UNI KASSEL
VERSITÄT

Universität Kassel

Bachelorarbeit

Effizienzvergleich von zwei Lastenbalancierungsverfahren für parallele Programmiersysteme

vorgelegt vor
Fachbereich Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik

Konstantin Stitz
35532242

Kassel, 16. März 2023

Gutachter:
Prof. Dr. Claudia Fohry
Prof. Dr. Gerd Stumme

Inhaltsverzeichnis

Selbstständigkeitserklärung **II**

1. Einleitung **1**

2. Grundlagen **4**
2.1. APGAS 4
2.2. Dynamische unabhängige Tasks 6
2.3. Work Stealing 7
2.4. GLB 9

3. Lifeline-Schema **11**

4. Randomisiertes Work Stealing und Terminierungserkennung **14**

5. Implementierung **17**
5.1. Initiale Verteilung 18
5.2. Randomisiertes Work Stealing 19
5.3. Terminierungserkennung 20

6. Experimente **22**
6.1. Überblick 22
6.2. Benchmarks 23
6.3. Vorversuche 25
6.4. Ergebnisse 28
6.5. Auswertung 33

7. Schlussbemerkungen **35**

Literatur **VII**

A. Anhang **X**

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Kassel, 16. März 2023

Konstantin Stitz

1. Einleitung

Programme können parallelisiert werden, um die Ausführungszeit zu verkürzen. Dabei werden Teilberechnungen, sogenannte *Tasks*, auf mehrere Rechner beziehungsweise Rechenkerne verteilt und gleichzeitig ausgeführt. Um eine hohe Effizienz und damit eine kurze Programmlaufzeit zu erreichen, sollten alle Kerne möglichst gleichmäßig ausgelastet werden. Dies lässt sich mithilfe von *Lastenbalancierungsverfahren* erreichen. Die vorliegende Bachelorarbeit vergleicht zwei spezifische Lastenbalancierungsverfahren, von denen eines selbst implementiert wurde. Der Vergleich erfolgt mithilfe mehrerer Benchmarks, indem die jeweiligen Laufzeiten gemessen werden.

Als paralleles Programmiersystem wird die Bibliothek *GLB* (Global Load Balancing [14]), in Kombination mit dem Programmiermodell *APGAS* (Asynchronous Partitioned Global Adress Space [13]) verwendet. Es werden *dynamische unabhängige Tasks* betrachtet. Darunter versteht man Tasks, die weitere Tasks starten können und deren Ergebnisse erst am Ende durch Reduktion zusammengeführt werden. Die beiden zu vergleichenden Lastenbalancierungsverfahren basieren auf *kooperativem Work Stealing*. Dabei ist ein *Worker* ein Thread, der Tasks abarbeitet. Ein Task-Pool ist die Ansammlung aller Tasks eines Workers. Bei kooperativem Work Stealing stellt ein Worker, dessen Task-Pool leer ist, eine Stehlanfrage an einen anderen Worker, um von diesem einen Teil seiner Tasks zu erhalten. Der anfragende Worker wird dementsprechend *Dieb*, und der angefragte Worker *Opfer* genannt. Das Opfer überprüft regelmäßig eingegangene Anfragen und bearbeitet diese.

Das erste Lastenbalancierungsverfahren, für das zu Beginn der Bachelorarbeit bereits eine Implementierung vorlag [4], nutzt das sogenannte *Lifeline-Schema* [14]. Nachfolgend wird die Implementierung dieses Lastenbalancierungsverfahrens mit *LL* abgekürzt. Hat ein Worker seine

Tasks abgearbeitet, fragt er zuerst per Zufall andere Worker nach weiteren Tasks. Haben die angefragten Worker keine Tasks abzugeben, fragt er die ihm durch den sogenannten *Lifeline-Graphen* zugeordneten Worker nach Tasks. Haben auch diese keine weiteren Tasks abzugeben, legt sich der Worker schlafen. Er wird wieder aufgeweckt, wenn einer der ihm über den Lifeline-Graphen zugeordneten Worker genügend Tasks hat und ihm welche schickt.

Ziel dieser Bachelorarbeit war es, ein zweites Lastenbalancierungsverfahren zu implementieren, welches ohne einen Lifeline-Graphen auskommt und nur auf zufälligem *Work Stealing* basiert. Nachfolgend wird dieses Lastenbalancierungsverfahren als *randomisiertes Work Stealing* bezeichnet. Die erstellte Implementierung wird mit *RAN* abgekürzt. Sobald ein Worker alle seine Tasks beendet hat, versucht er so lange zufällig bei anderen Workern zu stehlen, bis er entweder erfolgreich ist oder alle Worker ihre Tasks beendet haben. Um dies zu erkennen, wurde eine Terminierungserkennung implementiert.

Die Messungen mit den Benchmarks haben gezeigt, dass LL in den meisten Fällen eine deutlich kürzere Laufzeit als RAN hat. Durch eine Änderung der Auswahl des Opfers kann RAN verbessert werden, sodass es schneller als LL ist. Allerdings ist die Auswahl des Opfers dann nicht mehr zufällig, zudem sind die Verbesserungen auch in LL aktivierbar. Daher ist das Lifeline-Schema dem randomisierten Work Stealing in den meisten Fällen überlegen und daher vorzuziehen.

Die vorliegende Arbeit gliedert sich in sieben Kapitel. Nach der Einleitung in Kapitel 1 folgen in Kapitel 2 allgemeine Grundlagen, beispielsweise zu APGAS und GLB. Kapitel 3 beschreibt das Lifeline-Schema. In Kapitel 4 wird das randomisierte Work Stealing sowie die Auswahl und Funktionsweise des verwendeten Terminierungsverfahrens vorgestellt. Kapitel 5 behandelt die Implementierung des randomisierten Work Stealings. Kapitel 6 stellt die Experimentierumgebung sowie die durchgeführten Experimente vor und diskutiert die Messergebnisse. Eine Zusammenfassung, die Einordnung der

Ergebnisse in den Kontext anderer Arbeiten und ein Ausblick auf weitere zu untersuchende Lastenbalancierungsverfahren schließen die Arbeit in Kapitel 7 ab.

2. Grundlagen

2.1. APGAS

Weit verbreitete Standards wie *MPI* [9] ermöglichen die Zusammenarbeit mehrerer Rechner durch den Austausch von Nachrichten. Es gibt keinen gemeinsamen Speicher, daher muss der Programmierer die Daten explizit verteilen. Dies ist für den Programmierer aufwendig zu implementieren und fehleranfällig. Als Alternative wurde das Programmiermodell *PGAS* (Partitioned Global Adress Space [5]) entwickelt. Hier werden die Rechner durch sogenannte *Places* repräsentiert. Jeder Place besitzt seinen eigenen lokalen Speicher, auf den die Prozessoren des jeweiligen Places direkt zugreifen können. Durch die Software werden jedoch die lokalen Speicher der verschiedenen Places zu einem gemeinsamen virtuellen Speicher zusammengefasst, sodass über das Netzwerk auch auf den Speicher der anderen Places zugegriffen werden kann.
APGAS (Asynchronous Partitioned Global Adress Space [13]) ist eine Weiterentwicklung des PGAS-Modells, die in der Bibliothek APGAS für Java implementiert wurde [8, 15]. Diese Implementierung wird gleichnamig wie das Modell mit APGAS abgekürzt. Im Gegensatz zum klassischen PGAS-Modell wird bei APGAS das Programm sequentiell mit einer *Aktivität* auf Place 0 gestartet. Während der Ausführung des Programms können dann weitere Aktivitäten gestartet oder beendet werden. Eine Aktivität ist dabei vergleichbar mit einem Java-Thread [9], jedoch effizienter implementiert.

Nachfolgend werden die wichtigsten Konstrukte vorgestellt, um mithilfe von APGAS eine parallele Programmausführung zu erreichen. Der auszuführende Programmcode **f** wird dabei als ein Lambda-Ausdruck angegeben:

- **async(f)**: Erzeugt eine parallele Aktivität mit dem Programmcode **f**. Die Aktivität wird auf dem gleichen Place gestartet, auf welchem **async(f)** ausgeführt wurde, sobald genügend Ressourcen zur Verfügung stehen.
- **at(p, f)**: Erzeugt eine synchrone Aktivität **f** auf Place **p**. Bei einer synchronen Aktivität wartet die aufrufende Aktivität auf die aufgerufene Aktivität, während bei einer asynchronen Aktivität nicht auf das Ergebnis gewartet wird.
- **asyncAt(p, f)**: Kombiniert beide Funktionen und ermöglicht die Erzeugung einer asynchronen Aktivität auf Place **p**.
- **finish(f)**: Die Umschließung eines Codeabschnitts **f** mit diesem Konstrukt sorgt dafür, dass am Ende auf die Beendigung aller in **f** erzeugten Aktivitäten gewartet wird. Eine Ausnahme bilden Aktivitäten welche mit **uncountedAsyncAt(p, f)** erzeugt wurden. Diese Aktivitäten haben die gleiche Funktionalität wie mit **asyncAt(p, f)** erzeugte Aktivitäten, jedoch ohne die Terminierungserkennung von **finish(f)**. Dadurch ist die Performance etwas besser.
- **here()**: Gibt den aktuellen Place zurück.
- **place(ID)**: Durch Angabe einer **ID**, zum Beispiel 0 für den ersten Place, erhält man den zu der **ID** gehörenden Place.
- **places()**: Gibt eine Liste aller vorhandenen Places zurück.

```java
public class ApgasExample {
    public static void main(String[] args) {
        finish(() -> {
            for (final Place p: places()) {
                asyncAt(p, () -> {
                    System.out.println("PlaceID:" + here().id);
                });
            }
        }
        System.out.println("Finished!")
    }
}
```

Listing 2.1: Beispielcode für die Verwendung von APGAS-Konstrukten

Zur Veranschaulichung der Verwendung von APGAS ist in Listing 2.1 Beispielcode abgebildet. Dort wird auf jedem Place eine asynchrone Aktivität gestartet, welche jeweils die ID des eigenen Places in der Konsole ausgibt. Erst nachdem alle Places ihre ID ausgegeben haben, wird Zeile 10 ausgeführt.

2.2. Dynamische unabhängige Tasks

Dynamische unabhängige Tasks, abgekürzt *DIT*, sind eine Form von Tasks, die in einem parallelen Programmiersystem gestartet und deren berechnete Ergebnisse wieder zusammengeführt werden können. Werden in einem parallelen Programmiersystem dynamische unabhängige Tasks verwendet, bedeutet dies, dass das Programm mit einem oder mehreren Tasks startet. Diese Tasks können weitere Tasks generieren, wobei die Schachtelungstiefe unbegrenzt ist. Es ist nicht festgelegt, ob und wie viele neue Tasks jeweils generiert werden.

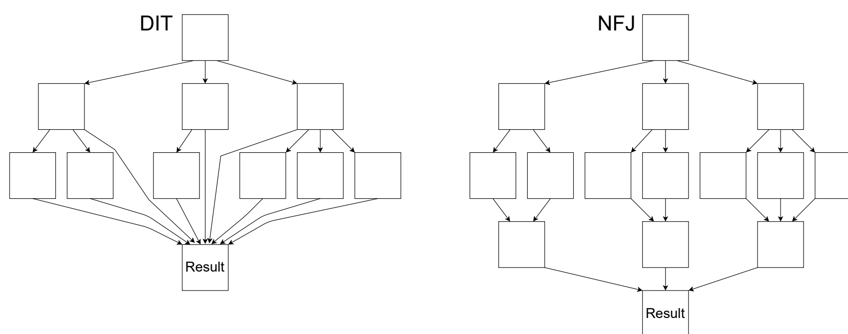

Abbildung 2.1.: Vereinfachter schematischer Vergleich von DIT (links) gegenüber NFJ (rechts)

Jeder Task berechnet ein Ergebnis. Bei DIT kann das Ergebnis nur durch die Elterntasks beeinflusst werden. Seiteneffekte durch den globalen Speicher sind also explizit verboten. Wenn ein Worker einen Task abschließt, wird sein Ergebnis mit den bisherigen Ergebnissen aller auf diesem Worker ausgeführten Tasks zusammengeführt. Sobald alle Tasks beendet sind, werden die Teilergebnisse aller Worker zusammengeführt. Eine Alternative zu DIT wäre beispielsweise *NFJ*, dort wird das Ergebnis eines Tasks im Unterschied zu DIT wieder an den Elterntask zurückgegeben. Dadurch enthält NFJ bereits implizit eine Terminierungserkennung, welche bei DIT zusätzlich benötigt wird. In Abbildung 2.1 sind sowohl DIT als auch NFJ sehr vereinfacht schematisch dargestellt. In dieser Arbeit wird ausschließlich DIT verwendet.

