Dies hat zur Folge, dass die Wahrscheinlichkeit, eines erfolgreichen Stehlversuchs,

Task-Erzeugung

```
1 def spawnTask(sched, task):
2   if task.affinity:
3     i = task.affinity
4   else:
5     i = rand()%sched.nCores
6   sched.deques[i].push(task)
```

Task-Suche

```
1 def getTask(sched, core):
2   task = sched.deques[core].pop()
3   if not task:
4     d = sched.getLargestDeque(core)
5   task = d.steal()
6   return task
```

Abbildung 5.1.: Task-Erzeugung und Suche nach Task

deutlich größer ist und somit schneller ein Task zur Ausführung gefunden werden kann. Wichtig ist, dass `getLargestDeque` die größte Deque innerhalb der eigenen NUMA Domäne sucht, um NUMA-Lokalitätsprobleme zu vermeiden.

Um sicherzustellen, dass die gemessene Performance nicht durch eine ungeeignete Deque Implementierung negativ beeinflusst wird, wurden drei verschiedene Deques evaluiert: `std::deque`, `boost::deque`[1] sowie die eigene Implementierung. Damit die Deques in einer Umgebung mit mehreren Threads verwendet werden können, werden Zugriffe mittels eines Mutexes synchronisiert. Da es sich bei zwei der drei Deques um C++ Bibliotheken handelt, ist ein C-Wrapper erforderlich. Um dabei gewährleisten zu können, dass die Deques leicht ausgetauscht werden können, wurde mittels des `_Generic` Attributs in C11 ein Deque-Interface entwickelt. In Abbildung 5.2 ist beispielhaft die `push` Methode dargestellt. Dabei wird bei der Kompilierung des Quelltextes anhand des Datentypen des `deque` Parameters der Aufruf des Makros `deque_push` durch den zugeordneten Funktionsaufruf ersetzt. Damit muss innerhalb des Schedulers nur noch der Datentyp der Deques geändert werden, um eine andere Deque-Implementierung zu verwenden. Dies macht den Vergleich verschiedener Implementierung sehr einfach.

Die drei Deque-Implementierungen wurden im Hinblick auf ihre Performance verglichen. Dafür wurde sowohl die Task als auch die Thread-Skalierung überprüft. Die Ergebnisse sind in Abbildung 5.3 dargestellt. Für den Test wurde der parallel Regionen Testfall verwendet. Als Stichprobengrenze wurde ein 95% Median-Konfidenzintervall von 0.05s (Task), beziehungsweise 0.01s (Thread), verwendet. Bei

[1] `https://www.boost.org/doc/libs/1_74_0/doc/html/boost/container/deque.html`

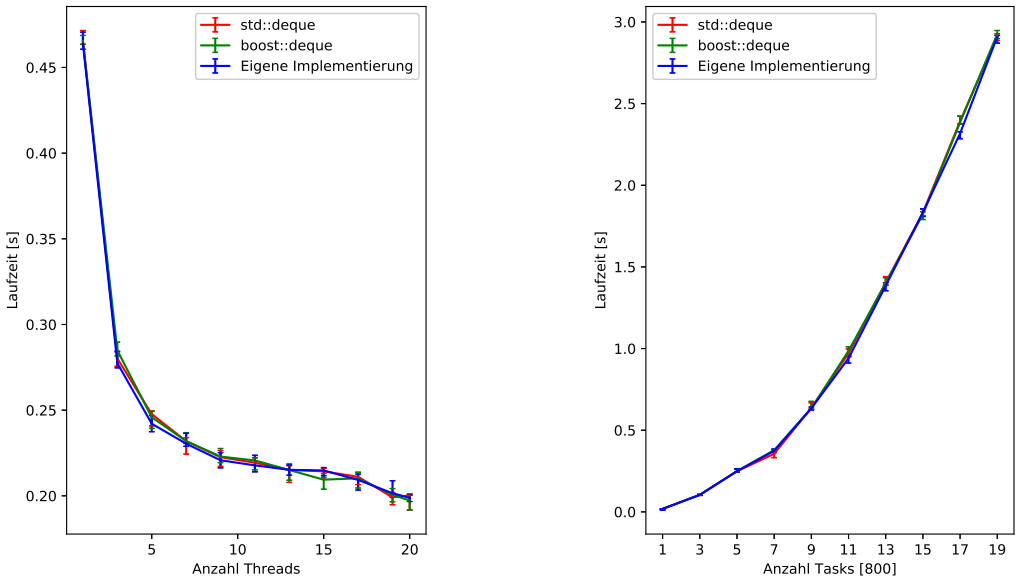
```
#define deque_push(deque, value) _Generic((deque), \
    StdDeque *: std_deque_push, \
    BoostDeque *: boost_deque_push, \
    TsDeque *: ts_deque_push, \
    default: _not_implemented)(deque, value)
```

Abbildung 5.2.: Generisches Deque Interface am Beispiel von `push`

der Thread-Skalierung wurden bei einer fixen Anzahl an Tasks in den Deques immer mehr Threads verwendet. Dadurch steigt die Konkurrenz um die Mutexe. Bei der Task-Skalierung wurde bei einer festen Anzahl an Threads die Anzahl der Tasks immer weiter erhöht. An beiden Graphen ist klar zu erkennen, dass es keinen signifikanten Unterschied zwischen der Performance der einzelnen Deques gibt. Zusätzlich wurde untersucht, ob eine höhere Konkurenz um den Mutex zu möglichen Performance Problemen führt. Hierfür wurde der $G(n, p)$-Testfall mit 2500 Tasks verwendet. Insgesammt wurden für jeden Wert von p insgesamt 30 Stichproben aufgenommen, jeweils mit einem unterschiedlichen Seed für jede Stichprobe. Je höher der Wert für p gewählt wird, desto mehr Abhängigkeiten existieren zwischen den Tasks. Dies führt dazu, dass weniger Tasks gleichzeitig verfügbar sind und die einzelnen Threads um weniger Tasks konkurrieren und daher häufiger unerfolgreiche Zugriffe auf die Task-Deques durchführen. Die Ergebnisse sind in Abbildung 5.3c dargestellt. Erkennbar ist, dass alle Deques nahezu identisch auf eine erhöhte Konkurrenz reagieren. Aus diesem Grund wird für die weiteren Untersuchungen die etablierte Implementierung der C++-Standardbibliothek (`std::deque`) verwendet.

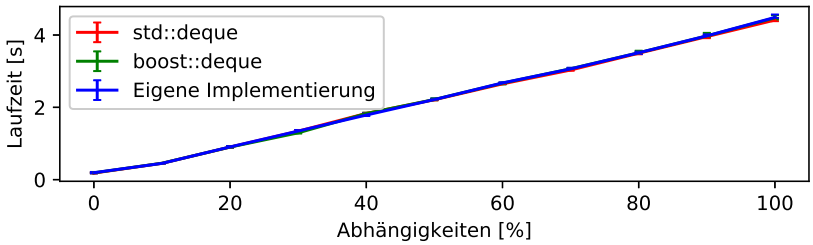
Blockierende Ready-Deque

Im vorherigen Abschnitt ist eine Deque beschrieben, welche als Alternative zu dem Task-Arrays des Schedulers dient. Da die einzelnen Deques jeweils mit einem Mutex geschützt sind, könnte es bei hoher Lastimbalanz dazu kommen, dass nur einzelne Threads einen Task ausführen, während die anderen durchgehend versuchen aus den Deques Tasks zu stehlen. Dabei könnte es möglicherweise zu starker Konkurrenz um den Mutex kommen, welche die Task-Ausführung verzögert, da Threads mit verfügbaren Tasks gegebenenfalls abwarten müssen, bis sie auf ihre eigene Deque



(a) Thread-Skalierung

(b) Task-Skalierung



(c) Task-Parallelität

Abbildung 5.3.: Vergleich verschiedener Deque-Implementierungen

zugreifen können, da sie von anderen Workern blockiert ist. Aus diesem Grund wurde eine blockierende (synchrone) Deque evaluiert. Hier prüft ein Worker-Thread ein einziges Mal, ob er einen Task von einer anderen Deque stehlen kann. Ist dies nicht der Fall, so pausiert er bis ein neuer Task in seiner Deque eingefügt wird, oder der Scheduler beendet wird.

Zwar führt dieses Verfahren zu einer niedrigeren Konkurrenz um die Mutexe der einzelnen Deques, dafür ist es möglich das mehrere Threads in ihrem blockierten Zustand verweilen, obwohl sie inzwischen Tasks von anderen Threads stehlen könnten. Es muss also experimentell geprüft werden, ob die möglicherweise reduzierte

Konkurrenz die potentiell weniger parallel abgearbeiteten Aufgaben rechtfertigt.

