Vergleich von zwei Work-Stealing-Verfahren für Nested Fork-Join Programme

Bachelorarbeit

Vorgelegt im Fachbereich 16 – Elektrotechnik/Informatik

der Universität Kassel

von Felix Nolte

Erstgutachterin:

Prof. Dr. Claudia Fohry

Zweitgutachter:

Prof. Dr. Gerd Stumme

Eingereicht am 26. Mai 2023 in Kassel

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Kassel, 26. Mai 2023

Felix Nolte

Inhaltsverzeichnis

1 Einleitung

Zahlreiche aktuelle Problemstellungen, vor allem im wissenschaftlichen Bereich, erfordern die Verarbeitung immer größerer Datenmengen. Das *High-Performance Computing* befasst sich mit der möglichst schnellen Verarbeitung dieser Daten. Die hierfür notwendige Rechenleistung wird durch Supercomputer bereitgestellt, die heutzutage zumeist als Cluster aus bis zu mehreren tausend einzelnen Rechnern bestehen, die man als Nodes bezeichnet.

Bei der Entwicklung von Software, die einen derartigen Grad der Parallelisierung unterstützen soll, entstehen besondere Herausforderungen, zu deren Bewältigung verschiedenste Technologien eingesetzt werden. Ein häufig verwendetes Paradigma ist die *taskbasierte* parallele Programmierung, bei der Programme in *Tasks* (dt. Aufgaben) aufgeteilt werden, die jeweils Teile der Berechnung realisieren. Es existieren verschiedene Modelle, nach denen solche Programme strukturiert werden können, wobei wir in dieser Bachelorarbeit ausschließlich das *Nested Fork-Join*-Modell (NFJ) betrachten, in dem Tasks zur Laufzeit weitere Tasks erzeugen und sich dadurch eine Ausführungsstruktur in Form eines Baumes ergibt.

Verarbeitet werden die Tasks parallel durch *Worker* (dt. Arbeiter), die auf Softwareebene üblicherweise durch Threads oder Prozesse repräsentiert werden. Die Verteilung der Tasks erfolgt zur Laufzeit, um trotz der unterschiedlichen Dauer

der Tasks und der dynamischen Erzeugung eine hohe Effizienz zu erreichen. Diese Umverteilung von Arbeit bezeichnet man als Lastenbalancierung.

Es existieren verschiedene Techniken zur Lastenbalancierung, die unterschiedliche Vor- und Nachteile haben. Der Fokus dieser Arbeit liegt auf dem *Workstealing*, bei dem Worker, die neue Arbeit benötigen, Tasks von anderen Workern stehlen. Der stehlende Worker wird hier *Thief* und sein Pendant *Victim* genannt.

Die spezifischen Aspekte der Lastenbalancierung, wie der Ablauf des Workstealings, die Auswahl der Victims und die Kommunikation zwischen Workern können sehr unterschiedlich umgesetzt werden. Diese genaue Umsetzung hat einen großen Einfluss auf die Effizienz der implementierten Programme.

Ziel dieser Bachelorarbeit ist der Vergleich zwischen zwei Verfahren zur Lastenbalancierung. Verglichen werden ein Verfahren, bei dem Victims zufallsbasiert ausgewählt werden, und eines, das die Zufallsauswahl um das *Lifeline*-Schema erweitert, sowie weitere Optimierungen. Dafür werden die Verfahren für das Programmiersystem NFJGLB implementiert, das verteilte Lastenbalancierung für NFJ-Programme verwendet. NFJGLB ist in Java geschrieben und basiert auf der APGAS-Bibliothek, die grundlegende Funktionen zur verteilten und parallelen Programmierung bereitstellt. Der Vergleich erfolgt durch Benchmark-Programme.

Im Vergleich zeigt sich, dass das Lifeline-Schema im Allgemeinen eine bessere Performance als die rein zufallsbasierte Auswahl bietet. Zudem sorgen die implementierten Optimierungen nicht immer für eine Verbesserung, sodass die genaue Auswahl und Einstellung des Verfahrens essenziell für eine Maximierung der Performance ist.

Kapitel 2 geht auf die theoretischen und praktischen Hintergründe ein. Kapitel 3 erklärt die verschiedenen Varianten. Die Implementierung auf Basis von NFJGLB wird in Kapitel 4 beschrieben. Kapitel 5 erläutert die eingesetzten Benchmarks

sowie die Ergebnisse des Vergleichs. Eine Zusammenfassung der Arbeit und ihrer Ergebnisse sowie der Ausblick auf weitere Entwicklungen folgen in Kapitel 6.

Relevante Arbeiten, die die Grundlage für diese Bachelorarbeit bildeten, werden im entsprechenden Kontext erwähnt.

2 Hintergrund

2.1 Nested Fork-Join

Wir betrachten in dieser Arbeit Programme, bei denen Tasks nach dem Nested Fork-Join-Modell strukturiert sind. NFJ ist ein bekanntes Modell für taskbasierte Programmierung und wird unter anderem von Cilk [1] eingesetzt.

Beginnend in einer einzelnen Root-Task erzeugen sogenannte *dynamische Tasks* zur Laufzeit rekursiv weitere Tasks, sodass eine Baumstruktur entsteht. Man bezeichnet die erzeugende Task als Parent-Task und die erzeugten Tasks als Child-Tasks. Zur Berechnung des Gesamtergebnisses senden Child-Tasks ihre Ergebnisse zu den Parent-Tasks, die dann die Ergebnisse aller der von ihnen erzeugten Tasks zusammenführen, sodass in der Root-Task das Gesamtergebnis entsteht.