2.3. Work Stealing

Wenn ein Worker alle zum Programmstart erhaltenen Tasks und deren Kindertasks abgearbeitet hat, versucht er, Tasks von anderen Workern zu stehlen. Beim kooperativen Work Stealing [10] sendet dieser Worker, der Dieb, dazu eine Anfrage an das Opfer. Jeder Worker, einschließlich des Opfers, unterbricht

regelmäßig die Abarbeitung der Tasks, um die eingegangenen Anfragen zu beantworten. Der Dieb wartet in der Zwischenzeit auf die Antwort des Opfers. Kann das Opfer die Anfrage des Diebes erfüllen, schickt es einige Tasks an ihn. Diese fügt der Dieb seinem Task-Pool hinzu.
Falls keine Abgabe von Tasks möglich ist, teilt das Opfer dies dem Dieb mit. Dieser kann dann eine weitere Stehlanfrage an den nächsten Worker senden. Der nächste Worker wird durch das verwendete Verfahren ausgewählt, beispielsweise per Zufall. Je nach Implementierung kann sich der Dieb auch deaktivieren, beispielsweise nachdem mehrere Stehlanfragen nicht erfolgreich waren.
Eine andere Methode für Work Stealing ist das *koordinierte Work Stealing* [10]. Hier legt das Opfer die zu stehlenden Tasks in einer speziellen Datenstruktur ab, sodass ein Dieb sie sich aneignen kann, ohne das Opfer zu unterbrechen. In dieser Arbeit wird jedoch ausschließlich kooperatives Work Stealing verwendet.
Weiterhin kann bei Work Stealing unterschieden werden, nach welcher Methode der Dieb das Opfer auswählt. Bei randomisiertem Work Stealing wird anhand von Zufallszahlen bestimmt, welcher Worker das nächste Opfer ist. Dies erfordert zwar keine zusätzlichen Berechnungen, kann aber dazu führen, dass ein Worker mit nur wenigen Tasks angefragt wird, während andere Worker noch viele Tasks besitzen.
LL enthält eine Erweiterung, die eine mögliche Optimierung für die zufälligen Stehlanfragen darstellt. Diese Erweiterung wurde in RAN integriert. Mit dem Parameter **localopt** kann diese Erweiterung aktiviert werden. Standardmäßig ist **localopt** auf 0 gesetzt. Bei diesem Wert ist die Erweiterung deaktiviert und die Stehlanfragen sind völlig zufällig.
Wird der Wert von **localopt** auf 2 gesetzt[1], ist die Erweiterung aktiviert. Es wird auf Lokalität optimiert und dementsprechend zuerst eine Stehlanfrage an einen Worker auf dem selben Place gesendet. Zudem wird das Opfer nicht per Zufall

[1] In [4] wurde auch ein Wert 1 betrachtet, der aber unterlegen war und daher verworfen wurde.

ausgewählt, sondern es wird über eine zusätzliche Funktion derjenige Worker bestimmt, welcher auf dem selben Place die meisten Tasks besitzt. Dadurch erhöht sich die Chance auf einen erfolgreichen Diebstahl, dem steht jedoch der zusätzliche Rechenaufwand gegenüber, um den Worker mit den meisten verbleibenden Tasks zu finden. Sollte die Stehlanfrage dennoch nicht erfolgreich sein, wird sie nicht sofort abgewiesen, sondern zunächst der Worker mit den meisten Tasks auf dem Place des Opfers bestimmt. Gibt es einen Worker auf dem Place des aktuellen Opfers, der mehr Tasks als das Opfer hat, wird die Stehlanfrage an diesen Worker weitergeleitet. Schlägt auch dies fehl, findet anschließend wieder randomisiertes Work Stealing statt, wobei jeder Worker von jedem Place das Opfer sein kann. Wenn die Stehlanfrage bei einem zufälligen Stehlversuch nicht erfolgreich ist, wird sie ebenfalls nicht sofort abgewiesen, sondern der Worker mit den meisten Tasks auf dem Place des Opfers bestimmt und die Stehlanfrage an diesen weitergeleitet.

2.4. GLB

GLB (Global Load Balancing [14]) bezeichnet ein Framework für die automatische Lastenbalancierung bei APGAS-Programmiersystemen. Sowohl LL als auch RAN sind in Java implementierte Varianten von GLB, welche mithilfe von APGAS, siehe Abschnitt 2.1, realisiert wurden. GLB wurde bereits in vielen verschiedenen Varianten implementiert [9, 17, 18].

Die hier genutzte GLB-Bibliothek [4] arbeitet mit dynamischen unabhängigen Tasks, siehe Abschnitt 2.2, und verwendet kooperatives Work Stealing, siehe Abschnitt 2.3. Worker werden in APGAS mithilfe einer Aktivität umgesetzt und sind für die Ausführung von Tasks erforderlich.

Jedem Worker wird eine sogenannte *Bag* zugeordnet. Darin werden die Tasks jedes Workers gespeichert. Die Bag ist eine Java-Klasse und der Task-Pool eines Workers. Sie übernimmt viele Aufgaben, wie beispielsweise die Abarbeitung eines

Tasks. Die notwendigen Funktionen müssen vom Anwender der GLB-Bibliothek implementiert werden.
Die GLB-Bibliothek ist so aufgebaut, dass jeder Place die gleiche, vor der Ausführung festgelegte, Anzahl an Workern besitzt. Die Initialisierung des Task-Pools kann dynamisch oder statisch erfolgen, wobei bei der statischen Initialisierung die Tasks bereits vor dem eigentlichen Programmstart den Workern zugeordnet werden. Dazu wird auf jedem Worker eine Funktion ausgeführt, die Werte für die Start-Tasks setzt. Bei der statischen Initialisierung ist keine Verteilung zu Beginn der Ausführung notwendig, da jeder Worker seine Start-Tasks aus den gesetzten Werten selbst generieren kann. Dagegen ist es bei der dynamischen Initialisierung notwendig, die vorhandenen Start-Tasks beim Programmstart zu verteilen.

3. Lifeline-Schema

In diesem Kapitel wird LL vorgestellt, eine Variante von GLB [4]. Diese nutzt das Lifeline-Schema, welches die initiale Startphase der Worker bestimmt, das Work Stealing beeinflusst und für die Terminierungserkennung verantwortlich ist. Genutzt wird dafür der sogenannte Lifeline-Graph. Jeder Worker wird als Knoten in dem Graphen betrachtet und besitzt *Lifeline-Buddies*, was andere Worker sind. Die Verbindungen zwischen den verschiedenen Workern, den Lifeline-Buddies, können als Kanten des Lifeline-Graphen betrachtet werden. Abbildung 3.1 zeigt beispielhaft einen möglichen Lifeline-Graphen für sechs Worker. In dieser Variante hat jeder Worker zwei Lifeline-Buddies und ist gleichzeitig Lifeline-Buddy von zwei anderen Workern. Es gibt verschiedene Strategien zur Generierung des Lifeline-Graphen, worauf in dieser Arbeit jedoch nicht genauer eingegangen wird.

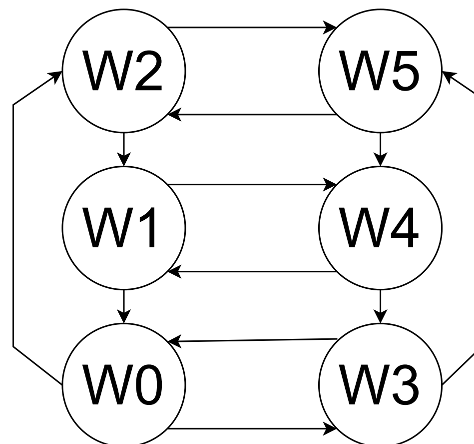

Abbildung 3.1.: Beispiel eines Lifeline-Graphen bei 6 Workern

Beim Programmstart wird bei einer dynamischen Initialisierung zunächst Worker 0 auf Place 0 gestartet und diesem alle Start-Tasks übergeben. Bei einer statischen Initialisierung wird von jedem Place der erste Worker gestartet, eine Übergabe von Tasks beim Programmstart ist bei einer statischen Initialisierung nicht notwendig. Die Startwerte für die Tasks wurden bei der statischen Initialisierung

bereits vor dem eigentlichen Programmstart auf jedem Worker gesetzt. Sowohl bei der dynamischen als auch bei der statischen Initialisierung werden die Worker innerhalb eines **finish**-Konstrukts gestartet.

```
1  while(true) {
2      do {
3          bag.processTasks()
4          if (isInitialStartup) {
5              activateAdditionalWorkersAndDistributeStartTasks()
6          } else {
7              bag.processStealAndLifelineRequests()
8          }
9      } while (bag.notEmpty())
10     if (localopt != 0) {
11         tryLocalOptimisedSteal()
12     }
13     tryRandomSteals()
14     if (bag.isEmpty()) {
15         sendLifelineRequests()
16         return()
17     }
18 }
```

Listing 3.1: Pseudocode für den Ablauf eines Workers bei LL

Jeder Worker besteht im Wesentlichen aus einer Endlosschleife, die nur durch ein **return** verlassen werden kann, wenn bestimmte Bedingungen erfüllt sind. Der vereinfachte Pseudocode in Listing 3.1 verdeutlicht den Ablauf dieser Endlosschleife. Der gestartete Worker beginnt zunächst mit der Abarbeitung einer in den GLB-Einstellungen festgelegten Anzahl von Tasks, siehe Zeile 3. Unmittelbar nach dem Programmstart, siehe Zeile 4 und 5, werden alle Worker aktiviert, von denen der aktuelle Worker der Lifeline-Buddy ist. Die neu gestarteten Worker starten in gleicher Weise weitere Worker, bis alle Worker im parallelen Programmiersystem gestartet wurden. Bei dynamischer Initialisierung werden in den Aktivierungsnachrichten Tasks zur Verteilung an die Worker

mitgeschickt. Im späteren Programmverlauf werden an dieser Stelle, siehe Zeile 6 und 7, eingegangene Stehlanfragen verarbeitet.

Der Worker durchläuft die Schleife von Zeile 2 bis 9, bis er keine Tasks mehr in seiner Bag hat. Dann verlässt er die Schleife und beginnt mit Work Stealing. Zuerst wird bei aktivierter Erweiterung ein optimierter Stehlversuch gestartet, siehe Zeile 10 bis 12. Anschließend wird randomisiertes Work Stealing versucht, siehe Zeile 13, welches bereits im Abschnitt 2.3 erläutert wurde. Wenn keine der zufälligen Stehlanfragen erfolgreich war, greift das Lifeline-Schema, Zeile 14 bis 17. Der Dieb sendet nacheinander eine Stehlanfrage an alle seine Lifeline-Buddies und verlässt anschließend über das **return** die Endlosschleife, falls noch keine der gesendeten Lifeline-Stehlanfragen positiv beantwortet wurde. Wenn der Dieb bereits deaktiviert wurde, aber einer seiner Lifeline-Buddies doch noch Tasks abgeben kann, wird der Dieb wieder aktiviert und die Endlosschleife aus Listing 3.1 erneut gestartet.