Globale Task-Queue

Eine weitere Modifikation ist es, statt einer Deque pro Rechenkern, eine globale Queue für alle Worker zu verwenden. Dies hat den Vorteil, dass es unerheblich ist, wo neue Tasks einsortiert werden, da alle Worker die gleiche Queue durchsuchen. Um die NUMA Affinität weiterhin gewährleisten zu können, fügt ein Worker einen entnommenen Task wieder hinten an die Deque an, wenn dieser nicht affin für ihn ist.

Für diese Implementierung wurde die vorher vorgestellte Deque Implementierung verwendet, da sich die benötigten Operationen nicht von einer normalen Queue unterscheiden. Um das Verhalten einer Queue nachzuahmen stehlen alle Worker aus dieser Deque, entnehmen also nicht am dem Ende, an dem sie die Tasks einfügen.

Ein potentielles Problem dieses Ansatzes ist es, dass durch den einzelnen globalen Mutex eine hohe Konkurenz um diesen entsteht, der, ähnlich wie in dem Fall der nicht-blockierenden Deques, die Laufzeit deutlich negativ beeinflussen könnte.

Multidimensionaler Heap

Jeder Tasks ist für die Ausführung auf einem bestimmten Rechenkern unterschiedlich gut geeignet. Diese Eignung soll in Form einer Priorität ausgedrückt werden. Jeder Eintrag erhält so k Prioritäten, wobei k den Index eines Rechenkerns darstellt. Die einzelnen Tasks werden so in einem k-D Heap, wie in Abschnitt 2.1.2 beschrieben, angeordnet.

Da die Anzahl der verwendeten Rechenkerne in TRACE relativ groß werden kann, muss hier die für große Schlüsselanzahlen optimierte Variante des k-D Heaps verwendet werden. Ein Thread, der einen Task zum Ausführen sucht, entfernt nun einfach den Knoten mit der höchsten Priorität für seinen Rechenkern. Damit ist garantiert, dass jeder Thread immer einen Task erhält, wenn geeignete Tasks im Heap verfügbar sind. Um zu verhindern, dass ein Thread einen Task auswählt, welcher nicht auf

diesem Rechenkern ausgeführt werden kann, wurde der Heap um eine `pop_cond` Methode erweitert, welches das größte Element über einem Prioritätsgrenzwert $p_k > 0$ aus dem Heap entfernt.

Einfache Sortierung

```
1  def getTaskPrio(task, core):
2    if task.affinity == core
3      || task.affintiy == None:
4      return 2
5    elif task.isAffine(core):
6      return 1
7
8    return 0
```

Klassische Sortierung

```
1  def getTaskPrio(task, core):
2    arr = getAllTasks()
3    sortTasks(arr, core)
4
5    pos = len(arr)
6    pos -= arr.index(task)
7
8    return pos
```

Abbildung 5.4.: Pseudocode für zwei Heap-Prioritätsfunktionen

Um die Tasks im Heap zu priorisieren, muss eine Funktion definiert werden, welche jedem Task je eine Priorität pro Rechenkern zuordnet. Hierfür wurden zwei verschiedene Methoden evaluiert. Diese sind in Abbildung 5.4 dargestellt. Für die klassische Sortierung wird zuerst ein Array aus allen Tasks erstellt, welches dann mit der bereits existierenden Sortierfunktion sortiert wird. Je weiter vorne ein Task anschließend in dem Array steht, desto höher ist seine Priorität. Selbstverständlich ist die Implementierung effizienter möglich als in Abbildung 5.4 dargestellt, indem in einem Funktionsaufruf die Prioritäten für alle Tasks bestimmt werden, da so nur ein Array angelegt und sortiert werden muss. Da die bestehende Sortierung auch dynamische Laufzeiteigenschaften berücksichtigt, muss diese Sortierung gegebenenfalls mehrfach zur Laufzeit neu berechnet werden. Der vereinfachte Alternativansatz ordnet jedem Tasks nur genau eine von drei möglichen Prioritäten zu, je nachdem ob der Tasks explizit für diesen oder alle Rechenkerne markiert wurde oder er unter der aktuellen Work-Stealing Konfiguration affin ist.

5.2.2. Komplexitäten

In Tabelle 5.1 sind die durchschnittlichen Laufzeitkomplexitäten der untersuchten Datenstrukturen aufgelistet. Es muss beachtet werden, dass das n für die Gesamtanzahl der Tasks im Scheduler und l für die Anzahl der aktuell lauffähigen Tasks steht.

Datenstruktur	Mutex	Task Suchen	Neuer Task	Tasks Sortieren
Array	Nein	$\mathcal{O}(\frac{n}{2})$	–	$\mathcal{O}(n \log(n))$
Deque	Ja	$\mathcal{O}(1)$ / $\mathcal{O}(1)$	$\mathcal{O}(1)$	–
Queue	Ja	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–
Heap	Ja	$\mathcal{O}(k^2 \log(l))$	$\mathcal{O}(k \log(l))$	–

Tabelle 5.1.: Durchschnittliche Laufzeitkomplexitäten der verwendeten Datenstrukturen

Zunächst fällt auf, dass die Referenzversion mittels des konstanten Arrays die Einzige ist, die ohne einen Mutex für die Datenstruktur auskommt. Dafür setzt sie voraus, dass die einzelnen Tasks eine atomare Flagge besitzen, um für alle Worker zu kennzeichnen, ob dieser Task bereits bearbeitet wird. Bei der Deque sind zwei Komplexitäten für das Entnehmen des ersten, bzw. letzten Elements, angegeben.

Wichtig zu beachten ist aber weiterhin, dass zwar das Erzeugen und Nehmen eines Tasks in einer Queue sehr effizient ist, jedoch nicht gesteuert werden kann welcher Task ausgewählt wird. Daher ist es wahrscheinlich, dass ein weniger sinnvoller Task gewählt wird als bei Datenstrukturen die eine Sortierung zulassen (Array und Heap). Welche Variante im Kontext von TRACE die bessere Performance liefert muss durch experimentelle Daten gezeigt werden.

5.3. Task-Reihenfolge

Eine Möglichkeit den TRACE Scheduler zu optimieren ist die Änderung der Vorgaben zur Sortierung der Tasks innerhalb der einzelnen Task-Arrays der Worker-Threads. In Abschnitt 4.1.1 wurde bereits die aktuelle Sortierung vorgestellt. In einem ersten Schritt wurde eine „Sortierung“implementiert, welche die Tasks zufällig anordnet. Damit soll der Einfluss der Task-Sortierung auf die Scheduling-Performance gezeigt werden.

5.3.1. Abhängigkeiten

Um die Parallelität des Task-Graphen möglichst effizient ausnutzen zu können, müssen möglichst viele Tasks gleichzeitig ausführbar sein. Dafür ist es sinnvoll, die Tasks auszuführen, welche im Anschluss möglichst viele neue Tasks generieren. Aus diesem Grund wurden zwei verschiedene Vergleichskriterien untersucht:

Kind-Abhängigkeiten Für jeden Task wird die Anzahl der Kindtasks bestimmt, die nur noch von einem Task abhängen. Da diese zur Ausführung bereit sind, sobald der Task ausgeführt wurde, werden Tasks bevorzugt einsortiert, wenn sie möglichst viele solcher Kindtasks haben.

Eltern-Abhängigkeiten Es werden die Tasks bevorzugt, welche möglichst wenig Eltern-Tasks besitzen. Dadurch, dass Tasks mit weniger Abhängigkeiten möglicherweise früher ausführbereit sind, ist die Wahrscheinlichkeit höher, das der Scheduler beim durchsuchen seines Task-Arrays früher einen Task findet.

5.3.2. Vorheriger Task

Um die Performance eines Worker zu erhöhen, sollten Tasks hintereinander ausgeführt werden, welche auf die gleichen Daten zugreifen, da dies die Cache-Performance erhöht[5]. Geht man davon aus, dass eine Abhängigkeit zwischen zwei Tasks bedeutet, dass die Tasks mit höherer Wahrscheinlichkeit auf die gleichen Daten zugreifen, so sind nach der Ausführung eines Tasks seine Kindtasks als nächste Tasks zu bevorzugen.

Da dies für affine Tasks bereits im TRACE Scheduler implementiert ist, musste lediglich eine `if`-Verzweigung entfernt werden, damit nach der Ausführung von allen Tasks die Kindtasks bevorzugt werden.

5.4. Task-Lokalität

In Abschnitt 5.3.2 wurde bereits eine Möglichkeit gezeigt, die Worker-Performance durch eine potentiell gesteigerte Cache-Performance zu erhöhen. Eine weitere Möglichkeit die Ausführzeit der Tasks zu reduzieren ist die Steigerung der NUMA-Lokalität. Hier sollte die Ausführung eines Tasks über mehrere Iterationen hinweg möglichst auf der gleichen NUMA-Domäne ausgeführt werden. Um dies gewährleisten zu können, wurden zwei mögliche Strategien untersucht, welche nachfolgend erläutert werden.