2.2 Workstealing

Es existieren zwei verschiedene Strategien zur Lastenbalancierung: *Worksharing* und *Workstealing*, wobei diese Arbeit letztere behandelt. Sie unterscheiden sich im Umgang mit neu erzeugten Tasks.

Beim Worksharing werden die Tasks sofort auf Worker verteilt.

Im Gegensatz dazu findet Lastenbalancierung beim Workstealing statt, indem Worker, die Arbeit benötigen, diese von anderen Workern stehlen, indem sie Anfragen (auch als Random-Anfragen bezeichnet) stellen. Wenn eine neue Task erzeugt wird, so wird die aktuelle Task am Kopfende einer Queue abgelegt und als nächstes die Child-Task ausgeführt. Die Parent-Task kann dann von einem anderen Worker gestohlen werden, was man als *Continuation stealing* bezeichnet (im Gegensatz dazu wird beim *Child stealing* die Child-Task in der Queue abgelegt). Steal-Anfragen werden immer vom Ende der Queue bedient, sodass die hierarchisch höchsten Tasks gestohlen werden.

2.3 Worker

Abbildung 2.1 zeigt den Ablauf eines einzelnen Workers. Nach dem Start des Workers beginnt dieser mit "Tasks abarbeiten" und wird danach versuchen, Anfragen anderer Worker zu bewantworten ("Beantworte Workstealing-Anfragen"). Sind keine Tasks vorhanden, so werden die Anfragen abgelehnt ("Lehne Steal-Anfragen ab") und der Worker beginnt selbst mit dem Stehlen von Tasks ("Führe Workstealing durch"). Ist dies erfolgreich, so werden die gestohlenen Tasks weiter verarbeitet.

2.4 APGAS

APGAS ist eine Bibliothek für verteilte, parallele Programmierung in Java [2]. APGAS steht für "Asynchronous Partitioned Global Address Space" und ist ein Programmiermodell [3], das wiederum auf dem PGAS-Programmiermodell ("Partitioned Global Address Space") basiert. APGAS ist ein erweitertes Modell mit