Sobald alle Worker deaktiviert sind, ist sichergestellt, dass sich keine weiteren Tasks im Umlauf befinden können. Dann greift das **finish**-Konstrukt von APGAS und die Ausführung der Worker ist beendet. Anschließend werden die Ergebnisse aller Worker, entsprechend dem Ablauf von dynamischen unabhängigen Tasks, mittels Reduktion zusammengeführt.

4. Randomisiertes Work Stealing und Terminierungserkennung

Die Terminierungserkennung in LL erfolgt dadurch, dass alle Worker deaktiviert sind, sobald keine Tasks mehr verfügbar sind, wodurch das **finish**-Konstrukt von APGAS greift. Demgegenüber werden Worker bei randomisiertem Work Stealing nicht deaktiviert. Dieses Lastenbalancierungsverfahren zeichnet sich dadurch aus, dass ein Worker ohne verbleibende Tasks beständig Stehlanfragen sendet. Die Terminierungserkennung bei RAN muss daher anders erfolgen.

Bereits 1989 schlug Mattern ein Terminierungserkennungsverfahren auf Basis von *Credits* vor [6]. Er ging von einem System aus, in dem mehrere Worker arbeiten, welche sich gegenseitig sogenannte Aktivierungsnachrichten senden können. Ein Worker kann aktiv oder passiv sein. Er kann jederzeit vom aktiven in den passiven Zustand wechseln, aber vom passiven in den aktiven Zustand nur, wenn er eine Aktivierungsnachricht erhält. Nur aktive Worker können eine Aktivierungsnachricht senden. Das von Mattern vorgeschlagene Terminierungserkennungsverfahren ist in der Lage zu erkennen, wann ein stabiler Zustand erreicht ist. Stabiler Zustand bedeutet, dass alle Worker passiv sind und keine Aktivierungsnachrichten mehr im System unterwegs sind.

Matterns Terminierungserkennung basiert auf der Idee einer Art virtueller Währung, den sogenannten Credits, die auf Worker und Aktivierungsnachrichten verteilt werden. Jeder aktive Worker und jede Aktivierungsnachricht besitzt Credits. Über die Aktivierungsnachrichten werden Credits von einem Worker zu einem anderen Worker transferiert. Passive Worker besitzen keine Credits mehr, beziehungsweise geben diese an eine übergeordnete Instanz, beispielsweise einen dedizierten Worker, ab. Das System startet mit einer festen Anzahl von Credits. Sobald alle Credits wieder an die übergeordnete Instanz zurückgegeben wurden, ist sichergestellt, dass sich keine weiteren aktiven Worker oder

Aktivierungsnachrichten mehr im System befinden, da sonst noch Credits fehlen würden.

Im Jahr 2021 verglichen Bosilca et al. verschiedene Verfahren zur Terminierungserkennung [1]: das oben erwähnte Verfahren, nachfolgend *CDA* (Credit Distribution Algorithm) genannt, die Verfahren namens *4C* und *EDOD* sowie eine an das Hochleistungsrechnen angepasste Variante von CDA. In [1] wurde festgestellt, dass die an das Hochleistungsrechnen angepasste Variante von CDA die wenigsten Kontrollnachrichten im System benötigte und damit am effizientesten ist. Daher wird diese Variante in einer an die GLB-Bibliothek angepassten Form als Terminierungserkennung in RAN verwendet.

Die angepasste Variante von CDA [1] funktioniert grundsätzlich sehr ähnlich wie die ursprüngliche Idee von Mattern [6]. Die Credits werden mithilfe einfacher Integer-Werte repräsentiert. Alle aktiven Worker starten mit einer vor der Ausführung festgelegten Anzahl von Credits, beispielsweise dem maximalen Integer-Wert. In RAN sind Worker als aktiv definiert, wenn sie mindestens einen Task besitzen, Worker ohne Tasks werden als passiv betrachtet. Während bei Mattern Credits mit Aktivierungsnachrichten gesendet werden, werden in RAN Credits gesendet, wenn ein Worker Tasks von einem anderen Worker stiehlt. Worker 0 wird als die übergeordnete Instanz betrachtet. Wenn er alle Credits besitzt, bedeutet dies, dass alle anderen Worker passiv sind und keine Tasks mehr zwischen Workern verschickt werden. Hat zusätzlich Worker 0 selbst keine Tasks mehr, befindet sich das System im stabilen Zustand. Worker 0 kann nun alle Worker deaktivieren und damit die Ausführung beenden.

Wie bei Mattern gelten auch in der angepassten Variante von CDA die Bedingungen, dass jeder aktive Worker mindestens einen Credit besitzen muss und dass ein Worker, der passiv wird, seine Credits an die übergeordnete Instanz abgibt. Hat ein Opfer bei einer Stehlanfrage nicht genügend Credits, um Credits an den Dieb zu senden, kann es sich Credits von der übergeordneten Instanz

gutschreiben lassen. Die übergeordnete Instanz verwaltet diese Gutschriften in einem einfachen Zähler.

Erhält ein Worker Credits von einem anderen Worker, werden diese addiert. Kommt es bei dieser Addition zu einem Overflow, werden die überschüssigen Credits an die übergeordnete Instanz zurückgegeben. Die übergeordnete Instanz verringert dann den Zähler der Gutschriften entsprechend.

In der angepassten Variante von CDA wurden zusätzlich weitere Festlegungen getroffen, die jedoch nicht vollständig in RAN übernommen wurden, siehe Abschnitt 5.3. Unter anderem wurde in [1] festgelegt, dass ein Worker bei der Abgabe von Credits an einen anderen Worker die Hälfte seiner Credits abgibt, wenn er noch mehr als **x** Credits besitzt. Der Wert **x** ist konstant und kann vor Programmstart festgelegt werden. Verfügt ein Worker über weniger als **x** Credits, wird nicht die Hälfte der Credits an einen Dieb abgegeben, sondern eine vor Programmstart festgelegte Anzahl von Credits. Zusätzlich wurde in [1] festgelegt, dass ein Worker, dessen Credits unter einen vordefinierten Wert fallen, proaktiv weitere Credits von der übergeordneten Instanz anfordert.

5. Implementierung

RAN basiert auf LL. Im gesamten Programm waren an vielen Stellen kleine Anpassungen nötig. Zum Beispiel wurde das Logging angepasst und erweitert, **finish**-Konstrukte konnten auf Grund der eigenen Terminierungserkennung entfernt werden und vieles mehr. Besonders im Ablauf der Endlosschleife jedes Workers wurden starke Veränderungen vorgenommen. Dieses Kapitel zeigt die wesentlichen Änderungen in RAN gegenüber LL. Zur Veranschaulichung ist in Listing 5.1 der Ablauf eines Workers in RAN als Pseudocode dargestellt.

```
1  activateAdditionalWorkersAndDistributeStartTasks()
2  while (true) {
3      if (shouldStop) {
4          return()
5      }
6      do {
7          bag.processTasks()
8          bag.processStealRequests()
9      } while (bag.notEmpty())
10     if (workerID != 0) {
11         giveAllCreditToSuperiorEntity()
12     }
13     if (localopt != 0) {
14         tryLocalOptimisedSteal()
15     }
16     tryRandomSteals()
17     if (workerID == 0 && bag.isEmpty() && hasInitialCredit()) {
18         stopAllOtherWorkers()
19         return()
20     }
21 }
```

Listing 5.1: Pseudocode für den Ablauf eines Workers bei RAN

5.1. Initiale Verteilung

Während in LL die Initialisierungsphase innerhalb der Endlosschleife stattfindet, erfolgt sie in RAN vor der Endlosschleife, siehe Zeile 1 in Listing 5.1. Die Funktionalität von **activateAdditionalWorkersAndDistributeStartTasks** besteht darin, die vorher festgelegte Anzahl von Workern zu aktivieren sowie die Tasks und Credits zu verteilen. In der Initialisierungsphase wird zwischen dynamischer und statischer Initialisierung unterschieden.
Bei der dynamischen Initialisierung startet zunächst nur Worker 0, der alle Tasks und damit auch alle Credits erhält, die zum Startzeitpunkt im Umlauf sind. Ausgehend von Worker 0 wird ein Binärbaum verwendet, um die anderen Worker zu aktivieren und die Tasks zu verteilen. Da sich zu Beginn der Programmausführung oftmals nur ein Start-Task in der Bag von Worker 0 befindet, wird dieser zunächst ausgeführt. Dieser erzeugt weitere Tasks, die in der Bag abgelegt werden. Anschließend entfernt Worker 0 die Hälfte der erzeugten Tasks aus seiner Bag und verteilt diese auf zwei neue Bags. Zusätzlich wird überprüft, ob genügend Credits vorhanden sind, um Tasks an zwei Worker zu senden. Ist dies nicht der Fall, leiht sich Worker 0 weitere Credits.
Anschließend startet Worker 0 zwei weitere Worker, Worker 1 und 2. In der Aktivierungsnachricht an diese Worker werden Credits und jeweils eine der beiden neu erzeugten Bags mit den darin enthaltenen Tasks mitgeschickt. Jeder so aktivierte Worker führt den eben beschriebenen Ablauf erneut aus und aktiviert gleichermaßen weitere Worker. Der Ablauf wiederholt sich, bis alle Worker gestartet wurden.
Bei der statischen Initialisierung wird auf jedem Place direkt der erste Worker gestartet. Dieser startet alle weiteren Worker des Places. Dabei werden keine Tasks, sondern nur Credits übergeben, da bei der statischen Initialisierung die Worker ihre Tasks zu Beginn selbst generieren.

Nachdem ein Worker einmalig die Initialisierungsphase durchlaufen hat, betritt er die Endlosschleife, siehe Zeile 2 bis 21 in Listing 5.1.

5.2. Randomisiertes Work Stealing

Sobald alle Tasks in der Bag eines Workers abgearbeitet sind und dabei mögliche Stehlanfragen bearbeitet wurden, siehe Zeile 6 bis 9 in Listing 5.1, gibt der Worker alle seine Credits ab, da er nun keine Tasks mehr besitzt, siehe Zeile 10 bis 12. Worker 0 gibt seine Credits nicht ab, da er als übergeordnete Instanz agiert und die Terminierung erkennt, wenn er alle im Umlauf befindlichen Credits besitzt.
Anschließend beginnt der Worker wie bei LL mit dem Stehlen. Ist die Erweiterung aktiviert, wird zunächst ein optimierter Stehlversuch gestartet, bei dem der Worker mit den meisten Tasks auf dem aktuellen Place das Opfer ist, siehe Zeile 13 bis 15. Schlägt dieser Versuch fehl, werden anschließend **x** zufällige Stehlversuche gestartet, siehe Zeile 16. Der Wert **x** kann vor der Programmausführung angegeben werden, wenn kein Wert angegeben wird, werden standardmäßig drei zufällige Stehlversuche gestartet. Schlagen auch diese fehl, legt sich der Worker bei RAN nicht schlafen. Aufgrund der Endlosschleife beginnt er erneut mit dem Work Stealing, bis ein Stehlversuch erfolgreich ist und der Dieb neue Tasks zur Bearbeitung erhält oder das Programm beendet wird.
Die grundlegende Implementierung des zufälligen Stehlens wurde von LL übernommen. Sie wurde jedoch erweitert, um die CDA-Terminierungserkennung umzusetzen. Sobald ein Opfer dem Dieb Tasks gibt, erhält der Dieb gleichzeitig Credits vom Opfer. Hat das Opfer nicht genügend Credits, leiht es sich weitere Credits von der übergeordneten Instanz. Standardmäßig startet das parallele Programmiersystem mit dem maximalen Integer-Wert als Anzahl der Credits, bei einer Ausleihe werden eine Milliarde zusätzliche Credits gutgeschrieben.