5.4.1. Feste Zuordnung

Wie bereits in Abschnitt 4.1.1 beschrieben ist es möglich, dass ein Task keine Affinität besitzt. Um die Lokalität zu maximieren, soll für dieses Experiment jeder Task eine feste Affinität erhalten. Daher wird vor der Ausführung des Schedulers jedem Task ohne Affinität eine Affinität zugewiesen. Diese entspricht der, welche für die meisten der Eltern-Tasks zutreffend ist. Ist dies nicht eindeutig, so wird eine zufällige Affinität ausgewählt.

Der mögliche Nachteil ist, dass die Tasks zwar nun auf einer NUMA-Domäne bleiben, dadurch jedoch die nutzbare Parallelität sinkt. Es muss daher experimentell untersucht werden, ob die Vorteile dieser Methode überwiegen oder ob sie die Laufzeit negativ beeinflusst.

5.4.2. Task Migration

Um die Lokalität für zukünftige Iterationen zu erhöhen, soll versucht werden, einen gestohlenen Task nach der Ausführung zu dem entsprechenden Rechenkern zu migrieren. Das bedeutet, dass der Task in Zukunft nicht mehr affin zu dem Rechenkern von dem er gestohlen wurde ist, sondern stattdessen eine neue Affinität erhält. Darüber hinaus wurde auch eine vererbende Migrationsstrategie implementiert, in welcher die Kinder eines gestohlenen Tasks ebenfalls zu dem neuen Rechenkern migriert werden

Diese Vorgehen ist durch die CAB-Scheduling Strategie[5] inspiriert. Diese zeichnet sich dadurch aus, dass zu Beginn der Ausführung der Task-Graph partitioniert wird. Dabei werden, ab einer bestimmten Tiefe, alle Tasks die auf einen Knoten folgen auf den gleichen CPU-Sockel ausgeführt[5].

6. Ergebnisse und Diskussion

In diesem Kaptiel werden zunächst die Ergebnisse der untersuchten Optimierungen aus dem vorherigen Kaptiel dargestellt. Anschließend werden die Ergebnisse unter Bezug auf die Ist-Analyse aus Kapitel 4 zusammengefasst und beurteilt.

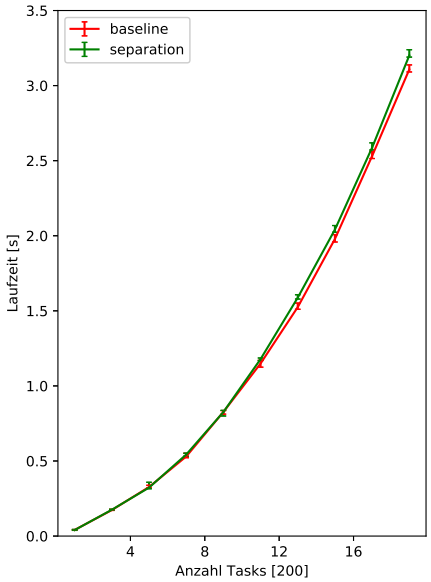
6.1. Benchmark-Ergebnisse

6.1.1. Task-Separation

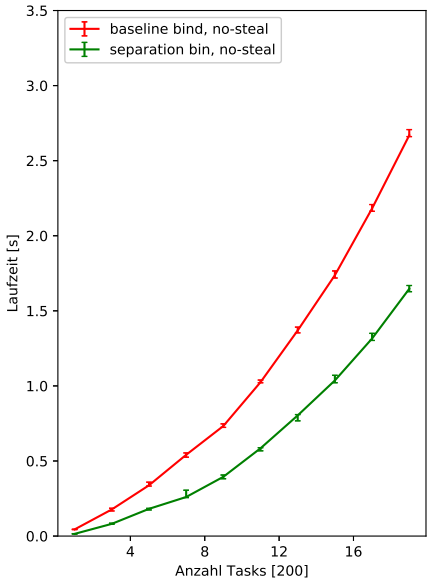
In Abbildung 6.1 ist ein Vergleich der Laufzeiten mit und ohne der in Abschnitt 5.1 vorgestellten Modifikation dargestellt. Für den Vergleich wurde der Testfall mit parallelen Regionen verwendet. Hierbei wurde stets der Parametersatz $\{4, 200n\}$ genutzt, wobei $n \in [1, 19]$ die wachsende Problemgröße darstellt. Alle Messungen wurde auf einer Workstation unter Verwendung aller 20 Rechenkerne durchgeführt. Für die Messungen wurde ein Konfidenzintervall von 0.05s verwendet.

Im ersten Test (Abbildung 6.1 (a)) ist ein Testfall ohne affine Tasks dargestellt. Man erkennt, dass die Laufzeit der Variante mit Task-Seaparation (grün) entweder auf einem gleichen Level oder sogar höher ist. Dies ist dadurch zu erklären, dass ohne Affinitäten die Zahl m, also die Anzahl Tasks in einer Task-Liste für einen Rechenkern, identisch mit der Gesamtzahl an Tasks (n) ist. Dadurch bietet die Optimierung keinerlei Vorteil, fügt aber einen extra Overhead hinzu.

In Abbildung 6.1 (b) ist wiederum eine gegenteilige Situation abgebildet: Jeder Task besitzt eine Affinität zu einem Kern. Jedem Kern ist dabei die gleiche Anzahl an Tasks zugeordnet. Zusätzlich wurde das Work-Stealing deaktiviert. Dies ist der



(a) Ohne Affinitäten, mit Work-Stealing



(b) Mit Affinitäten, Ohne Work-Stealing

Abbildung 6.1.: Vergleich der Laufzeiten mit und ohne Task-Separation

Optimalfall für die angesprochene Optimierung, da jeder Worker nun nur noch $\frac{1}{p}$ Tasks beachten muss, wobei p die Anzahl der verfügbaren Prozessoren darstellt. Hierbei zeigt sich eine deutlich höhere Performance im Vergleich zur Referenzversion. Diese nähert sich je nach Problemgröße sogar einem Faktor zwei an.

Damit diese Ergebnisse auf den realen Anwendungsfall in TRACE übertragen werden können, ist entscheidend wie groß der Anteil der Tasks ist, die nur auf einem bestimmten Kern ausgeführt werden dürfen. Außerdem ist wichtig zu beachten, dass während der Ausführung des Lösers Work-Stealing innerhalb einer NUMA Domäne aktiviert ist. Das bedeutet, dass die Anzahl der Tasks in einem Array für einen Rechenkern nie kleiner werden kann als $\frac{n}{d}$, wobei d die Anzahl der NUMA-Domänen ist. Im Beispiel der verwendeten Workstation sind dies zwei. Dadurch ist der Einfluss dieser Optimierung auf einer Workstation mit TRACE deutlich begrenzter, als mit dem synthetischen Beispiel in Abbildung 6.1 (b).

In Abbildung 6.2 ist der Einfluss der Task-Separation auf die Laufzeit des TRACE Minimalbeispiels dargestellt. Dabei wurde eine Problemgröße von 400 unter Ver-

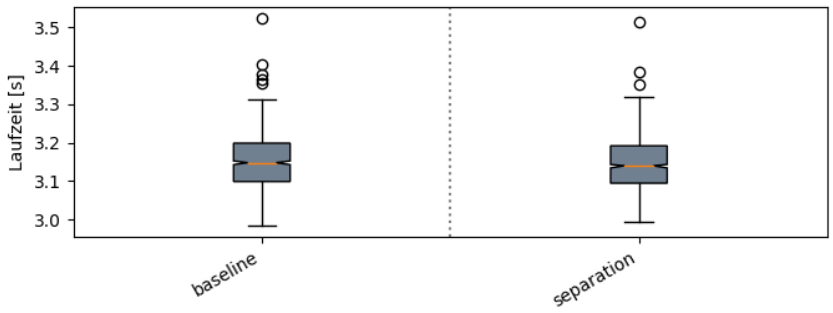


Abbildung 6.2.: Laufzeiten des Minimalbeispiels mit und ohne Task-Separation

wendung aller 20 Rechenkerne betrachtet. Es fällt auf, dass der Performancegewinn verschwindend gering ist. Zwar ist die modifizierte Variante minimal schneller als die Basisversion ohne Task-Separation, jedoch liegt der Unterschied der mittleren Laufzeit unter einem Prozent. Durch den geringen Unterschied zwischen den beiden Varianten wurde das Konfidenzintervall auf 0.01s beschränkt. Dennoch sind die beiden Konfidenzintervalle nicht klar trennbar, weshalb hier nicht von einem signifikanten Unterschied gesprochen werden kann.

6.1.2. Task-Sortierung

Um den Einfluss der Task-Sortierung auf die Scheduler-Performance zu untersuchen, wurden zuerst die einzelnen Task-Sortierungsstrategien aus dem Scheduler und den in Abschnitt 5.3 vorgestellten Konzepten einzeln verwendet. Die Ergebnisse dieses Experimentes finden sich in Abbildung 6.3. Die Untersuchungen wurden mit einem Konfidenzintervall von 0.05s durchgeführt. Dabei wurde das TRACE Minimalbeispiel mit den Parametern $\{400, 20\}$ verwendet.