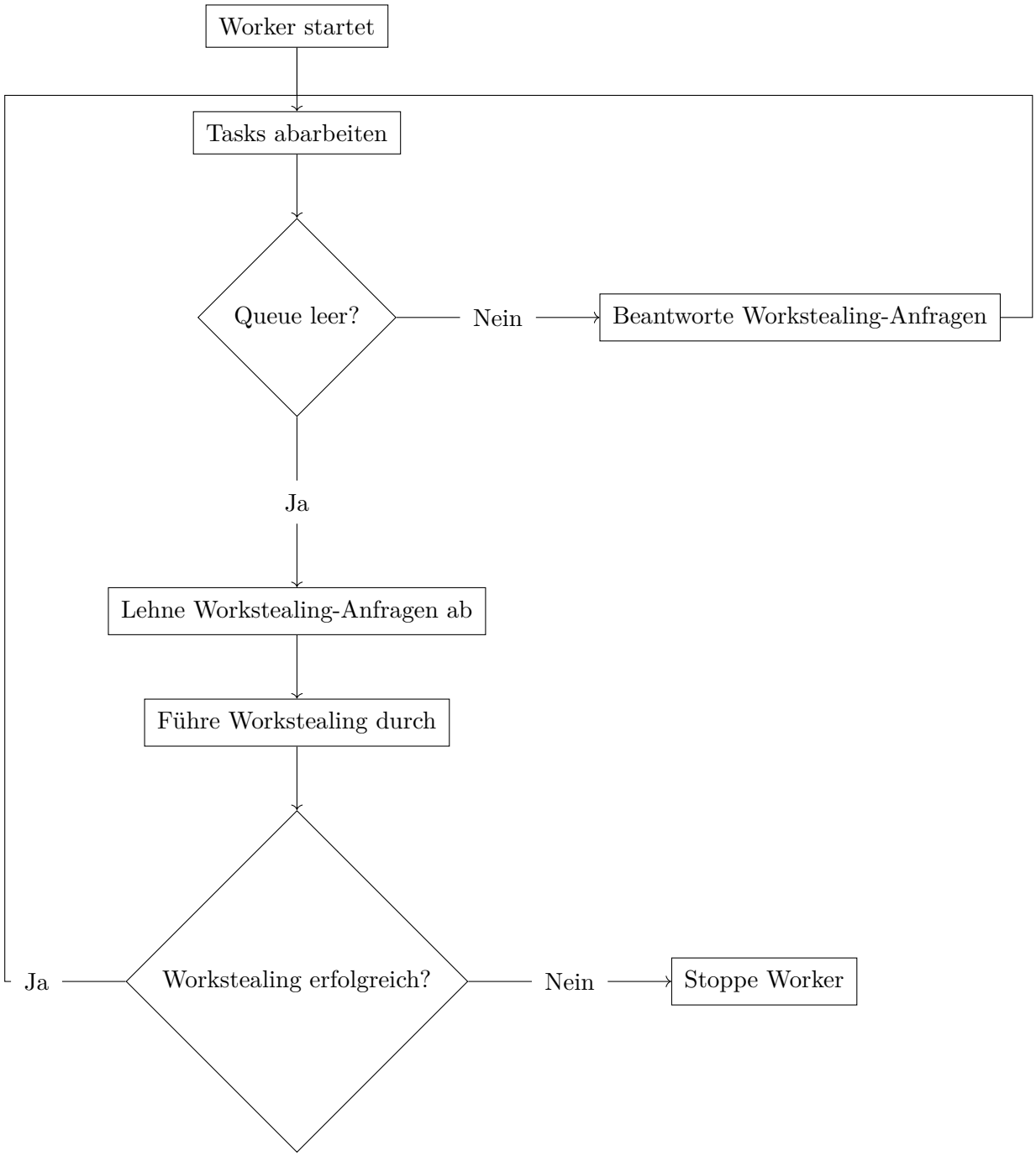

Abbildung 2.1: Worker-Ablaufdiagramm

Unterstützung für Asynchronizität.

Das Grundkonzept hinter (A)PGAS ist die Aufteilung des Speichers verteilter Systeme in einen globalen und lokale Bereiche, um Lokalitätseffekte ausnutzen zu können. Einen Prozess und seinen lokalen Speicherbereich bezeichnet man als *Place*, und die Places gemeinsam bilden ein *Cluster*. Auch in APGAS werden Tasks verwendet (die nicht dem NFJ-Modell entsprechen und hier als APGAS-Tasks bezeichnet werden). Diese APGAS-Tasks, die Teile von Berechnungen realisieren, werden auf den Places ausgeführt. Kommunikation findet dadurch statt, dass ein Place eine APGAS-Task an einen anderen Place sendet und auf eine Antwort wartet. In APGAS ist es durch asynchrone APGAS-Tasks möglich, mehrere APGAS-Tasks auf anderen Places auszuführen und die Antworten erst später zu erhalten.

Die APGAS-Bibliothek basiert auf der Programmiersprache X-10 [4] und macht das Programmiermodell für Java-Anwendungen nutzbar. In der Praxis bedeutet dies, dass sich der lokale Speicherbereich innerhalb eines Java-Prozesses befindet, während ein gemeinsamer globaler Bereich zwischen den Prozessen existiert. Dieser geteilte Speicherbereich wie auch die Kommunikation zwischen den Places werden mithilfe von Hazelcast [5], einer Plattform für verteilte Berechnungen und Speicher, implementiert.

Weitere Features von APGAS sind Resilienz und Elastizität: Es ist zur Laufzeit möglich, Places dem Cluster hinzuzufügen oder zu entfernen.

In der APGAS-Bibliothek wird ein Place durch die Klasse `Place` dargestellt und besitzt eine numerische ID. Die APGAS-Tasks werden durch Lambda-Funktionen beschrieben.

Zur Kommunikation zwischen den Places ist eine Serialisierung der Daten und APGAS-Tasks notwendig. Dementsprechend müssen sämtliche Objekte, die in AP-

GAS zwischen Places übertragen werden, das `Serializable`-Interface implementieren. Dies ist für primitive Datentypen bereits gegeben, eigene Klassen müssen jedoch mit `implements Serializable` markiert werden. Besonders gilt dies für die Lambda-Funktionen, die als Tasks verwendet werden. Bei diesen kann es sich um *Closures* handeln, die die in ihnen verwendeten Werte, die nicht aus dem eigenen Scope stammen, speichern. Dementsprechend müssen diese Werte ebenfalls serialisierbar sein. Das Interface `SerializableJob` stellt eine solche serialisierbare Lambda-Funktion dar.

APGAS stellt die Funktionen, mit denen Tasks gestartet und erwartet werden, als statische Methoden der `Constructs`-Klasse zur Verfügung. Fundamental sind die Funktionen `places()` und `here()`:

- `List<? extends Place> places()` gibt eine Liste der aktuell im Cluster befindlichen Places zurück. Diese können sich, wie oben beschrieben, zur Laufzeit ändern.
- `Place here()` ermöglicht den Zugriff auf den lokalen Place (von dem der Code ausgeführt wird).
- `Place place(int id)` ist eine Hilfsfunktion, die den Place mit der ID `id` zurückgibt.

Die anderen wichtigen Konstrukte, die in dieser Arbeit verwendet werden, sind:

- `void asyncAt(Place p, SerializableJob f)` Startet auf Place `p` eine neue, asynchrone Task `f`. Nachfolgender Code wird sofort ausgeführt, ohne dass auf das Ende der Task gewartet wird.

- `void finish(SerializableJob f)` Führt `f` aus und wartet auf alle Tasks, die von `f` gestartet werden.
- `void uncountedAsyncAt(Place p, SerializableJob f)` Startet auf Place `p` eine neue, asynchrone Task `f`. Das besondere an uncounted Tasks ist, dass sie nicht von einem `finish` erfasst werden. Zudem werden Exceptions, die von der Task geworfen werden, ignoriert.

2.5 NFJGLB

NFJGLB ist eine Bibliothek, die das Nested Fork-Join-Taskmodell implementiert und Workstealing zur Arbeitsverteilung und Lastenbalancierung nutzt. Die Abkürzung steht für "Nested Fork Join Global Load Balancing".