5.3. Terminierungserkennung

Wenn Worker 0 keine Tasks mehr in seiner Bag hat und gleichzeitig die Anzahl seiner Credits mit der Gesamtzahl der Credits im System übereinstimmt, siehe Zeile 17 in Listing 5.1, bedeutet dies, dass kein anderer Worker noch Credits und damit Tasks in seiner Bag hat. Worker 0 stoppt nun alle anderen Worker, siehe Zeile 3 bis 5 und Zeile 18. Anschließend beendet sich Worker 0 selbst, siehe Zeile 19. Nun ist die Abarbeitung der Tasks beendet und die Ergebnisse der einzelnen Worker können entsprechend dem Ablauf von DIT zusammengetragen werden. Dies ist in RAN genauso implementiert wie in LL.

Das verwendete Terminierungsverfahren ist eine an das Hochleistungsrechnen angepasste Variante von CDA, wie in Kapitel 4 beschrieben. Jedoch wurden nicht alle Vorgaben exakt übernommen. Das Opfer sendet dem Dieb immer die Hälfte seiner Credits, auch wenn es selbst nur noch wenige Credits hat. Dies wurde so entschieden, da das Opfer dem Dieb ebenfalls die Hälfte der Tasks aus seiner Bag übergibt. Opfer und Dieb besitzen demnach nach dem Diebstahl ungefähr gleich viele Tasks. Daher ist es sinnvoll, dass sie auch gleich viele Credits besitzen. Hätte das Opfer dem Dieb nur wenige Credits gesendet, um selbst Credits zu sparen, könnte dies schnell dazu führen, dass der Dieb weitere Credits ausleihen muss, wenn andere Worker eine Stehlanfrage an ihn stellen.

Des Weiteren werden Credits nicht proaktiv angefordert, wenn nur noch wenige Credits vorhanden sind, sondern erst dann, wenn sie tatsächlich benötigt werden. Dies wurde so umgesetzt, um unnötige Anfragen an Worker 0 zu vermeiden. Zudem werden bei der Ausleihe von Credits diese sofort, noch vor der Netzwerkkommunikation, bereitgestellt, sodass es zu keiner langen Verzögerung bei der Ausleihe kommt. Dazu schreibt sich ein Worker die zusätzlichen Credits selbst gut und informiert Worker 0 über das Netzwerk. Worker 0 kann dann seinen Zähler für die Gutschriften erhöhen und der Worker mit den zusätzlichen Credits kann direkt in seinem Ablauf fortfahren, ohne auf die Netzwerkkommunikation

zu warten. Dies ist möglich, da ein Worker, der zusätzliche Credits anfordert, selbst noch mindestens einen Credit besitzt. Auf diese Weise kann es nicht zu einer falschen, zu frühen Terminierungserkennung bei der Gutschrift von Credits kommen.

6. Experimente

6.1. Überblick

Um die beiden Lastenbalancierungsverfahren bezüglich ihrer Effizienz vergleichen zu können, wurden einige Messungen mit LL und RAN durchgeführt. Die Messungen erfolgten auf dem Linux-Cluster der Universität Kassel auf der Partition FB16. Diese enthält 12 Rechner, die mit je zwei Intel Xeon E5-2643 v4 Prozessoren mit jeweils sechs Kernen ausgestattet sind [16]. Somit besitzt die Partition insgesamt 144 Kerne, aufgeteilt auf 12 Rechner mit jeweils 12 Kernen. Jeder Rechner stellt einen Place mit genau 12 Workern dar. Die Messungen wurden mit 1, 2, 4, 6, 8, 10 und 12 aktiven Places durchgeführt.

Jede Messung wurde 20-mal wiederholt und der Durchschnitt gebildet, alle folgenden Angaben beziehen sich also immer auf den Durchschnitt und nie auf eine einzelne Messung. Zusätzlich wurde jeweils die Standardabweichung berechnet, die in den Diagrammen innerhalb dieses Kapitels als Fehlerindikator durch flache Rauten dargestellt ist. Die in den Diagrammen dargestellten Messwerte sind jeweils in Tabellenform im Anhang zu finden. Eine Datei mit allen einzelnen Messwerten befindet sich als Ergänzung in der digitalen Abgabe dieser Bachelorarbeit. Wenn unter den 20 Messwerten ein einzelner Messwert um mindestens 20% vom Mittelwert abwich, wurde diese Messung wiederholt. Von den insgesamt 2800 durchgeführten Messungen mussten insgesamt zwei Messungen wiederholt werden. Die abgewichenen Messungen gehen nicht mit in den Durchschnitt ein, können aber in der digitalen Abgabe als roter Wert eingesehen werden.

6.2. Benchmarks

Für die Messungen wurden vier Benchmarks verwendet. Alle Benchmarks waren bereits für LL implementiert und mussten für RAN nicht angepasst werden. Bei allen Benchmarks wurden die einstellbaren Parameter so gewählt, dass jede einzelne Messung, auch bei der Nutzung aller 12 Places, mindestens 30 Sekunden benötigte. Nachfolgend werden die vier Benchmarks vorgestellt:

- **UTS:** Der *Unbalanced Tree Search* Benchmark [2, 7] erzeugt einen unbalancierten zufälligen Baum und zählt die Anzahl der Knoten. Für jeden Knoten wird ein neuer Task erzeugt und die Anzahl seiner Kinder mit der kryptographischen Hashfunktion SHA-1 berechnet. Dadurch ist die Anzahl der Knoten vor der Ausführung nicht bekannt. Da der Baum nicht balanciert ist, ist die Lastenverteilung auf die einzelnen Worker ungleichmäßig, wodurch der Benchmark besonders für den Vergleich von Lastenbalancierungsverfahren geeignet ist. Wichtige Parameter des Benchmarks sind die Höhe des Baums **d**, die auf 17 eingestellt wurde, der Verzweigungsgrad **b**, der Cutoff-Wert **c**, sowie der Seed **s** für den Zufallsgenerator. Der Cutoff-Wert gibt an, ab welcher Tiefe im Baum für einen Kindknoten kein neuer Task mehr erzeugt werden soll, sondern die Berechnung innerhalb des aktuellen Tasks abgeschlossen wird. Der UTS-Benchmark verwendet die dynamische Initialisierung.

- **BC:** Beim *Betweenness Centrality* Benchmark [3] werden von einem generierten Graphen die kürzesten Wege zwischen allen Knotenpaaren berechnet. Die Betweenness Centrality eines Knotens gibt an, wie oft er Teil eines solchen kürzesten Weges ist. Der BC-Benchmark berechnet einen Vektor, der die Betweenness Centrality jedes Knotens enthält. Beim BC-Benchmark erfolgt die Initialisierung statisch. Jeder Worker kennt den Graphen und hat die Aufgabe, für einen Teil der Knotenpaare die kürzesten

Wege zu berechnen. Parameter sind **h** mit 18, wodurch der Graph 2^{18} Knoten hat, sowie der Seed **s**.

- **SYN:** Im *synthetischen* Benchmark [2, 11] ist die Ausführungszeit der Tasks ein wichtiger Parameter. Die Gesamtausführungszeit ergibt sich aus der Ausführungszeit der Tasks und dem *Overhead* des Laufzeitsystems. Der Overhead ist die Zeit, die das Laufzeitsystem benötigt, um die Tasks zu verwalten und die Lastenbalancierung durchzuführen. Da die Ausführungszeit der Tasks als Parameter angegeben wird, ist diese bei den verschiedenen Messungen immer gleich, nur der Overhead variiert, wodurch sich dieser Benchmark besonders gut für den Vergleich von Laufzeitsystemen eignet. Beim synthetischen Benchmark wird dementsprechend nicht die Gesamtausführungszeit, sondern der Overhead verglichen. Um diesen zu erhalten, muss die Ausführungszeit der Tasks von der Gesamtausführungszeit abgezogen werden.

  Der synthetische Benchmark erzeugt Tasks, die eine Dummy-Berechnung ausführen, über einen perfekten *w-ären* Baum. Für jeden Worker werden **m** Knoten und damit Tasks erzeugt. Die Laufzeiten der einzelnen Tasks variieren um den Faktor **v** und werden so gewählt, dass die vorgegebene Ausführungszeit der Tasks erreicht wird. Die Ausführungszeit der Tasks **t** wurde auf 100 Sekunden gesetzt.

- **FIB:** In diesem Benchmark [12] wird eine rekursive Berechnung auf Basis der Fibonacci-Folge durchgeführt. Es wird die **n**-te Fibonacci-Zahl berechnet, wobei **n** als Parameter angegeben werden kann. In dieser Bachelorarbeit wurde für **n** der Wert 61 gewählt. Ein weiterer Parameter ist die Angabe, ab welcher Fibonacci-Zahl die beiden rekursiven Berechnungen nicht mehr als neue Tasks ausgeführt werden sollen, sondern der aktuelle Worker die Berechnung übernimmt. Dieser sogenannte Cutoff-Wert **c** wurde auf 30 gesetzt.

In Tabelle 6.1 sind die Konfigurationen der Benchmarks dargestellt.

Benchmark	Parameter
UTS	$d = 17,\ s = 19,\ b = 4,\ c = 7$
BC	$h = 18,\ s = 2$
SYN	$t = 10^5$ms, dynamisch initialisiert, $m = 10^6$, $v = 20\%$
FIB	$n = 61,\ c = 30$

Tabelle 6.1.: Benchmark Konfigurationen

6.3. Vorversuche

Vor den eigentlichen Messungen wurden zwei Vorversuche durchgeführt. Der erste Vorversuch sollte ausschließen, dass die Ausleihe von Credits einen signifikanten Einfluss auf die Effizienz hat, da Worker 0 die Ausleihen verwaltet und dabei entsprechend über das Netzwerk kommuniziert wird. Um abschätzen zu können, wie oft eine Ausleihe aufgrund fehlender Credits notwendig ist, wurde der synthetische Benchmark mit der im Abschnitt 6.2 angegebenen Konfiguration auf 12 Places mit zusammen 144 Workern insgesamt 20-mal auf einer leicht modifizierten Version von RAN ausgeführt. Die Änderung bestand darin, dass die Gesamtzahl der ausgeführten Ausleihen aller 144 Worker ausgegeben wurde. **Localopt** wurde dabei auf 0 belassen. Über alle 20 Messungen hinweg ergab sich ein Durchschnitt von 4 Ausleihen pro Messung mit einer Standardabweichung von 2,68. Die geringste gemessene Anzahl an Ausleihen betrug 0, die höchste gemessene Anzahl 9. Pro Messung wurden dabei 180837 Stehlversuche gestartet, von denen 138563 erfolgreich waren (76,62%). Bei der in dieser Bachelorarbeit maximal verwendeten Anzahl von Workern hat die Ausleihe über Worker 0 somit einen vernachlässigbaren Einfluss.

Der zweite Vorversuch sollte herausfinden, ob die verwendete Verteilung der Tasks beim Programmstart über den Binärbaum Vorteile gegenüber einem

direkten Start mit randomisiertem Work Stealing bringt oder sogar nachteilig ist. Nachfolgend wird RAN mit der Anfangsverteilung über den Binärbaum weiterhin mit RAN abgekürzt. RAN ohne Anfangsverteilung wird als *RAN_NOTREE* bezeichnet. In den Diagrammen in Abbildung 6.1 und 6.2 werden die Messergebnisse der Laufzeiten von RAN_NOTREE relativ zu den Laufzeiten von RAN dargestellt. Die y-Achse der Diagramme stellt den relativen Unterschied in Prozent dar. Beispielsweise würde ein Wert von 5% in dem Diagramm bedeuten, dass RAN_NOTREE um 5% langsamer ist als RAN. Ein Wert von -5% würde bedeuten, dass RAN_NOTREE 5% schneller als RAN ist. Auf der x-Achse ist die Anzahl der Places abgetragen. Der Benchmark BC wurde in diesem Vorversuch ausgelassen, da aufgrund der statischen Initialisierung die Anfangsverteilung generell nicht über einen Binärbaum erfolgt.