Zuerst kann festgestellt werden, dass die Task-Reihenfolge einen großen Einfluss auf die Gesamtlaufzeit hat. Im schlechtesten untersuchten Fall, der Sortierung nach kritischen und nicht kritischen Tasks, liegt die mittlere Ausführzeit bei etwa 3.6s während im besten Fall, der Kern-Distanz der Tasks, die Laufzeit bei etwa 3.2s liegt. Das entspricht einer Laufzeitverbesserung von etwa 11%.

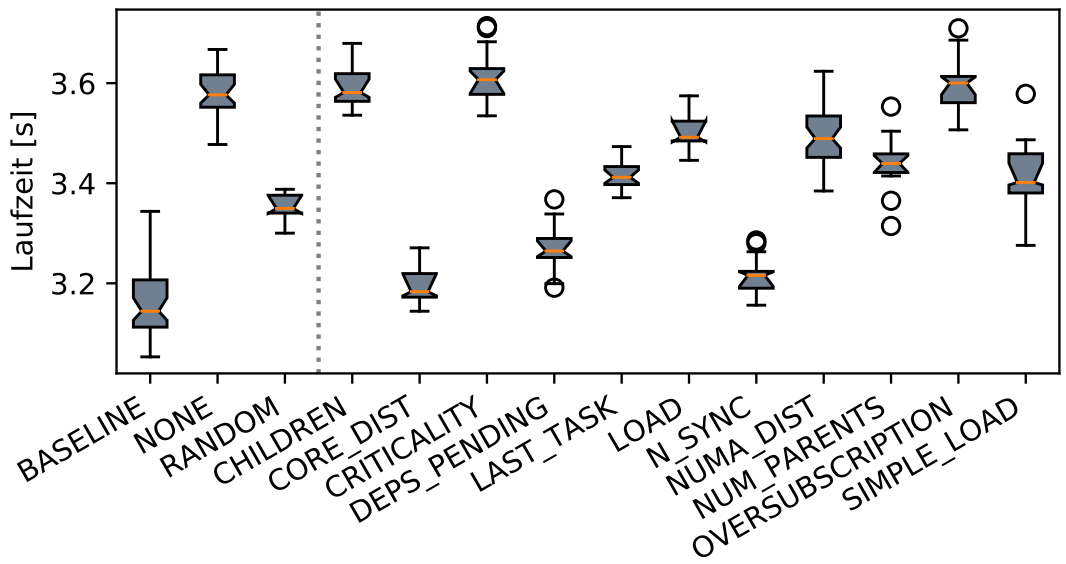


Abbildung 6.3.: Laufzeiten mit verschiedenen Task-Sortierungen

Außerdem fällt auf, dass die Performance der Referenzversion höher ist als alle einzelnen Sortierungskriterien. Daraus lässt sich folgern, dass die optimale Sortierung nicht ausschließlich durch ein einzelnes Kriterium bestimmen lässt und gegebenenfalls mehrere Kriterien verbunden werden müssen. Dabei können manche dieser Kriterien alleine eventuell einen negativen Effekt auf die Laufzeit haben. Da es jedoch aus Zeitgründen nicht sinnvoll möglich ist, alle Kombinationen zu bewerten, wurde nur eine händisch ausgewählte Untermenge untersucht. Im Wesentlichen lassen diese sich auf die folgenden vier Sortierungsstrategien reduzieren:

T1 OVERSUBSCRIPTION, CRITICALITY, CORE_DIST, N_SYNC, NUM_PARENTS

T2 OVERSUBSCRIPTION, CRITICALITY, CORE_DIST, N_SYNC, NUM_PARENTS, CHILDREN

T3 OVERSUBSCRIPTION, CRITICALITY, N_SYNC, CORE_DIST

T4 CHILDREN, NUM_PARENTS, OVERSUBSCRIPTION, CRITICALITY, N_SYNC, NUMA_DIST, CORE_DIST, LOAD

Bei der letzten Variante (`T4`) handelt es sich um eine leichte Modifikation der bestehenden Sortierungsstrategie. Lediglich die Anzahl der Kind- und Elterntasks wird zusätzlich betrachtet.

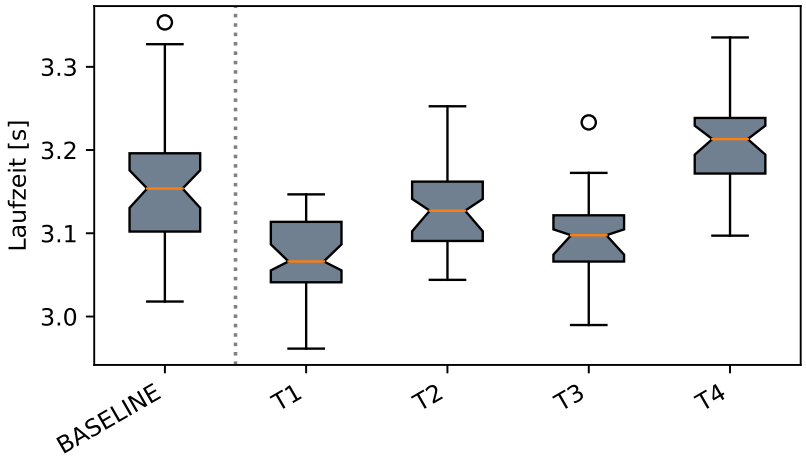


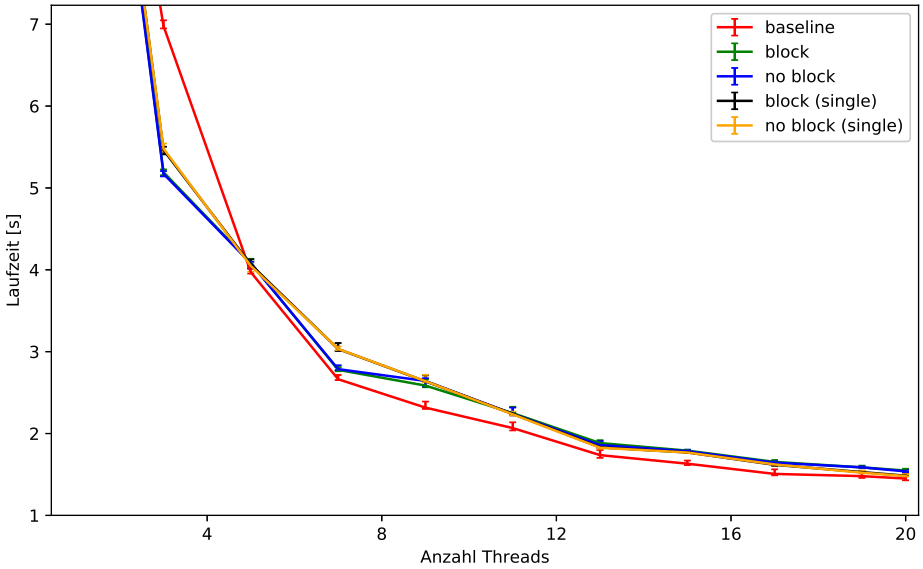
Abbildung 6.4.: Ergebnisse kombinierter Task-Sortierungen

Die Ergebnisse dieser Untersuchung sind in Abbildung 6.4 dargestellt. Auffällig ist, dass zwei der untersuchten Testfälle eine höhere Performance aufweisen als die Referenzversion. Für den Fall `T2` ist zwar eine geringere mittlere Laufzeit gemessen worden, jedoch überlappen die Konfidenzintervalle mit der Reverenzversion, weshalb hier keine signifikante Verbesserung vorliegt. Auffällig ist weiterhin, dass die Sortierung nach Anzahl der Kindtasks in jedem Fall einen negativen Effekt auf die Laufzeit hat. Hierbei ist vor allem der Unterschied zwischen `T1` und `T2` zu betrachten. Final lässt sich entweder `T1` oder `T3` als effizienteste untersuchte Sortierstrategie festhalten.

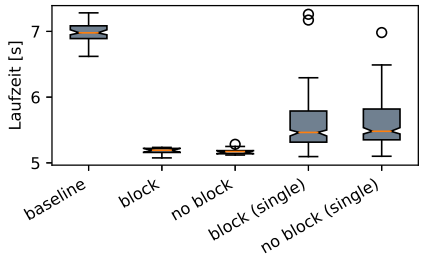
6.1.3. Ready-Deques

Um die möglichen Deque-Modifikationen bewerten zu können wurde eine konfigurierbare Implementierung geschaffen, welche wahlweise blockierende oder nichtblockierende Zugriffe auf entweder eine globale Deque oder eine Reihe von Kern-Deques ausführt. Als Testfall wurde erneut das TRACE-Minimalbeispiel verwendet.