NFJGLB ist eine Weiterentwicklung der originalen GLB-Bibliothek ("Global Load Balancing"), die für die Programmiersprache X-10 zur parallelen, verteilten Programmierung [6] entwickelt wurde. Die originale Bibliothek verwendete innovativ das Lifeline-Schema, das die Verteilung von Tasks beim Workstealing regelt. Es wird detaillierter in Unterabschnitt 4.1.1 beschrieben.

Die GLB-Bibliothek wurde mithilfe von APGAS in Java implementiert [7]. NFJGLB ist eine Weiterentwicklung dieser Bibliothek, die anstelle des DIT-Taskmodells NFJ verwendet [8]. Es existieren verschiedene Varianten von NFJGLB, wobei die als Grundlage für diese Arbeit verwendete Variante jeden Thread als einzelnen Worker modelliert [9]. Sie verwendet das Lifeline-Schema zum Workstealing und unterstützt verschiedene Optimierungen (auf die in Abschnitt 3.3 eingegangen wird).

3 Algorithmen

Workstealing kann auf verschiedene Weisen implementiert werden. Das Diagramm 3.1 zeigt sämtliche Varianten, die in dieser Arbeit betrachtet werden.

3.1 Auswahl der Victims

Der Fokus dieser Arbeit liegt auf dem Vergleich von zwei Verfahren zur Auswahl von Victims.

Einerseits kann die Auswahl zufallsbasiert stattfinden (**Random-Verfahren**): Ein Worker wählt so lange zufällig ein Victim aus, bis der Steal-Versuch erfolgreich ist. Diese Methode ist die Grundlage für zahlreiche andere Programmiersysteme (wie beispielsweise Cilk[1]).

Eine andere Möglichkeit zur Auswahl stellt das **Lifeline-Schema** [10] dar. Dieses wurde auch in NFJGLB ursprünglich eingesetzt. Dazu werden jedem Worker permanent mehrere andere als Lifeline-Partner zugeordnet.

Ein Worker wird beim Workstealing zuerst mehrere zufallsbasierte Versuche unternehmen. Schlagen diese fehl, so werden dann seine Partner gleichzeitig nach Arbeit angefragt. Dies geschieht über Lifeline-Anfragen, die jedoch separat zu den Random-Anfragen verarbeitet werden.