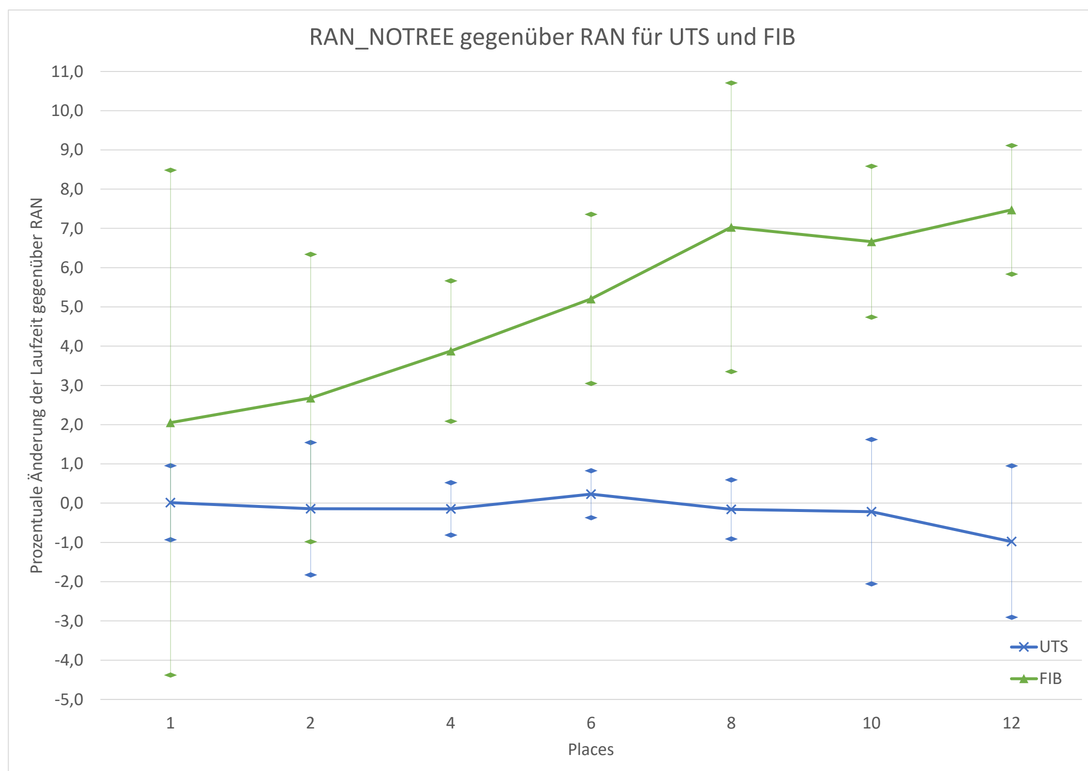

Abbildung 6.1.: Vergleich RAN_NOTREE gegenüber RAN bei UTS und FIB in Prozent

Abbildung 6.1 zeigt den Vergleich für die Benchmarks UTS und FIB, Abbildung 6.2 für SYN. Bei UTS verläuft der Graph ziemlich konstant bei etwa 0%, außer bei 12 Places. Dort ist RAN_NOTREE fast 1% schneller als RAN, allerdings ist auch die Standardabweichung bei 12 Places am höchsten. Bei FIB ist ein deutlicher Vorteil von RAN gegenüber RAN_NOTREE zu erkennen. Mit Zunahme der Anzahl an Places, mit einer kleinen Ausnahme bei zehn Places, erhöht sich die Laufzeit von RAN_NOTREE gegenüber RAN immer stärker. Bei 12 Places führt die Anfangsverteilung von RAN zu einer um 7,47% schnelleren Laufzeit.

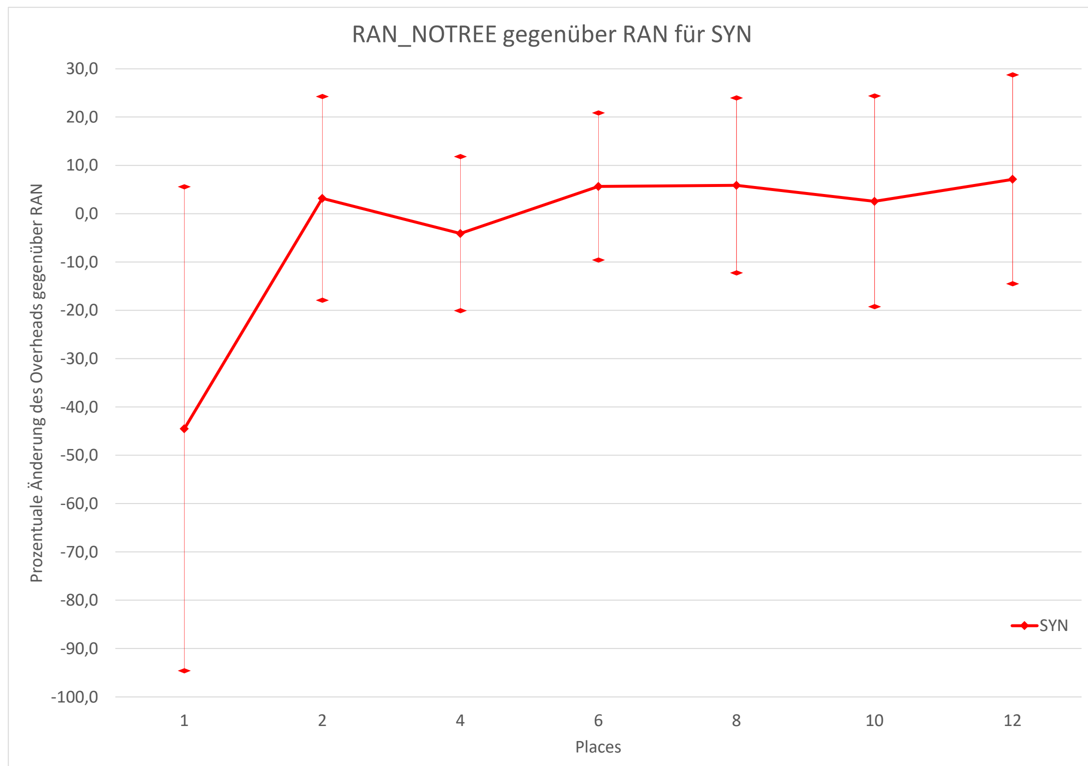

Abbildung 6.2.: Vergleich RAN_NOTREE gegenüber RAN bei SYN in Prozent

Bei SYN ist bis auf eine große Abweichung bei nur einem Place und einer kleinen Abweichung bei vier Places eine Laufzeitverbesserung von RAN gegenüber RAN_NOTREE zu erkennen. Mit steigender Anzahl der verwendeten Places nimmt die Laufzeitverbesserung zu und ist bei 12 Places mit 7,12% am größten.

Aus den Diagrammen lässt sich schlussfolgern, dass RAN gegenüber RAN_NOTREE in den meisten Fällen vorteilhaft ist, insbesondere mit zunehmender Anzahl von Places. Die Verschlechterung bei UTS ist minimal, während die Verbesserung bei FIB und SYN deutlich größer ist. Aus diesem Grund, sowie dem Vorhandensein einer Anfangsverteilung bei LL, wurden alle weiteren Messungen für die Implementierung des zufälligen Work Stealings mit RAN durchgeführt.

6.4. Ergebnisse

Die Messungen wurden mit vier verschiedenen Varianten durchgeführt, LL mit **localopt** 0 (*LL_LOC0*) und 2 (*LL_LOC2*), sowie RAN mit **localopt** 0 (*RAN_LOC0*) und 2 (*RAN_LOC2*). Bei SYN wird immer der Overhead verglichen, nicht die Gesamtlaufzeit. Da sowohl bei UTS als auch bei BC die Graphen bei einem absoluten Vergleich der Laufzeiten fast vollständig übereinander liegen, wurde für diese Benchmarks der relative Vergleich gewählt. Für FIB wurde ebenfalls der relative Vergleich gewählt, da die Unterschiede besser erkennbar sind als in der absoluten Darstellung. Der Referenzwert für relative Vergleiche ist LL_LOC0, da es sich hierbei um die ursprüngliche Variante ohne Erweiterungen handelt.
Wie in Abbildung 6.3 zu sehen, ist RAN_LOC0 im UTS-Benchmark vor allem ab acht Places langsamer als LL_LOC0, bei 12 Places ist die Verschlechterung der Laufzeit mit 1,75% am größten. Bei Verwendung der Optimierung ist der Unterschied deutlich geringer, aber immer noch vorhanden.

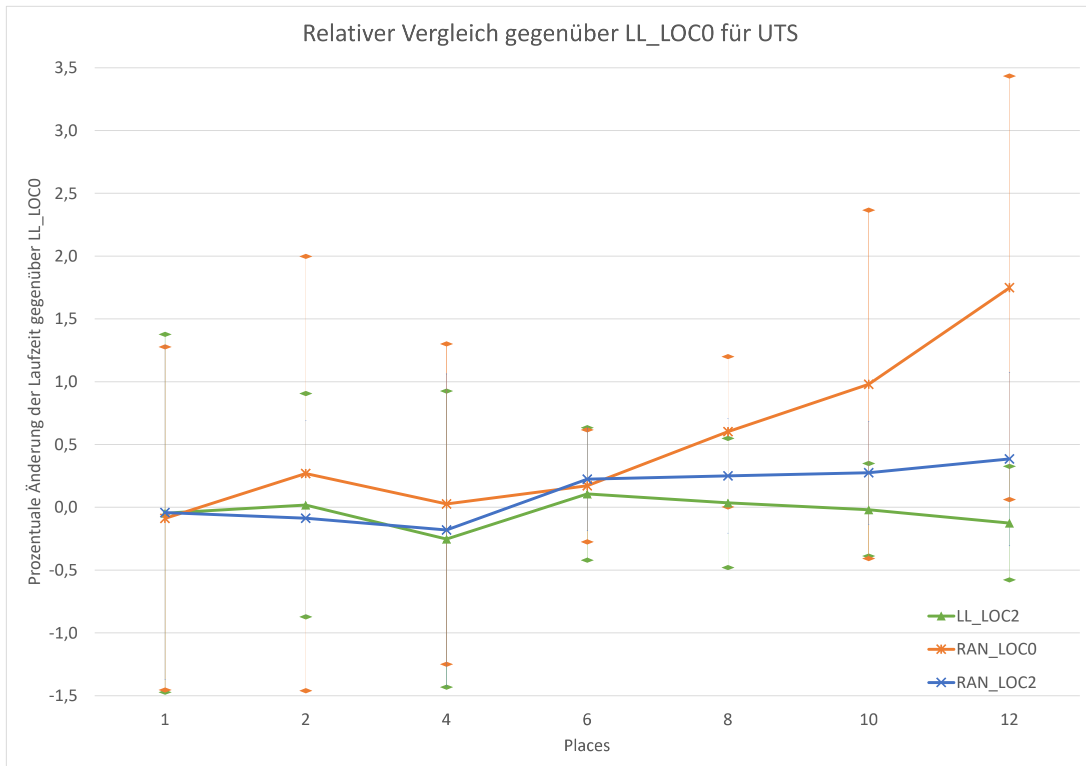

Abbildung 6.3.: Relativer Vergleich gegenüber LL_LOC0 bei UTS in Prozent

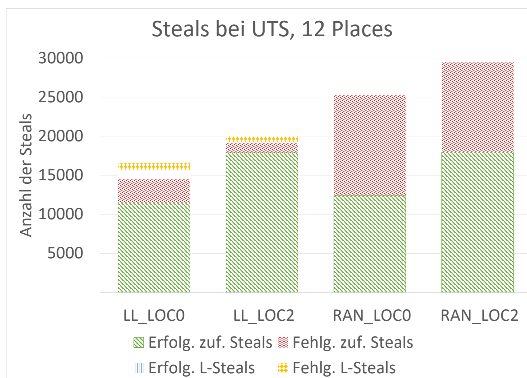

Abbildung 6.4.: Steals bei UTS

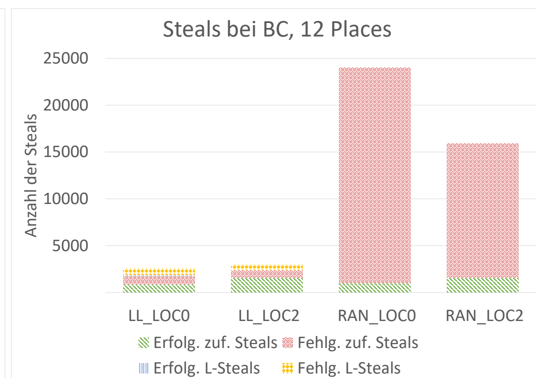

Abbildung 6.5.: Steals bei BC

Die Anzahl der Steals bei UTS ist beim randomisierten Work Stealing höher als beim Lifeline-Schema, siehe Abbildung 6.4. Insbesondere der Anteil der erfolgreichen zufälligen Steals ist bei RAN_LOC0 mit 49,10% deutlich geringer als bei LL_LOC0 mit 78,77%. Dies unterstreicht den Vorteil von LL gegenüber RAN bei UTS.