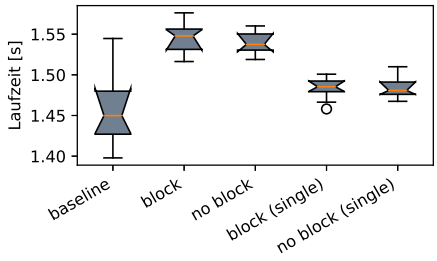
Es wurde versucht, das Konfidenzintervall auf 0.1s zu beschränken. War dies auch nach 1000 Iterationen nicht möglich, so wurde die Messung frühzeitig beendet.



(a) Übersicht über alle Thread-Anzahlen



(b) 3 Threads



(c) 20 Threads

Abbildung 6.5.: Laufzeitmessungen für Verschiedene Ready-Deques im Vergleich

Die Ergebnisse dieser Laufzeituntersuchung sind in Abbildung 6.5 dargestellt. In Teil `(a)` ist eine Übersicht über die Performance mit verschiedenen Thread-Anzahlen dargestellt. Dabei bezeichnet der Zusatz `(Single)` die Verwendung von einer globalen Task-Deque. Zuerst kann festgestellt werden, dass die Verwendung von blockierenden Zugriffen einen relativ geringen Einfluss auf die Performance des Lösers besitzt. Der Fakt, dass alle Threads um einen einzelnen globalen Mutex konkurrieren scheint in dieser Implementierung kein größeres Problem darzustellen.

Weiter fällt auf, dass es deutliche Unterschiede zwischen den verschiedenen Anzahlen verwendeter Threads gibt. Wie in Abbildung 6.5b dargestellt, sind alle Deque-Varianten für wenige Threads deutlich effizienter als die Referenzimplementierung. Dies ändert sich jedoch bereits bei mehr als 5 Threads, ab wo die Referenzimplementierung die bessere Performance aufweist. In Abbildung 6.5c ist jedoch erkennbar, dass die einzelne globale Deque nur eine minimal schlechtere Performance besitzt als die Referenzimplementierung. In diesem Fall liegt der zusätzliche Laufzeitaufwand bei unter 1.5%.

6.1.4. k-D Heap

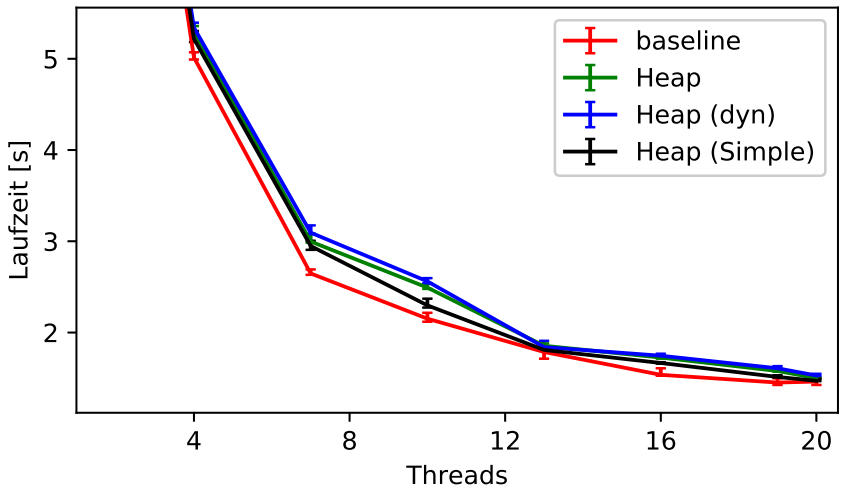


Abbildung 6.6.: Vergleich verschiedener Heap-Sortierungen (Thread Skalierung)

Um die Performance des k-D Heaps zu evaluieren, wurden beide Priorisierungsstrategien gegeneinander und gegen die Referenzversion verglichen. Diese Ergebnisse sind in Abbildung 6.6 dargestellt. Der Zusatz `dyn` bezeichnet dabei die Implementierung, welche die Prioritäten zur Laufzeit immer wieder aktualisiert. Für die Messung wurde das TRACE-Minimalbeispiel verwendet. Die Problemgröße beträgt $\{300, n\}$, wobei n die steigende Anzahl Threads darstellt. Die Messungen wurden so lange wiederholt bis die jeweiligen Konfidenzintervalle kleiner als 0.1s waren. Um die Lesbarkeit des

Diagramms im Bereich mit vielen Threads zu verbessern wurde der erste Messwert abgeschnitten.

Zuerst fällt auf, dass keine der untersuchten Heap-Varianten eine bessere Performance aufweist als die Referenzversion. Außerdem scheint die vereinfachte Sortierung deutlich effizienter zu sein als die vollständige bisherige Sortierung. Die dynamische Aktualisierung der Prioritäten verschlechtert die Laufzeit weiter. Ein möglicher Grund hierfür ist, dass die Sortierung des globalen Heaps nicht mehr in der Idle-Phase der Worker-Threads stattfindet, da dies eine zusätzliche Synchronisierung erfordern würde. Stattdessen wird die Sortierung über einen neuen Task implementiert. Es scheint, als wäre dieser zusätzliche Rechenaufwand größer als der durch ihn zu erwartende Vorteil.

Da der Heap jedoch theoretisch eine bessere Skalierungseigenschaft besitzt als die Referenzversion, wurden die drei Fälle in einem weiteren Benchmark miteinander verglichen. Dabei wurde der parallele Regionen Testfall verwendet. Die Anzahl der Threads wird dabei immer weiter erhöht. Da davon auszugehen ist, dass die finale Implementierung in TRACE mehr Tasks besitzen wird als das Minimalbeispiel, besteht die Möglichkeit, dass der Heap sich durch die möglicherweise bessere Skalierung als sinnvoller Kandidat für die tatsächliche Implementierung in TRACE eignet.

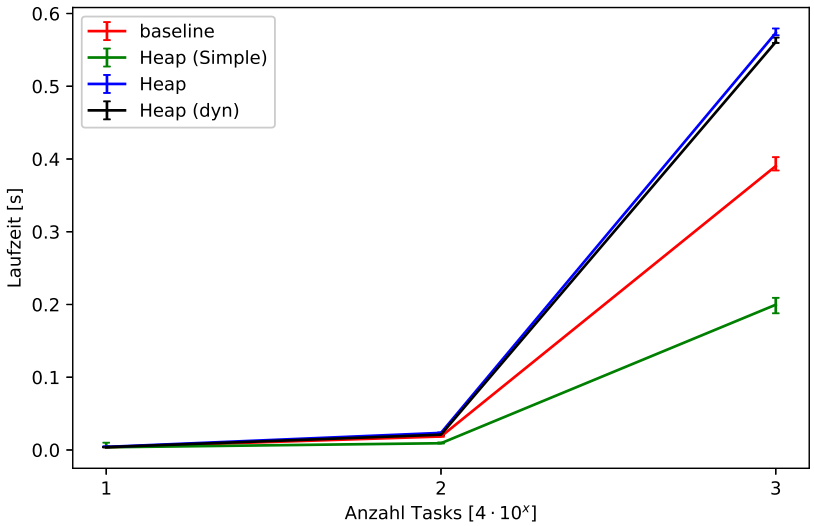


Abbildung 6.7.: Vergleich verschiedener Heap-Sortierungen (Task Skalierung)

Die Ergebnisse dieses Skalierungstests sind in Abbildung 6.7 dargestellt. Gemessen wurde bis zu einem maximalen Konfidenzintervall von 0.1s, jedoch sind die tatsächlichen Konfidenzintervalle deutlich kleiner. Zu beachten ist, dass die x-Achse logarithmisch eingeteilt ist. Das TRACE-Minimalbeispiel besitzt bei der Ausführung mit 20 Threads etwa 100 Tasks. Erkennbar ist, dass in diesem Beispiel bei 400 Tasks bereits ein deutlicher Laufzeitunterschied zwischen dem einfachen k-D Heap und der Referenzversion existiert. Im Fall mit 4000 Tasks ist der Performancevorteil des vereinfachten Task Heaps bereits annähernd Faktor zwei. Es muss jedoch beachtet werden, dass die Tasks in dem Beispiel keinerlei Daten verarbeiten und daher die Lokalität der Daten keinen signifikanten Einfluss hat. Um den Einfluss auf die Lokalität der einzelnen Sortierungskriterien zu testen, wurden für das Minimalbeispiel die Cache-Miss Raten gemessen. Dafür wurden mittels `perf` Messungen durchgeführt, bis das Konfidenzintervall für die Cache-Miss Rate kleiner als 0.5% war. Es wurde ein sehr kleiner Testfall mit vielen Threads mit einer Größe von $\{100, 20\}$ gewählt.