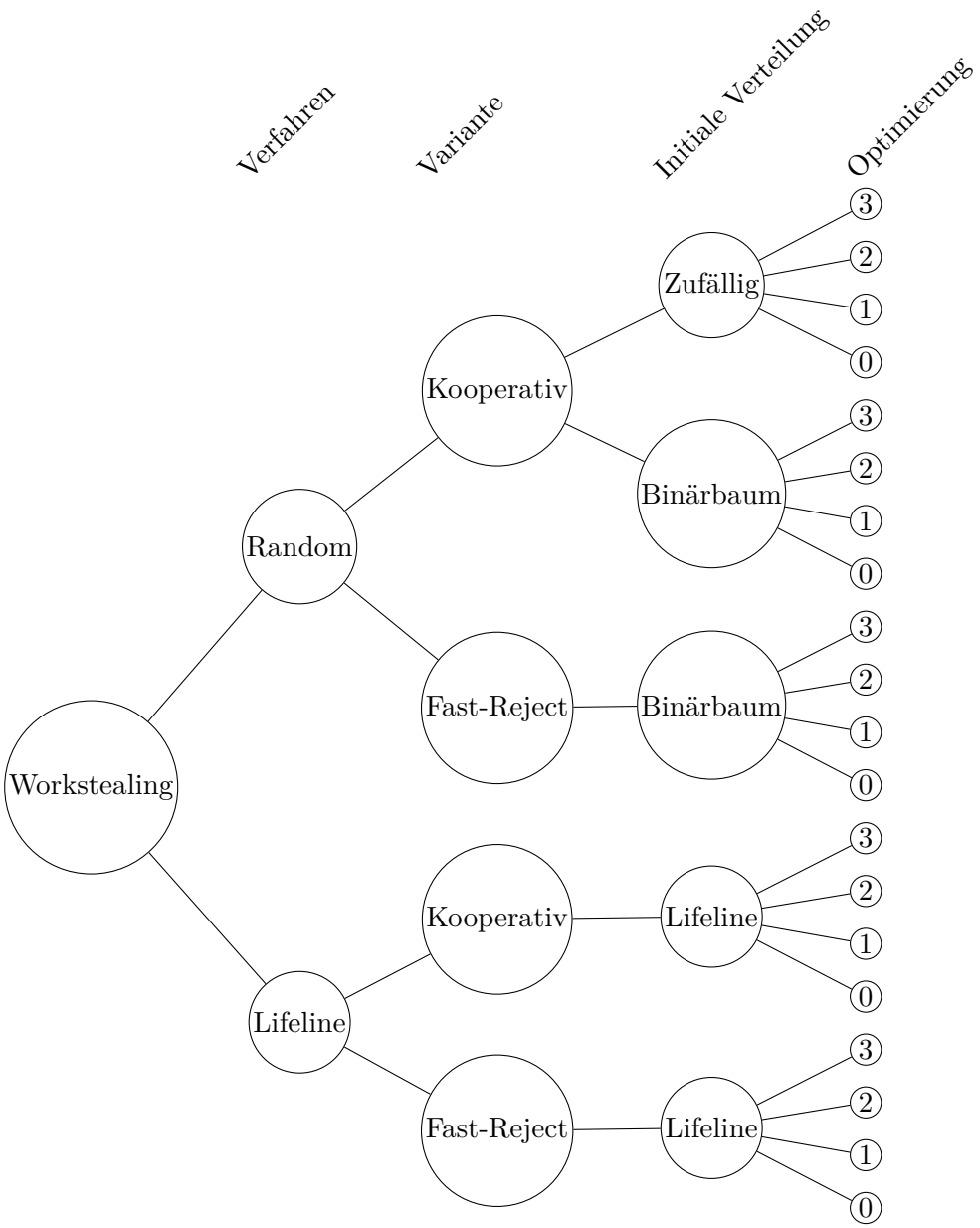

Abbildung 3.1: Alle untersuchten Workstealing-Varianten

Das Lifeline-Schema hat zum Ziel, die Anzahl der fehlschlagenden Anfragen zu minimieren, um eine bessere Performance als das Random-Verfahren zu bieten. Um dieses Ziel zu überprüfen, werden wir beide Verfahren vergleichen.

Da beide Methoden zufällig stehlen, können die im Folgenden beschriebenen Varianten auf beide angewendet werden.

3.2 Verarbeitung von Steal-Anfragen

Es gibt für zufallsbasiertes Workstealing verschiedene Möglichkeiten zur Implementierung der Kommunikation zwischen Thief und Victim.

Die Methode, nach der der Thief Anfragen an das Victim stellt, die von letzterem beantwortet oder abgelehnt werden, bezeichnet man als *kooperatives* Workstealing.

Hingegen entnimmt beim *koordinierten* Workstealing (wie es beispielsweise in Cilk eingesetzt wird) der Thief direkt der Queue des Victims Tasks.

NFJGLB verwendet kooperatives Workstealing (Variante **Kooperativ**), bei dem Anfragen, die von einem Victim nicht beantwortet werden können, da nicht genug Tasks vorhanden sind, zu einem späteren Zeitpunkt erneut betrachtet werden. Anfragen werden nur dann abgelehnt, wenn der Worker selbst zu stehlen beginnt oder beendet wird.

Wir vergleichen eine weitere kooperative Variante **Fast-Reject**, bei der Anfragen, die nicht beantwortet werden können, sofort abgelehnt werden. Dies erlaubt dem Thief, sofort die Suche nach einem weiteren Victim zu beginnen. Dadurch könnte diese Variante eventuell trotz häufigerer Fehlschläge eine höhere Performance bieten.

Es gibt auch eine NFJGLB-Variante, die koordiniertes Workstealing mithilfe einer *Splitqueue*-Datenstruktur implementiert [11]. Aus Gründen der Komplexität haben

wir uns jedoch nicht zur Umsetzung dieser Variante entschieden.

3.3 Lokalitätsoptimierungen

Bei zufälligen Steal-Versuchen können lokale Worker bevorzugt werden, um die Netzwerkkommunikation mit anderen Places zu verringern.

In dieser Arbeit betrachten wir die drei originalen Optimierungsstufen aus NF-JGLB [9], die wir hier mit (0, 2, 3) nummerieren, sowie eine Zwischenstufe (1), die eine bessere Analyse der Lokalitätseffekte ermöglichen soll.

0. Jeder Worker generiert beim eine zufällige Sequenz, in der die ID jedes anderen Workers vorkommt. Aus dieser Sequenz werden dann der Reihe nach Victims entnommen.

1. Ein Worker wählt beim ersten Versuch einen lokalen Worker aus und verwendet das Auswahlverfahren aus Stufe 0.

2. Ein Worker verwendet den lokalen Worker mit den meisten Tasks als Victim. Schlägt dieser Versuch fehl, so wird Stufe 0 verwendet.

3. Die Auswahl erfolgt wie in Stufe 2. Zusätzlich werden jedoch eingehende Steal-Anfragen an den lokalen Worker mit den meisten Tasks weitergeleitet.

3.4 Initiale Verteilung

In NFJ-Programmen existiert ein einzelner Root-Task, mit dem der erste Worker seine Arbeit beginnt. Die anderen Worker müssen jedoch ebenfalls gestartet werden

und können dafür initial Tasks erhalten.

Wir betrachten folgende Methoden:

- **Lifeline**: Bei der Verwendung des Lifeline-Verfahrens erfolgt die Verteilung automatisch, indem jeder Worker Arbeit an jene verteilt, die ihn als Lifeline-Partner haben, also in umgekehrter Richtung zu den normalen Steal-Anfragen. Diese Methode wird auch im originalen NFJGLB angewendet.
- **Random**: Für das zufällige Verfahren mit Fast-Reject kann diese Verteilung verwendet werden, bei der sämtliche Worker im Stealing-Zustand starten und selbst Tasks stehlen.
- **Binärbaum**: Für beide zufällige Varianten kann diese Verteilung verwendet werden, bei der jeder Worker sobald möglich Tasks an zwei weitere Worker abgibt.

4 Implementierung

Die Implementierung der in dieser Bachelorarbeit beschriebenen Algorithmen erfolgt auf Basis der NFJGLB-Software, die in Abschnitt 2.5 beschrieben wurde.