Beim BC-Benchmark ist die Anzahl der fehlgeschlagenen zufälligen Steals bei RAN um ein Vielfaches höher als bei LL, siehe Abbildung 6.5. Gleichzeitig ist die Anzahl der erfolgreichen zufälligen Steals in etwa gleich geblieben. Der Anteil der erfolgreichen zufälligen Steals beträgt somit bei RAN_LOC0 nur 4,15%, während bei LL_LOC0 50,40% aller zufälligen Steals erfolgreich sind.
Trotzdem ist bei dem Vergleich der Laufzeiten für RAN_LOC0 eine Laufzeitverbesserung von 1,05% gegenüber LL_LOC0 bei 12 Places zu erkennen, siehe Abbildung 6.6. Dies könnte auf die gute Verteilung durch die statische Initialisierung zurückzuführen sein, die das Stehlen erst am Ende der Programmausführung notwendig macht.

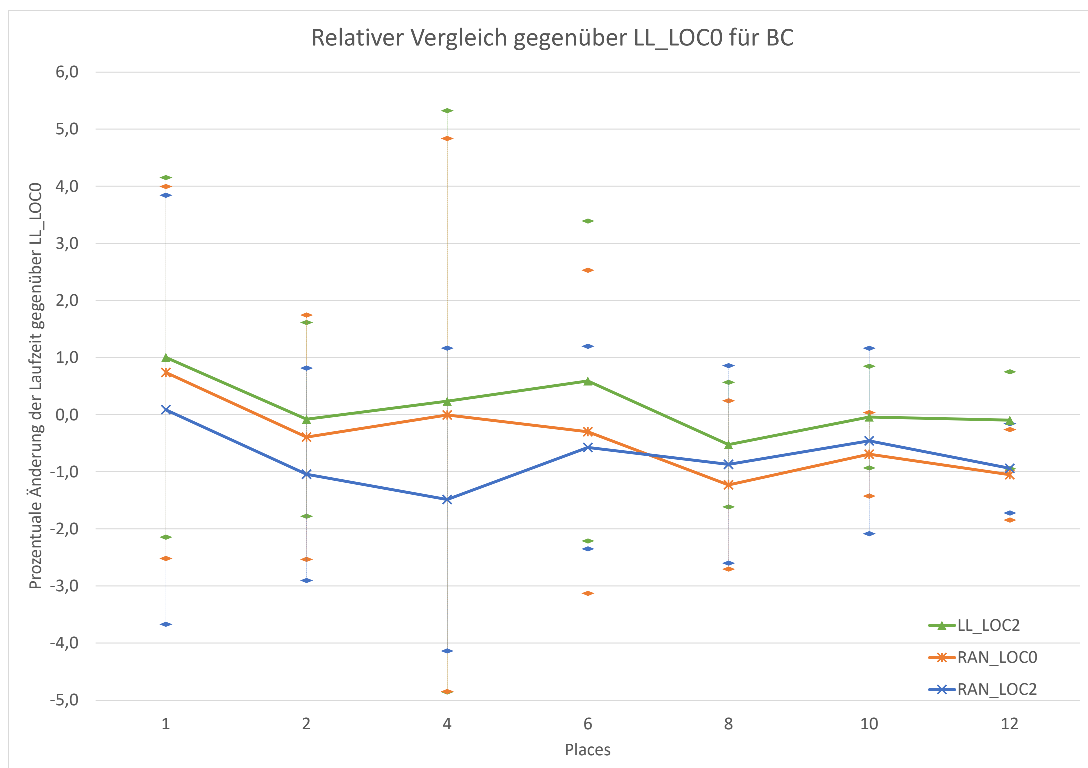

Abbildung 6.6.: Relativer Vergleich gegenüber LL_LOC0 bei BC in Prozent

Der synthetische Benchmark SYN zeigt einen deutlich größeren Unterschied zwischen den beiden Lastenbalancierungsverfahren als die bisherigen Benchmarks, siehe Abbildung 6.7. Dies ist auf die deutlich höhere Anzahl von Steals

zurückzuführen, siehe Abbildung 6.8. Die Anzahl der Steals spiegelt den Overhead des Laufzeitsystems wider, je mehr Steals, desto höher der Overhead. Außerdem wird beim synthetischen Benchmark nur der Overhead und nicht die Gesamtlaufzeit verglichen.

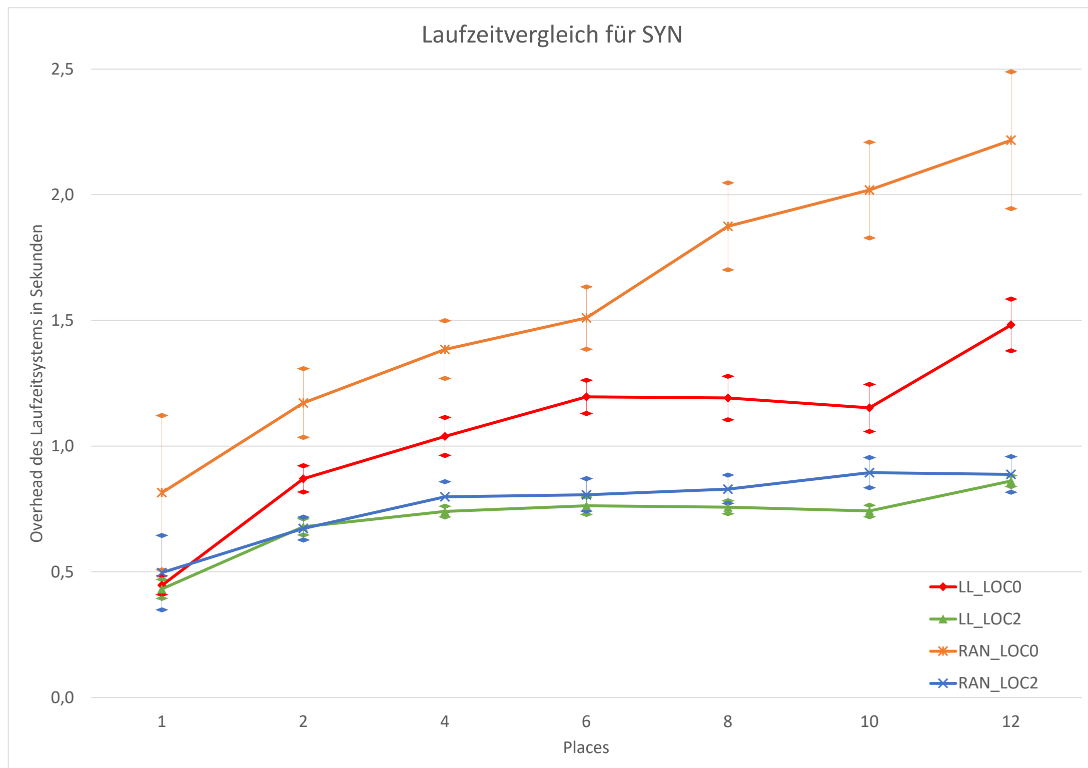

Abbildung 6.7.: Vergleich des Overheads bei SYN in Sekunden

Bei allen Implementierungen steigt der Overhead mit der Anzahl der Places, siehe Abbildung 6.7. Der Overhead des Laufzeitsystems von RAN_LOC0 bei 12 Places beträgt 2,22 Sekunden, während er bei LL_LOC0 nur 1,48 Sekunden beträgt. Damit ist der Overhead von RAN_LOC0 gegenüber LL_LOC0 bei 12 Places 49,57% höher. Der Anteil der erfolgreichen zufälligen Steals beträgt bei LL_LOC0 88,13%, bei RAN_LOC0 70,16%.

Gleichzeitig zeigt sich, dass die Erweiterung **localopt** 2 den Overhead deutlich reduziert, bei 12 Places und LL_LOC2 auf 0,86 Sekunden, bei RAN_LOC2 auf 0,89 Sekunden. Der Anteil der erfolgreichen zufälligen Steals beträgt bei

LL_LOC2 93,39% und bei RAN_LOC2 60,67%. Die Gesamtzahl der Steals ist bei RAN_LOC2 mit 39160 jedoch deutlich geringer als bei RAN_LOC0 mit 161519. Mit der aktiven Erweiterung ist der Overhead des randomisierten Work Stealings im Durchschnitt nur wenige Prozent höher als beim Lifeline Schema mit aktivierter Erweiterung.

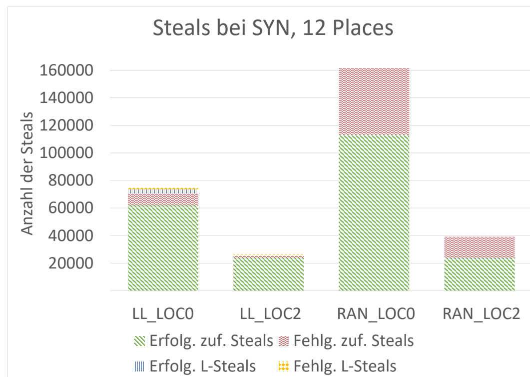

Abbildung 6.8.: Steals bei SYN

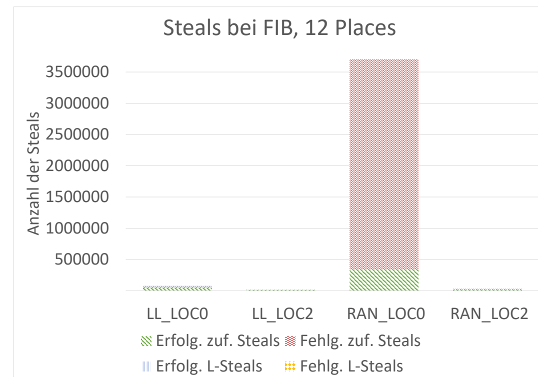

Abbildung 6.9.: Steals bei FIB

Der Benchmark FIB zeigt einen Extremfall, bei dem RAN_LOC0 bei 12 Places mit über 3,7 Millionen sehr viele Stehlversuche ausführt, siehe Abbildung 6.9. Das sind fast 47-mal so viele Stehlversuche wie bei LL_LOC0. Durch diese hohe Anzahl an Stehlversuchen, von denen nur 9,07% erfolgreich sind, ergibt sich für RAN_LOC0 bei 12 Places eine Laufzeit von 98,80 Sekunden gegenüber LL_LOC0 mit 47,38 Sekunden und 73,86% erfolgreichen zufälligen Stehlversuchen. Daraus ergibt sich bei 12 Places eine Verschlechterung der Laufzeit von RAN_LOC0 gegenüber LL_LOC0 um 108,53%, siehe Abbildung 6.10. Die geringe Standardabweichung verdeutlicht, dass es sich nicht um ungenaue Messungen handeln kann.