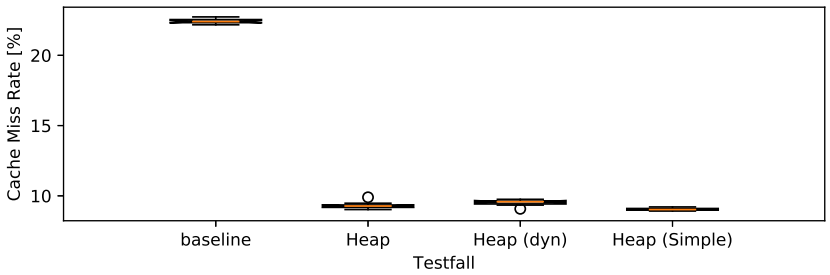


Abbildung 6.8.: Cache-Miss Raten für Heap-Implementierungen

Die Ergebnisse sind in Abbildung 6.8 dargestellt. Zwar scheint auf den ersten Blick die Referenzversion eine erheblich höhere Miss-Rate aufzuweisen, jedoch zeigt ein genauerer Blick in die Daten, dass diese auch etwa zehn mal mehr Cache-Zugriffe ausführt. Dieser Unterschied lässt sich vermutlich auf die vollständig unterschiedlichen Implementierungen zur Task-Auswahl zurückführen. Daher ist es schwierig die Messwerte mit denen der Heap-Implementierungen zu vergleichen. Jedoch kann man die drei Heap-Implementierungen miteinander vergleichen, da sich diese nur durch die Sortierungsstrategie unterscheiden. Dabei stellt man fest, dass die Cache-Miss Rate der vereinfachten Sortierung nahezu identisch ist mit der der klassischen Sortierung. Bestätigt wird dies zusätzlich durch die in Abbildung 6.3 gezeigte Effektivität der

`CORE_DIST` Strategie. Die hier untersuchte Sortierung ist eng mit der Sortierung nach der Distanz zum affinen Rechenkern verwandt, jedoch weiterhin eine Vereinfachung.

6.1.5. Einfluss von Abhängigkeiten auf Task-Datenstrukturen

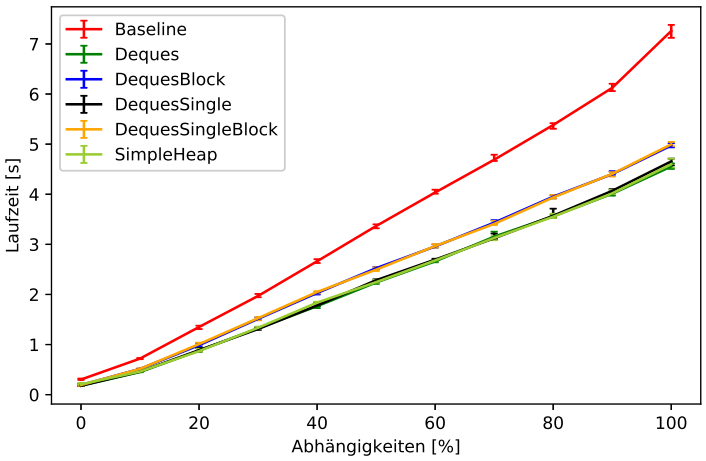


Abbildung 6.9.: Einfluss der Abhängigkeiten auf Task-Datenstrukturen

Um zu bewerten, welche der in den vorherigen Abschnitten untersuchten Datenstrukturen am besten geeigneten sind, muss auch der Einfluss der Parallelität des Task-Graphen betrachtet werden. Hierfür wurde der $G(n, p)$ Testfall mit 2500 Tasks und 30 Stichproben pro Messpunkt für p verwendet. In Abbildung 6.9 sind die gemessenen Laufzeiten abgebildet. Als ersten kann festgestellt werden, dass die Referenzversion besonders schlecht auf zusätzliche Abhängigkeiten reagiert. Bei einer vollständige sequentiellen Task-Ausführung (100%) entsteht beinahe ein Faktor zwei zwischen den alternativen Datenstrukturen und der Referenzimplementierung. Eine weitere Auffälligkeit ist, dass die blockierenden Versionen der Deques schlechter auf viele Abhängigkeiten reagieren als die nicht-blockierenden Varianten. Zwischen den nicht-blockierenden Deques und dem k-D Heap kann kein signifikanter Unterschied registriert werden. Es lässt sich daher festhalten, dass besonders für Task-Graphen mit vielen Abhängigkeiten zwischen den einzelnen Tasks eine nicht-blockierende Deque oder ein k-D Heap einen signifikanten Performancegewinn bieten können.

Daher ist es wichtig, dass für die finale Implementierung in TRACE dieser Aspekt ebenfalls berücksichtigt wird.

6.1.6. Lokalitäts-Optimierung

Um die in Abschnitt 5.4 vorgestellten Variationen bewerten zu können, wurde zunächst die Auswirkungen auf die Cache-Performance untersucht. Dafür wurden einige Experimente mit dem `perf` Profiler und dem TRACE Minimalbeispiel durchgeführt. Als Parameter wurde $\{100, 20\}$ verwendet.

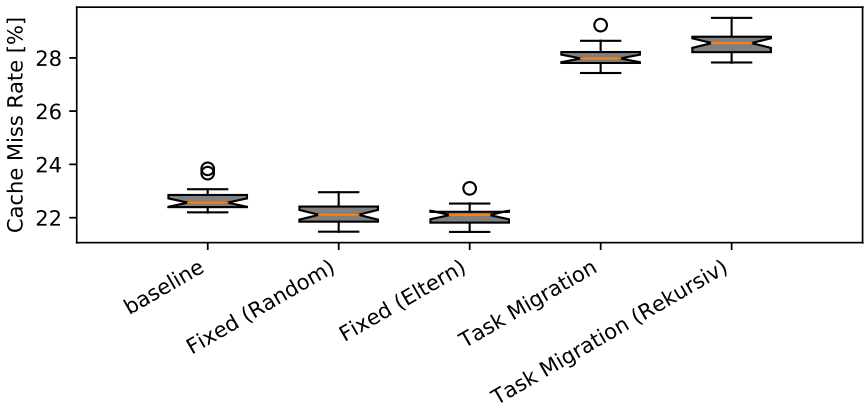


Abbildung 6.10.: Cache-Performance verschiedener Experimente

Die Ergebnisse der Untersuchung sind in Abbildung 6.10 dargestellt. Zuerst fällt auf, das beide Task-Migrationsstrategien eine erheblich schlechtere Cache-Performance als die Referenzversion aufweisen. Beide Varianten zeigen mehr als 20% mehr Cache-Misses als die Referenz. Auf der anderen Seite zeigen beide Modifikationen mit festen Zuordnungen eine leichte Verbesserung der Cache-Performance. Die Differenz zwischen einer zufälligen Zuordnung und der Zuordnung nach Elterntasks ist zu gering, um einen statistisch signifikanten Unterschied festzustellen. Unter Betrachtung dieser beiden Ergebnisse wurde eine Untersuchung der Laufzeit des Minimalbeispiels durchgeführt. Zu erwarten ist, dass die Testfälle mit aktivierter Task-Migration eine schlechtere Laufzeit aufweisen. Ob die minimale Cache-Verbesserung der Task-

Fixierung die reduzierte Parallelität ausgleicht, muss untersucht werden. Für die Messung wurde eine Testfallgröße von 400 gewählt.

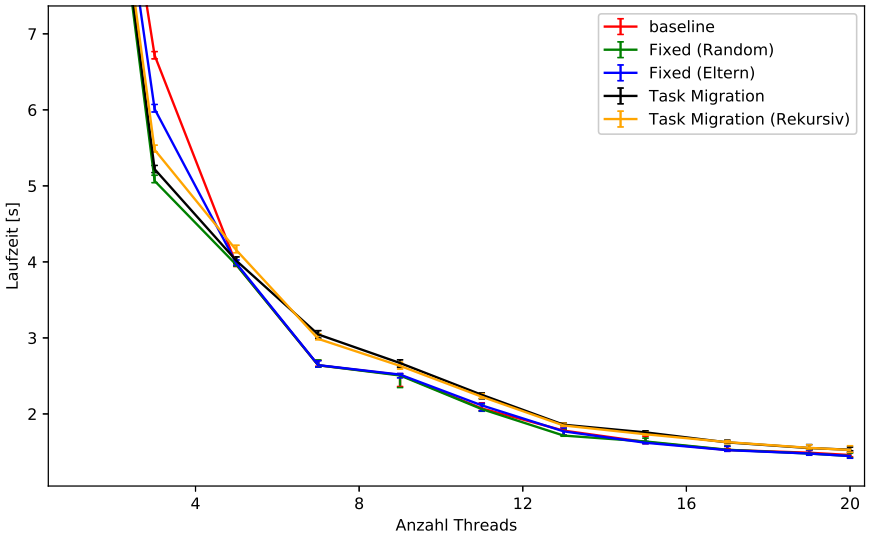


Abbildung 6.11.: Laufzeitvergleich verschiedener Lokalitäts-Untersuchungen

In Abbildung 6.11 sind die gemessenen Laufzeiten für verschiedene Thread-Anzahlen dargestellt. Ähnlich zu den Untersuchungen zu den Ready-Deques in Abschnitt 6.1.3 zeigen sich die untersuchten Änderungen für eine geringe Anzahl Threads deutlich performanter. Erneut zeichnet sich nach fünf Threads eine Inversion dieser Beobachtung ab.