Um die verschiedenen Workstealing-Verfahren und -Varianten zu implementierten, mussten die relevanten Teile der Software angepasst werden.

An der grundlegenden Struktur der Worker hat sich jedoch nichts geändert. Worker werden über IDs identifiziert, die sowohl eine globale Komponente (Place) als auch eine lokale enthalten. Sie speichern ihre Tasks in einer Queue und besitzen weitere Datenstrukturen zur Speicherung von Steal-Anfragen und für die Umsetzung des NFJ-Modells.

Das Flussdiagramm Abbildung 2.1 demonstriert den Ablauf eines Workers. Für das Workstealing relevant sind hier die Schritte “Beantworte Workstealing-Anfragen”, “Lehne Workstealing-Anfragen ab” und “Führe Workstealing durch”.

“Lehne Workstealing-Anfragen ab” ist bei allen Verfahren identisch und lehnt sämtliche ausstehenden zufälligen Anfragen ab.

4.1 Auswahl der Victims

Zuerst werden wir auf die in Abschnitt 3.1 beschriebenen Verfahren eingehen und betrachten die originale Implementierung des Lifeline-Schemas und danach das rein zufallsbasierte Random-Verfahren.

Der in diesem Kapitel verwendete Pseudocode verwendet aus Übersichtlichkeit Äquivalente der in Abschnitt 2.4 beschriebenen APGAS-Funktionen, die von der ID eines Workers zu einem Place übersetzen und wie `asyncAtWorker` als Äquivalent von `asyncAt` benannt sind.

4.1.1 Lifeline-Schema

Die Implementierung des Lifeline-Schemas im originalen NFJGLB wird in Listing 4.1 durch Pseudocode dargestellt. Die Funktion `workstealing` übernimmt dabei den Schritt "Führe Workstealing durch" aus Abbildung 2.1. Zuerst wird das zufällige Workstealing versucht (Zeilen 2-5). Bleibt dieses erfolglos, so werden an alle Lifeline-Partner (Zeile 7) Anfragen gestellt (Zeilen 8-10) und der Worker danach gestoppt (Zeile 12). Er wird die Arbeit wieder aufnehmen, sobald eine der Anfragen beantwortet wurde.

Das zufällige Workstealing wird in der Funktion `randomWorkstealing` dargestellt. Es werden `W` Versuche unternommen. Jeder Versuch wählt ein Victim aus (Zeile 20) und sendet eine Anfrage (Zeilen 21-23) und wartet danach auf eine positive Antwort (Zeilen 25-27) und beginnt ansonsten den nächsten Versuch. Die Auswahl über die Funktion `chooseVictim` erfolgt durch die in Abschnitt 3.3 beschriebenen Optimierungsstufen.

Die Anfragen werden im Schritt "Beantworte Workstealing-Anfragen" aus

```
1  workstealing() {
2    randomSuccess := randomWorkstealing()
3    if (randomSuccess) {
4      return true
5    } else {
6      thief := this worker's ID
7      for (lifeline in lifelines) {
8        asyncAtWorker(lifeline, (worker) -> {
9          worker.handleLifelineRequest(thief)
10       })
11     }
12     stopWorker()
13     return false
14   }
15 }
16
17 randomWorkstealing() {
18   thief := this worker's ID
19   for (i = 0; i < W, i++) {
20     victim := chooseVictim(i)
21     asyncAtWorker(victim, (worker) -> {
22       worker.handleStealRequest(thief)
23     })
24
25     if (waitForAnswer()) {
26       return true
27     }
28   }
29   return false
30 }
```

Listing 4.1: Führe Lifeline-Workstealing durch

Abbildung 2.1 verarbeitet. Zuerst werden die zufälligen Anfragen, danach die Lifeline-Anfragen beantwortet. Listing 4.2 beschreibt diesen Schritt: Der Parameter `requestQueue` gibt an, aus welcher Queue die Anfragen entnommen werden (zufällige oder Lifeline). Solange Anfragen vorhanden sind (Zeile 2) und Tasks, um sie zu beantworten (Zeilen 3-4), werden Tasks vom Ende der Queue entfernt (Zeile 5) und die Anfrage beantwortet (Zeilen 8-10).

```
1  answerRequests(requestQueue) {
2    while (requestQueue is not empty &&
3            size of taskQueue > 1 &&
4            taskQueue can be split) {
5      loot := Split tasks from taskQueue
6      thief := requestQueue.next()
7
8      uncountedAyncAtWorker(thief, (worker) -> {
9        worker.receiveAnswer(victim, loot)
10     })
11   }
12 }
```

Listing 4.2: Beantworte Anfragen

4.1.2 Zufallsbasiertes Workstealing

Das Random-Verfahren für das Workstealing ist für diese Arbeit aus der Implementierung des Lifeline-Schemas entwickelt worden. Für den Schritt "Führe Workstealing durch" wird die Funktion `workstealing` in **??** dargestellt und ersetzt die Lifeline-Variante.

Die Funktion `randomWorkstealing` aus dem Lifeline-Schema wird wiederverwendet (sodass die Optimierungen identisch sind) und einfach in einer Endlosschleife

aufgerufen (Zeile 2) bis Erfolg eintritt (Zeile 4) oder der Worker von außen beendet wurde (Zeile 8).

```
1  workstealing() {
2    while (true) {
3      if (randomWorkstealing()) {
4        return true
5      }
6
7      if (Worker is stopped) {
8        return false
9      }
10   }
11 }
```

Listing 4.3: Führe Workstealing durch

4.2 Verarbeitung von Steal-Anfragen

Für die Verarbeitung der Random-Anfragen existiert, wie in Abschnitt 3.2 beschrieben, zusätzlich die **Fast-Reject**-Variante.