Die Verwendung von **localopt** 2 hat wiederum einen sehr großen Einfluss, sodass RAN_LOC2 und LL_LOC2 in etwa die gleiche Laufzeit aufweisen. Wie bei SYN reduziert die Erweiterung auch bei FIB die Anzahl der Stehlversuche stark, die Auswahl eines Workers mit vielen Tasks als nächstes Opfer scheint deutliche Vorteile zu bringen. Wurden bei RAN_LOC0 bei 12 Places noch über 3,7

Millionen Stehlversuche durchgeführt, sind es bei RAN_LOC2 nur noch 31258 Stehlversuche, von denen 39,80% erfolgreich sind. LL_LOC2 führt bei 12 Places 15914 zufällige Stehlversuche (zuzüglich 1772 Lifeline Steals) durch, von denen 78,77% erfolgreich sind.

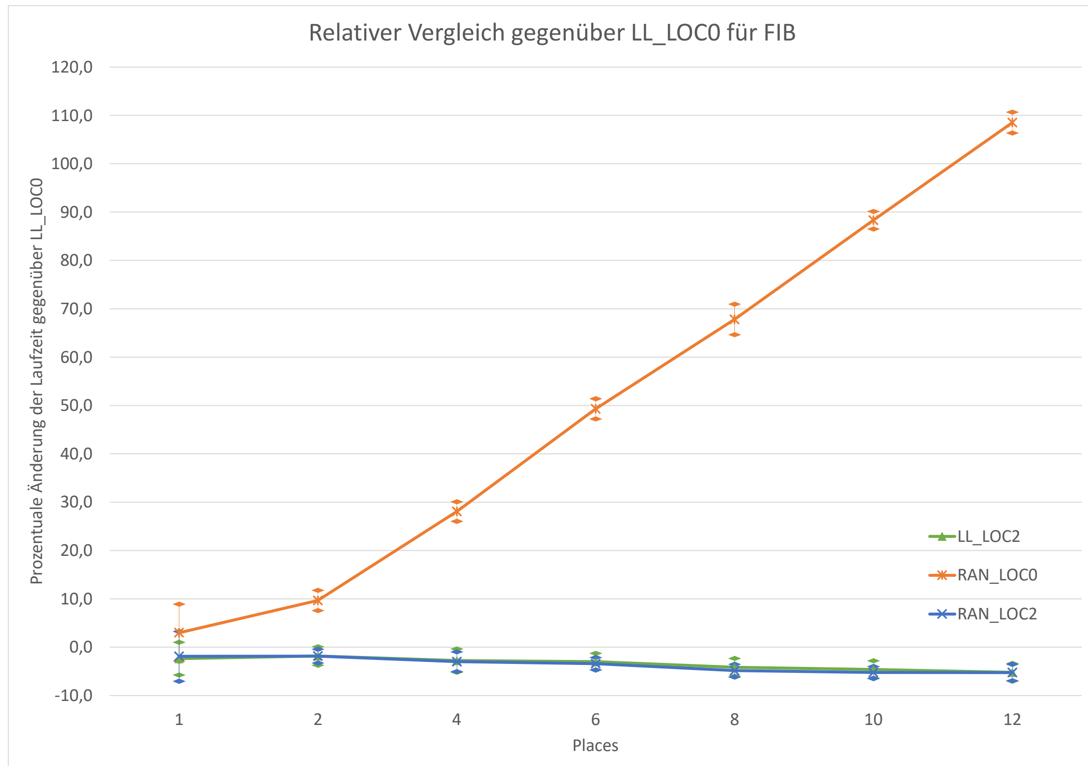

Abbildung 6.10.: Relativer Vergleich gegenüber LL_LOC0 bei FIB in Prozent

6.5. Auswertung

Beim einzigen verwendeten Benchmark mit statischer Initialisierung, BC, ist zwar nur ein geringer Unterschied zwischen LL_LOC0 und RAN_LOC0 bei gleichzeitig relativ hoher Standardabweichung zu erkennen, jedoch ist hier RAN mit oder ohne Erweiterung je nach Anzahl der verwendeten Places minimal besser als LL. Ab etwa acht verwendeten Places beträgt die Verbesserung im Durchschnitt 1%. Dies könnte damit zusammenhängen, dass bei BC die Tasks zu Beginn durch die statische Initialisierung bereits gut auf die Worker verteilt sind. Dadurch ist für die

meiste Zeit der Programmausführung kein Work Stealing notwendig. Erst kurz vor Ende der Ausführungszeit findet Work Stealing statt, wenn einzelne Worker früher alle Tasks abgearbeitet haben als andere Worker. Der zusätzliche Rechenaufwand durch das Lifeline-Schema hat in diesem Fall folglich mehr Einfluss als die erhöhte Anzahl fehlgeschlagener zufälliger Stehlversuche am Programmende. Allerdings ist der gemessene Unterschied, besonders im Zusammenhang mit der Standardabweichung, relativ gering, sodass ein eindeutiger Trend bei der Verwendung von mehr als 12 Places noch nicht sicher vorhergesagt werden kann.
Die drei anderen verwendeten Benchmarks mit dynamischer Initialisierung zeigen einen Vorteil des Lifeline-Schemas gegenüber dem randomisierten Work Stealing. In den meisten Fällen nimmt der Vorteil von LL gegenüber RAN mit der Anzahl der verwendeten Places zu. Bei 12 Places ist der Vorteil von LL_LOC0 gegenüber RAN_LOC0 beim UTS-Benchmark mit 1,75% eher gering, bei SYN mit 49,57% und FIB mit 108,53% jedoch sehr groß. Bei SYN und FIB resultieren diese großen Unterschiede aus einer hohen Anzahl von Stehlversuchen, bei FIB insbesondere von fehlgeschlagenen Stehlversuchen. Allerdings korreliert die Anzahl der Stehlversuche nicht immer mit der Laufzeit, wie ein Vergleich der Anzahl der Stehlversuche von BC, siehe Abbildung 6.5, mit der Laufzeit von BC, siehe Abbildung 6.6, zeigt.
Die Laufzeiten von RAN bei UTS, SYN und FIB können jedoch durch die Aktivierung der Erweiterung **localopt** 2 stark verbessert werden, sodass bei SYN und FIB sogar eine geringere Laufzeit als bei LL_LOC0 erreicht wird. Bei SYN ist der Overhead bei 12 Places von RAN_LOC2 nur um 0,03 Sekunden höher als bei LL_LOC2, bei FIB liegen LL_LOC2 und RAN_LOC2 trotz des sehr großen Unterschieds zwischen LL_LOC0 und RAN_LOC0 etwa gleichauf. Die Auswahl des Opfers hat dementsprechend einen großen Einfluss auf die Effizienz des Laufzeitsystems. Dieser ist so groß, dass er den Nachteil des randomisierten Work Stealings gegenüber dem Lifeline-Schema fast ausgleicht.

7. Schlussbemerkungen

Beim Vergleich der beiden in dieser Bachelorarbeit vorgestellten Lastenbalancierungsverfahren stellte sich heraus, dass das Lifeline-Schema, repräsentiert durch die Implementierung LL, dem randomisierten Work Stealing, repräsentiert durch die Implementierung RAN, in den meisten Fällen überlegen ist. Bei allen Benchmarks, UTS, BC, SYN und FIB, konnte gezeigt werden, dass das randomisierte Work Stealing deutlich mehr Stehlversuche durchführt als das Lifeline-Schema. Die Erfolgsrate der Stehlversuche ist niedriger als beim Lifeline-Schema. Bei den dynamischen Benchmarks, insbesondere bei SYN und FIB, führt dies zu einer deutlich langsameren Programmausführung. Im Gegensatz dazu ist das randomisierte Work Stealing bei BC trotz der hohen Anzahl fehlgeschlagener Stehlversuche etwas schneller. Dieser Unterschied ist jedoch gering und angesichts der Standardabweichung nicht sehr aussagekräftig.

Diese Bachelorarbeit bestätigt somit die von Saraswat [14] aufgestellte Behauptung, dass das Lifeline-Schema in UTS die gleiche oder eine bessere Performance erreichen kann als eine Implementierung mit rein zufälligem Stehlen und die Anzahl der Stehlversuche reduziert. Darüber hinaus scheint diese Behauptung nicht nur für UTS, sondern auch für viele weitere Fälle mit ungleichmäßig verteilten Tasks zutreffend zu sein. Die Erweiterung **localopt** 2 verbessert das randomisierte Work Stealing stark, führt aber dazu, dass die Auswahl der Opfer nicht mehr zufällig ist, ebenso ist diese Optimierung auch für das Lifeline-Schema verfügbar.

Im Fachgebiet Programmiersprachen/-methodik der Universität Kassel wurden bereits viele Varianten von GLB implementiert, beispielsweise eine Variante mit koordiniertem Work Stealing [17] und fehlertolerante Varianten [11]. Diese Bachelorarbeit baut auf der Implementierung von Hardenbicker [4], eine Variante mit Work Stealing für DIT, in welcher das Lifeline-Schema genutzt wird, auf.

Aus dieser Variante wurde die neue Implementierung RAN ohne Lifeline-Schema erstellt, die ausschließlich randomisiertes Work Stealing verwendet.

Ausgehend von dieser Arbeit gibt es weitere mögliche Varianten der Implementierung von GLB, die noch auf ihre Performance hin untersucht werden können. In dieser Bachelorarbeit wurde das Lifeline-Schema mit randomisiertem Work Stealing unter der Verwendung von kooperativem Work Stealing mit DIT verglichen. Ausstehend ist ein Vergleich bei der Verwendung von NFJ anstelle von DIT. Weiterhin fehlt ein Vergleich des Lifeline-Schemas mit dem randomisierten Work Stealing bei der Verwendung von koordiniertem Work Stealing, sowohl für DIT als auch für NFJ.

Zu beachten ist zudem, dass in dieser Arbeit die maximale Anzahl der verwendeten Places 12 und die Anzahl der Worker 144 betrug. Insbesondere in Anbetracht der Möglichkeit der Ausleihe von Credits über Worker 0 und dem damit verbundenen möglichen Flaschenhals bleibt abzuwarten, wie sich die Verfahren bei einer deutlich höheren Anzahl an Workern verhalten. Ab einer bestimmten Anzahl von Workern könnte es sinnvoll sein, dedizierte Worker für die Ausleihe von Credits zu verwenden.

Interessant ist auch die Fragestellung, wie sich das randomisierte Work Stealing gegenüber dem Lifeline-Schema in weiteren Benchmarks, beispielsweise N-Queens, schlägt. Insbesondere in Benchmarks, in denen die Tasks ähnlich wie bei BC bereits gut verteilt sind, könnte das randomisierte Work Stealing vorteilhaft sein.

Literatur

[1] G. Bosilca, A. Bouteiller, T. Herault, V. L. Fevre, Y. Robert und J. Dongarra. „Revisiting Credit Distribution Algorithms for Distributed Termination Detection“. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Juni 2021. doi: 10.1109/ipdpsw52791.2021.00095.

[2] C. Fohry, J. Posner und L. Reitz. „A Selective and Incremental Backup Scheme for Task Pools“. In: *2018 International Conference on High Performance Computing Simulation(HPCS)*. IEEE, Juli 2018. doi: 10.1109/hpcs.2018.00103.

[3] L. C. Freeman. „A Set of Measures of Centrality Based on Betweenness“. In: *Sociometry, Vol. 40, No. 1 (Mar., 1977), pp. 35-41* (1977). url: https://www.jstor.org/stable/3033543.

[4] K. Hardenbicker. „Eine lokalitätsoptimierte Lastenbalancierung für Task-basierte parallele Programmiersysteme in Rechenclustern“. In: *Bachelorarbeit*. 2022.

[5] J. Jeffers, J. Reinders und A. Sodani. „PGAS programming models“. In: *Intel Xeon Phi Processor High Performance Programming*. Elsevier, 2016, S. 369–382. doi: 10.1016/b978-0-12-809194-4.00016-8.