Es zeigt sich, dass die Varianten mit aktivierter Task-Migration, wie erwartet, eine deutlich schlechtere Performance aufweisen. Auf der anderen Seite zeigt sich, das die Task-Fixierung bei hohen Thread-Anzahlen nahezu keine Auswirkung auf die Laufzeit hat. Es ist nicht möglich einen statistisch signifikanten Unterschied in den Daten zu identifizieren. Da jedoch eine klare Performanceverbesserung für niedrige Thread-Anzahlen gemessen werden kann, muss für eine potentielle finale Scheduler-Implementierung evaluiert werden, ob keine negativen Auswirkungen für höhere Thread-Anzahlen beziehungsweise Task-Anzahlen entstehen. Da der typische Anwendungsfall die komplette Auslastung eines Rechenknotens umfasst, muss die Priorität dabei auf einer optimalen Performance für eine hohe Anzahl Threads liegen.

6.2. Fazit

Im Zuge dieser Arbeit wurde eines Analyse des Schedulers von TRACE angefertigt. Diese zeigt, dass der Scheduler im allgemeinen Tasks effektiv ausführt. Bei der Analyse wurden vor allem zwei potentielle Probleme festgestellt:

1. Schlechte Skalierung des Task-Arrays für große Task-Anzahlen
2. Möglicherweise ineffektive Task-Reihenfolge

Für ersteres Problem wurden eine Reihe von potentiellen Optimierungen vorgeschlagen, von denen jedoch oft nur in synthetischen Testfällen eine Beschleunigung gemessen werden konnte. Diese müssen für eine finale Implementierung in TRACE erneut evaluiert werden.

Für das Sortierungsproblem wurde gezeigt, dass die aktuelle Sortierungsstrategie nicht das Optimum darstellt. Stattdessen wurde eine effizientere Sortierung vorgestellt und implementiert.

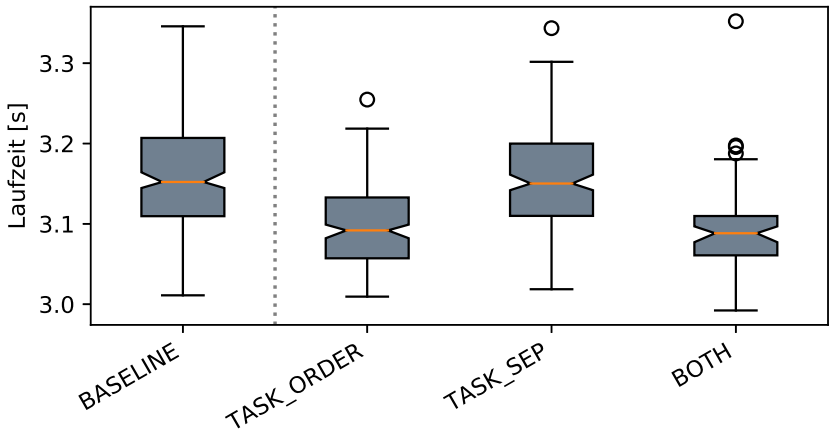


Abbildung 6.12.: Übersicht über effektivste Änderungen

Um abschließend beurteilen zu können, welche Modifikationen sinnvoll erscheinen, wurde eine finale Messreihe mit einigen der Änderungen durchgeführt. Die Ergebnisse sind in Abbildung 6.12 dargestellt. Hierbei wird die verbesserte Sortierung (Testfall `T1`) je einmal mit und einmal ohne Task-Separation mit der Referenzversion verglichen.

Die Messungen wurden erneut mit dem Minimalbeispiel unter Verwendung der Parameter $\{400, 20\}$ durchgeführt. Als Konfidenzintervall wurde ein Limit von 0.02s festgelegt.

Zuerst fällt auf, dass immer noch kein signifikanter Unterschied zwischen der Referenzversion und der Task-Separation festgestellt werden kann. Dies bestätigt sich auch weiter dadurch, dass ebenfalls kein signifikanter Unterschied zwischen den beiden Sortierungsfällen erkennbar ist. Jedoch stellt man erneut fest, dass durch die neue Sortierung, in beiden Fällen, ein deutlicher Performancevorteil für den Scheduler erzielt werden kann. Abschließend lässt sich feststellen, dass die Optimierungen in dieser Arbeit zu einer Laufzeitreduktion um etwa zwei bis drei Prozent führen.

Da der Implementierungsaufwand aller Optimierungen überschaubar ist, sind alle untersuchten Varianten für die Anwendung in einer späteren TRACE Implementierung geeignet.

7. Ausblick

Der in dieser Arbeit dokumentierte und analysierte Scheduler wird in den nächsten Monaten in die TRACE Hauptversion implementiert. Dabei ist es wichtig, die in dieser Arbeit untersuchten Modifikationen erneut zu überprüfen, um sicherzustellen, dass die Ergebnisse auf die neue Umgebung übertragbar sind. Insbesondere bei den Sortierstrategien, als auch bei der Komplexität der Datenstrukturen, sind Unterschiede wahrscheinlich, da sich sowohl die Anzahl als auch das Verhalten der Tasks deutlich ändert.

Neben diesen Kontrollaufgaben sollen auch weitere potentielle Optimierungen für den Scheduler implementiert und evaluiert werden. Eine in dieser Arbeit noch nicht näher betrachtete Optimierungsform ist das Optimieren des Task-Graphens. Wie bereits in Abschnitt 4.1.1 erwähnt, können eine Reihe von weiteren Methoden zum Kombinieren von Tasks untersucht werden. Dies ist besonders interessant, da sich die Anzahl der Tasks deutlich erhöhen wird.

Auf Seiten der zentralen Task-Datenstrukturen sollte untersucht werden, ob die gesteigerte Anzahl an Tasks möglicherweise dafür sorgt, dass der multidimensionale Heap eine höhere Performance zeigt als das bestehende Task-Array. Darüber hinaus muss überprüft werden, ob eine geeignete Alternative zum Task-Heap existiert, welche das Problem des Mutex umgeht. Hierfür kommen vor allem Lock-Free Datenstrukturen in Frage.

Um die Task-Reihenfolge weiter zu optimieren, können ebenfalls Methoden der künstlichen Intelligenz bzw. des maschinellen Lernens eingesetzt werden. Diese könnten entweder online verwendet werden, um die Task-Reihenfolge direkt zu steuern, oder offline zur Gewichtung der einzelnen Sortierungskriterien benutzt werden.

Literatur

[1] E. Ayguade u. a. „The Design of OpenMP Tasks“. In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (03/2009), S. 404–418. DOI: 10.1109/tpds.2008.105.

[2] Robert D. Blumofe und Charles E. Leiserson. „Scheduling multithreaded computations by work stealing“. In: *Journal of the ACM (JACM)* 46.5 (09/1999), S. 720–748. DOI: 10.1145/324133.324234.

[3] Robert D. Blumofe u. a. „Cilk“. In: *ACM SIGPLAN Notices* 30.8 (08/1995), S. 207–216. DOI: 10.1145/209937.209958.

[4] François Broquedis u. a. „hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications“. In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Hrsg. von IEEE. Pisa, Italy, 02/2010. DOI: 10.1109/PDP.2010.67. URL: https://hal.inria.fr/inria-00429889.

[5] Quan Chen u. a. „CAB: Cache Aware Bi-tier Task-Stealing in Multi-socket Multi-core Architecture“. In: *2011 International Conference on Parallel Processing*. IEEE, 09/2011. DOI: 10.1109/icpp.2011.32.

[6] Daniel Cordeiro u. a. „Random graph generation for scheduling simulations“. In: *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. ICST, 2010. DOI: 10.4108/icst.simutools2010.8667.

[7] Yuzheng Ding und Mark Allen Weiss. „The K-D Heap: An Efficient Multi-Dimensional Priority Queue“. In: *Proceedings of the Third Workshop on Algorithms and Data Structures*. WADS '93. Berlin, Heidelberg: Springer-Verlag, 1993, S. 302–313.

[8] Christoph Ersfeld. *Organisation von Caches*. 08. 12. 2002. URL: https://berrendorf.inf.h-brs.de/lehre/ws0203/vups2/Seminar/01_Ersfeld_Caches.pdf (besucht am 06. 08. 2020).

[9] Daniel Grünewald, Roman Iakymchuk und Dimitar Stoyanov. *SPECTRA Task granularity analysis report*. Techn. Ber. Fraunhofer-Institut für Techno- und Wirtschaftsmathematik ITWM, 17. 05. 2019.

[10] Heinz Peter Gumm u. a. *Einführung in die Informatik*. Berlin, Boston: De Gruyter, 2013. DOI: https://doi.org/10.1524/9783486719956. URL: https://www.degruyter.com/view/title/313768 (besucht am 28. 07. 2020).

[11] Torsten Hoefler und Roberto Belli. „Scientific benchmarking of parallel computing systems“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. ACM Press, 2015. DOI: 10.1145/2807591.2807644.