Sie implementiert den Schritt "Beantworte Workstealing-Anfragen". **??** zeigt den Ablauf: Anfragen werden der Queue entnommen, bis keine mehr vorhanden sind (Zeile 2). Wenn es möglich ist, sie zu beantworten (Zeilen 5-6), so werden dem Thief Tasks zurückgegeben (Zeilen 8-10). Andernfalls wird die Anfrage abgelehnt (Zeile 12).

Im Ergebnis erhält ein Thief schnell Feedback, wenn seine Anfrage erfolglos war.

```
1  answerRequests(requestQueue) {
2    while (requestQueue is not empty) {
3      thief := requestQueue.next()
```

```
4
5       if (size of taskQueue > 1 &&
6       taskQueue can be split) {
7         loot := Split tasks from taskQueue
8         uncountedAyncAtWorker(thief, (worker) -> {
9           worker.receiveAnswer(victim, loot)
10        })
11      } else {
12        reject(thief)
13      }
14    }
15  }
```

Listing 4.4: Semi-Koordinierte Variante

4.3 Lokalitätsoptimierungen

Die Implementierung der in Abschnitt 3.3 beschriebenen Optimierungen wurde aus NFJGLB übernommen. Die Optimierungsstufe 1 (zuerst zufällige Auswahl aus den lokalen Workern) wurde neu implementiert.

4.4 Initiale Verteilung

Wie in Abschnitt 3.4 beschrieben gibt es neue Möglichkeiten, um Tasks initial zu verteilen. Die Verteilung durch Lifelines wurde aus NFJGLB übernommen und die anderen Verfahren neu implementiert.

4.4.1 Änderungen für Random-Verteilung

Bei der Verwendung des Programmiersystems im Random-Workstealing-Modus entsteht ein Problem beim Beenden des Programms. Wenn das Lifeline-Schema verwendet wird, werden Worker, die keine Tasks mehr haben und ihre Lifeline-Partner anfragen, automatisch gestoppt. Bei der Verwendung des Random-Verfahrens geschieht dies jedoch nicht, stattdessen versuchen die Worker immer weiter, Tasks von anderen zu stehlen. Somit kann das Programm nicht korrekt beendet werden.

Um dies zu beheben, wurde Code eingefügt, der, sobald die Root-Task ein Ergebnis hat (und die Berechnung abgeschlossen ist), sämtliche Worker in den gestoppten Zustand versetzt. Beim zufälligen Workstealing wird dann der Zustand überprüft und das Workstealing abgebrochen, wenn der Worker gestoppt wurde, sodass das Programm beendet werden kann.

4.5 Laufzeitparameter

Für NFJGLB existieren verschiedene Parameter, mit denen das Verhalten der Software und die Auswahl der Workstealing-Variante gesteuert werden. Die wichtigsten und in dieser Bachelorarbeit verwendeten sind:

- **N** bzw. `glb.multiworker.n` Gibt die Anzahl von Tasks an, die ein Worker auf einmal verarbeitet.
- **W** bzw. `glb.multiworker.w` Gibt die Anzahl von Versuchen beim zufallsbasierten Workstealing an.
- **Steal tasks** bzw. `glb.multiworker.steal.numtasks` Gibt an, wie viele Tasks

bei einem Workstealing-Versuch gestohlen werden

- **Local optimization** bzw. `glb.multiworker.localopt` Gibt die Stufe der Lokalitätsoptimierung an (siehe Abschnitt 3.3)
- `glb.workstealing.strategy` Legt fest, welches Workstealing-Verfahren (`random` oder `lifeline`) verwendet werden soll.
- `glb.workstealing.random.strategy` Legt fest, wie das zufallsbasierte Workstealing arbeiten soll (`cooperative` oder `fast-reject`).
- `glb.workstealing.random.distribution.strategy` Legt die initiale Verteilung fest, sofern nicht Lifeline-Workstealing verwendet wird (`binary` für die Binärbaum-Strategie, `noop` für pures Workstealing)

5 Vergleich

Dieses Kapitel beschreibt den Vergleich der in dieser Arbeit beschriebenen Verfahren, vor allem hinsichtlich ihrer Laufzeitperformance.

5.1 Benchmarks

Der Vergleich der verschiedenen Verfahren erfolgt mithilfe von Benchmarks. Sie sind Teil der NFJGLB-Software-Suite und wurden bereits vorher zum Benchmarking verschiedener Modifikationen eingesetzt. An den Benchmarks selbst erfolgten keine Veränderungen.

Insgesamt werden drei Programme zur Evaluierung der Performance eingesetzt.