[6] F. Mattern. „Global quiescence detection based on credit distribution and recovery“. In: *Information Processing Letters* 30.4 (Feb. 1989), S. 195–200. doi: 10.1016/0020-0190(89)90212-3.

[7] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P.Sadayappan und C.-W. Tseng. „UTS: An Unbalanced Tree Search Benchmark“. In: *Springer LNCS 4382, Languages and Compilers for Parallel Computing* (2006).

[8] *PLM-APGAS 1.0-SNAPSHOT API.* URL: `https://www.plm.eecs.uni-kassel.de/apgas-doc/index.html`.

[9] J. POSNER. „Global Load Balancing and Intra-Node Synchronization with the Java Framework APGAS“. Magisterarb. Kassel: Department for Electric Engineering/Computer Science, Research Group Programming Languages/Methodologie, Jan. 2016.

[10] J. POSNER und C. FOHRY. „Cooperation vs. coordination for lifeline-based global load balancing in APGAS“. In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10.* ACM, Juni 2016. DOI: `10.1145/2931028.2931029`.

[11] J. POSNER, L. REITZ und C. FOHRY. „Task-Level Resilience: Checkpointing vs. Supervision“. In: *International Journal of Networking and Computing* 12.1 (2022), S. 47–72. DOI: `10.15803/ijnc.12.1_47`.

[12] L. REITZ, K. HARDENBICKER, T. WERNER und C. FOHRY. „Lifeline-based Load Balancing Schemes for Asynchronous Many-Task Runtimes in Clusters“. In: Research Group Programming Languages/Methodologies, University of Kassel. Nov. 2022.

[13] V. SARASWAT, G. ALMASI, G. BIKSHANDI, C. CASCAVAL, D. CUNNINGHAM, D. GROVE, S. KODALI, I. PESHANSKY und O. TARDIEU. *The Asynchronous Partitioned Global Address Space Model.* Techn. Ber. Toronto, Canada, Juni 2010.

[14] V. A. SARASWAT, P. KAMBADUR, S. KODALI, D. GROVE und S. KRISHNAMOORTHY. „Lifeline-based global load balancing“. In: *ACM SIGPLAN Notices* 46.8 (Feb. 2011), S. 201–212. DOI: `10.1145/2038037.1941582`.

[15] O. Tardieu. „The APGAS library: resilient parallel and distributed programming in Java 8“. In: *Proceedings of the ACM SIGPLAN Workshop on X10*. ACM, Juni 2015. doi: 10.1145/2771774.2771780.

[16] Universität-Kassel. *Hardware des Clusters*. url: https://www.uni-kassel.de/its/dienstleistungen/wissenschaftliche-datenverarbeitung/anleitung/hardware-und-software/hardware-des-clusters.

[17] T. Werner. „Anwendung einer SplitQueue Datenstruktur auf Work Stealing in Laufzeitsystemen taskbasierter paralleler Programmiersysthemen“. In: *Bachelorarbeit*. 2022.

[18] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat und M. Takeuchi. „GLB“. In: *Proceedings of the first workshop on Parallel programming for analytics applications*. ACM, Feb. 2014. doi: 10.1145/2567634.2567639.

A. Anhang

Digitale Abgabe

Inhalt von CD und Repository:

- Programmcode der Implementierungen
- Alle Ausgabedateien
- Tabelle mit allen Messwerten
- Digitale Fassung der Arbeit und der Diagramme
- Programme zum automatisierten Auslesen der Ausgabedateien

Wertetabellen

Die nachfolgenden Wertetabellen enthalten die wichtigsten Messergebnisse. Es handelt sich um Durchschnittswerte aus jeweils 20 Messungen. Laufzeiten in Sekunden und Prozentangaben wurden auf zwei Nachkommastellen gerundet. Die Anzahl der Steals ist auf ganze Zahlen gerundet. Bei der Angabe der Steals in Tabelle A.7 bis A.10 gibt der erste Wert die Anzahl der erfolgreichen Steals und der zweite Wert die Gesamtzahl aller Steals an. Eine umfangreichere Tabelle mit allen Messwerten findet sich in der digitalen Abgabe.

	Mittelwert	**Standardabweichung**
Anzahl der Ausleihen	4	2,68
Erfolgreiche Steals	138563 (76,62%)	28147 (20,31%)
Gesamtzahl aller Steals	180837	34206 (18,92%)

Tabelle A.1.: Vorversuch zur Anzahl der Ausleihen bei SYN (100s, 12 Places)

Tabelle A.2.: Laufzeiten für RAN_LOC0_NOTREE

Benchmark/Places	**Mittelwert [s]**	**Standardabweichung [s]**
UTS/1	391,31	1,44
UTS/2	196,13	0,67
UTS/4	98,31	0,27
UTS/6	65,93	0,24
UTS/8	49,66	0,16
UTS/10	40,00	0,26
UTS/12	33,45	0,17
BC/1	1708,07	37,64
BC/2	846,98	9,10
BC/4	424,71	2,30
BC/6	284,81	4,10
BC/8	212,69	2,33
BC/10	170,12	0,81
BC/12	141,95	0,78
SYN/1	100,45	0,06
SYN/2	101,21	0,11
SYN/4	101,33	0,10
SYN/6	101,59	0,11
SYN/8	101,98	0,17
SYN/10	102,07	0,26
SYN/12	102,38	0,22
FIB/1	561,75	21,32
FIB/2	300,74	7,78
FIB/4	180,54	2,17
FIB/6	143,47	1,45
FIB/8	125,38	2,02

FIB/10	113,33	1,18
FIB/12	106,18	0,70

Tabelle A.2.: Laufzeiten für RAN_LOC0_NOTREE

Tabelle A.3.: Laufzeiten für LL_LOC0

Benchmark/Places	**Mittelwert [s]**	**Standardabweichung [s]**
UTS/1	391,60	3,09
UTS/2	195,88	0,76
UTS/4	98,43	0,87
UTS/6	65,66	0,14
UTS/8	49,45	0,08
UTS/10	39,69	0,07
UTS/12	33,20	0,08
BC/1	1707,70	30,29
BC/2	856,31	6,21
BC/4	430,75	8,61
BC/6	285,55	2,84
BC/8	214,71	1,40
BC/10	171,41	0,58
BC/12	143,38	0,62
SYN/1	100,45	0,04
SYN/2	100,87	0,05
SYN/4	101,04	0,08
SYN/6	101,20	0,07
SYN/8	101,19	0,09
SYN/10	101,15	0,09
SYN/12	101,48	0,10

FIB/1	534,43	17,51
FIB/2	267,05	2,72
FIB/4	135,72	1,94
FIB/6	91,33	0,86
FIB/8	69,81	0,76
FIB/10	56,42	0,52
FIB/12	47,38	0,58

Tabelle A.3.: Laufzeiten für LL_LOC0

Tabelle A.4.: Laufzeiten für LL_LOC2

Benchmark/Places	**Mittelwert** [s]	**Standardabweichung** [s]
UTS/1	391,42	2,48
UTS/2	195,91	0,98
UTS/4	98,18	0,29
UTS/6	65,73	0,21
UTS/8	49,46	0,17
UTS/10	39,69	0,07
UTS/12	33,16	0,07
BC/1	1724,84	23,71
BC/2	855,61	8,31
BC/4	431,76	13,35
BC/6	287,24	5,19
BC/8	213,59	0,93
BC/10	171,34	0,95
BC/12	143,24	0,60
SYN/1	100,43	0,04
SYN/2	100,68	0,03

SYN/4	100,74	0,02
SYN/6	100,76	0,03
SYN/8	100,76	0,03
SYN/10	100,74	0,02
SYN/12	100,86	0,02
FIB/1	521,80	0,60
FIB/2	262,21	2,35
FIB/4	131,96	1,29
FIB/6	88,60	0,68
FIB/8	66,91	0,50
FIB/10	53,82	0,47
FIB/12	44,92	0,29

Tabelle A.4.: Laufzeiten für LL_LOC2

Tabelle A.5.: Laufzeiten für RAN_LOC0

Benchmark/Places	**Mittelwert [s]**	**Standardabweichung [s]**
UTS/1	391,26	2,25
UTS/2	196,40	2,64
UTS/4	98,45	0,38
UTS/6	65,78	0,16
UTS/8	49,74	0,21
UTS/10	40,08	0,48
UTS/12	33,78	0,48
BC/1	1720,32	25,51
BC/2	852,94	12,06
BC/4	430,73	12,25
BC/6	284,70	5,23

BC/8	212,07	1,74
BC/10	170,22	0,67
BC/12	141,87	0,51
SYN/1	100,82	0,31
SYN/2	101,17	0,14
SYN/4	101,38	0,11
SYN/6	101,51	0,12
SYN/8	101,87	0,17
SYN/10	102,02	0,19
SYN/12	102,22	0,27
FIB/1	550,46	14,52
FIB/2	292,89	3,14
FIB/4	173,81	1,02
FIB/6	136,37	1,56
FIB/8	117,14	2,42
FIB/10	106,25	0,94
FIB/12	98,80	0,97

Tabelle A.5.: Laufzeiten für RAN_LOC0

Tabelle A.6.: Laufzeiten für RAN_LOC2

Benchmark/Places	**Mittelwert [s]**	**Standardabweichung [s]**
UTS/1	391,44	2,10
UTS/2	195,71	0,76
UTS/4	98,25	0,35
UTS/6	65,81	0,13
UTS/8	49,57	0,14
UTS/10	39,80	0,09

UTS/12	33,33	0,15
BC/1	1709,16	33,89
BC/2	847,38	9,61
BC/4	424,35	2,77
BC/6	283,91	2,22
BC/8	212,84	2,29
BC/10	170,62	2,19
BC/12	142,03	0,50
SYN/1	100,50	0,15
SYN/2	100,67	0,05
SYN/4	100,80	0,06
SYN/6	100,81	0,06
SYN/8	100,83	0,06
SYN/10	100,89	0,06
SYN/12	100,89	0,07
FIB/1	524,32	9,78
FIB/2	262,14	1,06
FIB/4	131,65	0,81
FIB/6	88,21	0,31
FIB/8	66,42	0,16
FIB/10	53,46	0,17
FIB/12	44,90	0,24

Tabelle A.6.: Laufzeiten für RAN_LOC2

Implementierung	Zufällige Steals	Lifeline Steals
LL_LOC0	11447/14532 (78,77%)	1098/2030 (54,10%)
LL_LOC2	17898/19171 (93,36%)	110/775 (14,23%)
RAN_LOC0	12405/25264 (49,10%)	-
RAN_LOC2	17993/29447 (61,10%)	-

Tabelle A.7.: Anzahl der Steals bei UTS für 12 Places

Implementierung	Zufällige Steals	Lifeline Steals
LL_LOC0	887/1760 (50,40%)	137/841 (16,25%)
LL_LOC2	1583/2376 (66,63%)	23/642 (3,59%)
RAN_LOC0	997/24005 (4,15%)	-
RAN_LOC2	1590/15934 (9,98%)	-

Tabelle A.8.: Anzahl der Steals bei BC für 12 Places

Implementierung	Zufällige Steals	Lifeline Steals
LL_LOC0	62309/70703 (88,13%)	2945/4109 (71,68%)
LL_LOC2	23895/25585 (93,39%)	175/933 (18,83%)
RAN_LOC0	113319/161519 (70,16%)	-
RAN_LOC2	23760/39160 (60,67%)	-

Tabelle A.9.: Anzahl der Steals bei SYN für 12 Places

Implementierung	Zufällige Steals	Lifeline Steals
LL_LOC0	52257/70748 (73,86%)	6395/8267 (77,35%)
LL_LOC2	12535/15914 (78,77%)	725/1772 (40,93%)
RAN_LOC0	336105/3706711 (9,07%)	-
RAN_LOC2	12439/31258 (39,80%)	-

Tabelle A.10.: Anzahl der Steals bei FIB für 12 Places