[12] Wooyoung Kim und Michael Voss. „Multicore Desktop Programming with Intel Threading Building Blocks“. In: *IEEE Software* 28.1 (01/2011), S. 23–31. DOI: 10.1109/ms.2011.12.

[13] Christoph Lameter. „NUMA (Non-Uniform Memory Access): An Overview“. In: *Queue* 11.7 (07/2013), S. 40. DOI: 10.1145/2508834.2513149.

[14] Marc Moreno Maza. *Multithreaded Parallelism and Performance Measures*. University of Western Ontario, London, Ontario (Canada). URL: https://www.csd.uwo.ca/~mmorenom/cs3101_Winter_2013/Multithreaded_Parallelism_and_Performance_Measures.pdf (besucht am 10. 07. 2020).

[15] Marius Messerschmidt. *Entwurf von Verfahren zur Bewertung von Task-basierten Schedulern für Shared Memory Parallelisierung*. Deutsches Zentrum für Luft- und Raumfahrt e.V. & Duale Hochschule Baden-Würtemberg Mannheim.

[16] o.V. *Perf Wiki*. 12. 06. 2020. URL: https://perf.wiki.kernel.org/index.php/Main_Page (besucht am 09. 09. 2020).

[17] o.V. *Was ist NUMA?* VMWare, Inc. 31. 05. 2019. URL: https://docs.vmware.com/de/VMware-vSphere/6.5/com.vmware.vsphere.resmgmt.doc/GUID-1DAB8F35-BA86-4063-8459-55D2979B593E.html (besucht am 06. 08. 2020).

[18] *OpenMP Application Programming Interface*. Version 5,0. OpenMP Architecture Review Board. November 2018. Kap. 2.10.6 Task Scheduling. URL: https://www.openmp.org/spec-html/5.0/openmpsu51.html (besucht am 20. 07. 2020).

[19] Kristin Potter u. a. „Methods for presenting statistical information: The box plot“. In: *Visualization of large and unstructured data sets* 4 (2006), S. 97–106.

[20] Moinuddin K. Qureshi u. a. „Adaptive insertion policies for high performance caching“. In: *Proceedings of the 34th annual international symposium on Computer architecture - ISCA '07*. ACM Press, 2007. DOI: 10.1145/1250662.1250709.

[21] H. El-Rewini, H. H. Ali und T. Lewis. „Task scheduling in multiprocessing systems“. In: *Computer* 28.12 (1995), S. 27–37.

[22] Arch Robison, Michael Voss und Alexey Kukanov. „Optimization via Reflection on Work Stealing in TBB“. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 04/2008. DOI: 10.1109/ipdps.2008.4536188.

[23] Corentin Rossignon u. a. „A NUMA-Aware Fine Grain Parallelization Framework for Multi-core Architecture“. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 05/2013. DOI: 10.1109/ipdpsw.2013.204.

[24] Andrew S. Tanenbaum und Herbert Bos. „Moderne Betriebssysteme“. In: 4. Aufl. Pearson Studium - IT Ser. Pearson Education Deutschland GmbH, 05/2016. Kap. 2 Prozesse und Threads, S. 126–226.

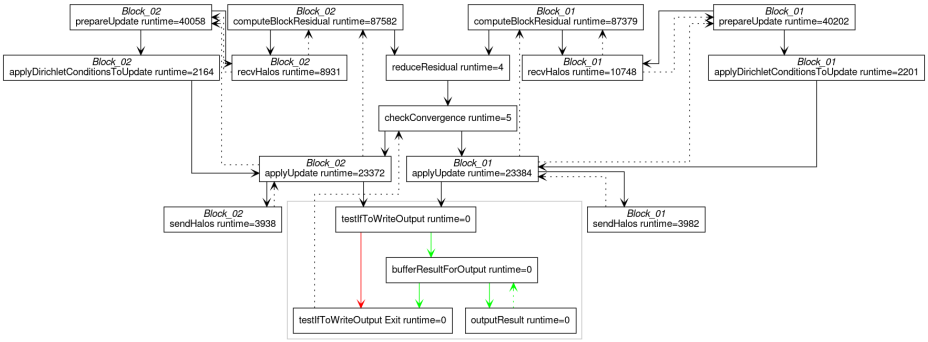
[25] Peter Thoman u. a. „A taxonomy of task-based parallel programming technologies for high-performance computing“. In: *The Journal of Supercomputing* 74.4 (01/2018), S. 1422–1434. DOI: 10.1007/s11227-018-2238-4.

[26] *Transitive reduction - MATLAB transreduction*. MathWorks. URL: https://www.mathworks.com/help/matlab/ref/digraph.transreduction.html (besucht am 17. 08. 2020).

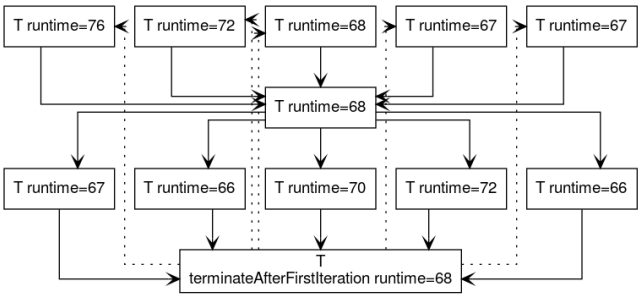
[27] Dmitry Vyukov. *Task Scheduling Strategies*. URL: http://www.1024cores.net/home/scalable-architecture/task-scheduling-strategies (besucht am 10. 09. 2020).

Anhang A.

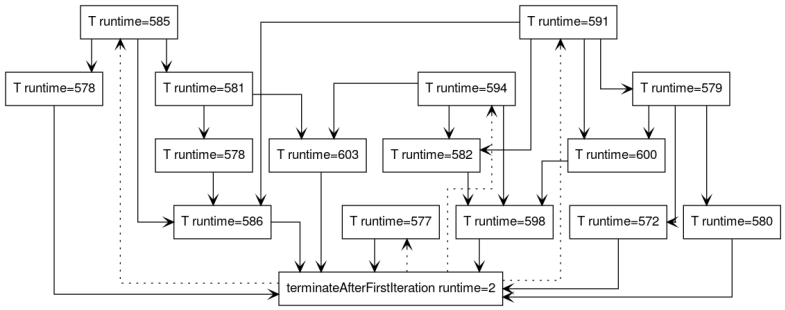
Beispielhafte Task Graphen der Testfälle



(a) TRACE-Minimalbeispiel (2 Threads)



(b) Parallele Regionen, 2 Regionen mit je 5 Tasks

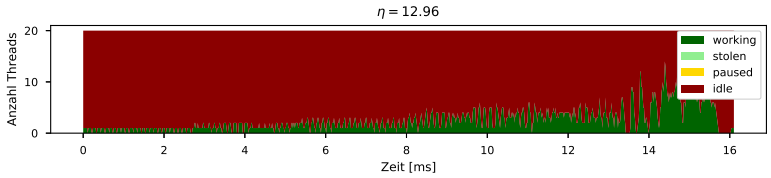


(c) $G(n, p)$ Methode mit $G(15, 0.25)$ und Seed 2

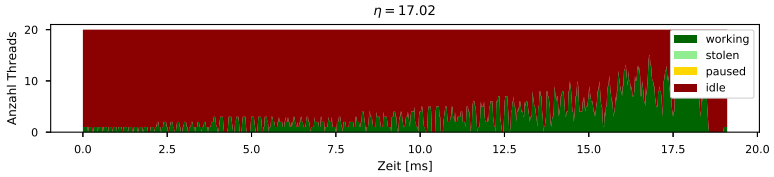
Abbildung A.1.: Task-Graphen verschiedener Testfälle

Anhang B.

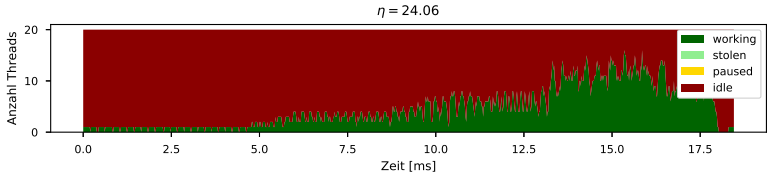
Traces verschiedener Task-Größen



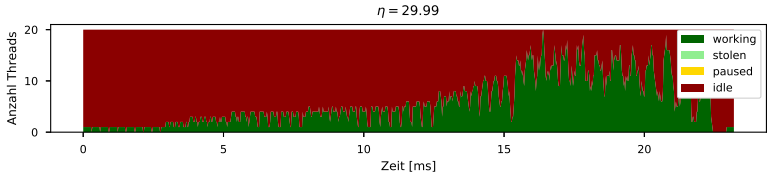
(a) Task Größe 10μs



(b) Task Größe 50μs



(c) Task Größe 100μs



(d) Task Größe 200μs

Abbildung B.1.: Vergleich der Thread-Auslastung verschiedener Task-Größen des parallele Regionen Testfalls