- **Synthetic**: Die Benchmark erwartet eine Zeitangabe g, eine Taskanzahl t und eine Varianz u (in Prozent) als Parameter. Pro Worker werden m Tasks erzeugt, die jeweils einen gleichen Anteil der Zeit g mit einer zufälligen Varianz von u dauern. Insgesamt dauert dann die reine Berechnung so lange wie die Zeitangabe g, sodass die Differenz zur Gesamtdauer der Benchmark durch das Laufzeitsystem verursacht wird.
- **UTS** (Unbalanced Tree Search): Die Benchmark generiert einen zufälligen, unbalancierten Baum der Tiefe t. Weitere Parameter sind der Verzweigungs-

faktor b und eine Wahrscheinlichkeitsverteilung (binomial oder geometrisch), die die Form des Baumes bestimmt. Ein weiterer Parameter *cutoff* gibt an, ab welcher verbleibenden Rekursionshöhe keine neuen Tasks mehr erstellt werden.

- **Fib** (Fibonacci): Die Benchmark berechnet die n-te Fibonaccizahl rekursiv, wobei für jeden Schritt neue Tasks erzeugt werden. Auch hier existiert ein *cutoff*-Parameter, sodass ab einer bestimmten Rekursionshöhe das Ergebnis einer Task direkt berechnet wird.

5.2 Ausführung

Die Ausführung der Benchmarks erfolgte auf dem ITS-Cluster der Universität Kassel. Als Java-Version wurde 19.0.2 eingesetzt. Verwendet wurde die **public4**-Partition, die aus 40 Nodes mit jeweils 24 CPU-Kernen (Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz) und je 128 GB RAM besteht. Es war entsprechend möglich, bis zu 24 Worker pro Node zu verwenden.

Um sowohl den Vergleich zwischen der Ausführung auf nur einem Worker als auch eine repräsentative Dauer der Benchmarks zu gewährleisten, haben wir uns entschieden, maximal 192 Worker (entsprechend 8 Nodes) zu verwenden. Die Parameter für die oben beschriebenen Benchmarks wurden entsprechend ausgewählt, um auch mit 192 Workern eine höhere Dauer als 30 Sekunden zu erreichen.

Tabelle 5.1 zeigt die verwendeten Parameter.

Zudem erfolgte noch die Festlegung der NFJGLB-Parameter (beschrieben in Abschnitt 4.5). Hier war vor allem die Anpassung des Parameters `glb.multiworker.n`, der die Anzahl der in einem Schritt des Workers zu verarbeitenden Tasks angibt,

Benchmark	**Verwendete Parameter**
Synthetic	$g = 100000$, $t = 1000000$ms, $u = 20$
UTS	$d = 17$, $b = 4$, *cutoff* $= 3$, Geometrische Verteilung
Fib	$n = 60$, *cutoff* $= 19$

Tabelle 5.1: Verwendete Parameter

notwendig. Die Werte (siehe Tabelle 5.2) wurden auf Basis der in [9] verwendeten ausgewählt und angepasst, wenn sich ein modifizierter Wert als effizienter erwies.

Benchmark	`glb.multiworker.n`
Synthetic	1
UTS	511
Fib	4

Tabelle 5.2: Verwendeter Wert für `glb.multiworker.n`

Für den Parameter `glb.multiworker.steal.numtasks`, der die Anzahl der gestohlenen Tasks pro Versuch bestimmt, wurde wie in vergangenen Benchmarks der Wert 1 gewählt.

5.3 Ergebnisse

Aus Zeitgründen wurden 7 Iterationen durchgeführt. Von den insgesamt 3780 Messungen wurden 3747 erfolgreich abgeschlossen, wobei bei 33 wiederholt Fehler auftraten, die jedoch keiner bestimmten Systematik folgen und durch Nichtterminierung der Benchmarks verursacht wurden. Die entsprechenden Messwerte sind nicht enthalten.

Aus den Messwerten wurden jeweils Durchschnitt, Minimum, Maximum, Speedup-Faktor und die Standardabweichung berechnet. Die gekürzten Resultate befinden sich in Tabellenform im Anhang.

Die Varianten werden nach dem Schema **Verfahren/Variante Optimierungsstufe mit initialer Verteilung** (also z.B. **Lifeline/Kooperativ Opt 3 mit Lifeline-Verteilung**) benannt.

Da drei verschiedene Benchmarks verwendet wurden, zeigen die Diagramme in diesem Kapitel, sofern nicht anders angegeben, Durchschnittswerte und durchschnittliche Standardabweichungen der relativen Laufzeiten über die drei Benchmarks hinweg an, um trotz der zahlreichen Messwerte übersichtlich zu sein. Weichen die Ergebnisse einer Variante für eine einzelne Benchmark deutlich ab, so wird dies im Text erwähnt.

Das Hauptziel dieser Arbeit liegt im Vergleich zwischen dem Lifeline-Schema und dem Random-Verfahren, wobei von beiden Verfahren verschiedene Varianten betrachtet werden. Insofern wollen wir jeweils die beste Variante beider Verfahren vergleichen, um das insgesamt schnellste Verfahren zur Lastenbalancierung zu finden.

Da wir vor allem an der Skalierbarkeit der Varianten interessiert sind, wird beim Vergleich der Fokus auf den Messwerten mit hohen Anzahlen von Workern liegen.

5.3.1 Lifeline-Schema

Zuerst betrachten wir, welche Verbesserungen für das Lifeline-Schema durch Auswahl von Varianten und Optimierungen erreicht werden konnten.

Abbildung 5.1 zeigt die Auswirkungen der Optimierungsstufen bei allen Benchmarks, wenn die kooperative Variante verwendet wird. Dabei zeigt die y-Achse

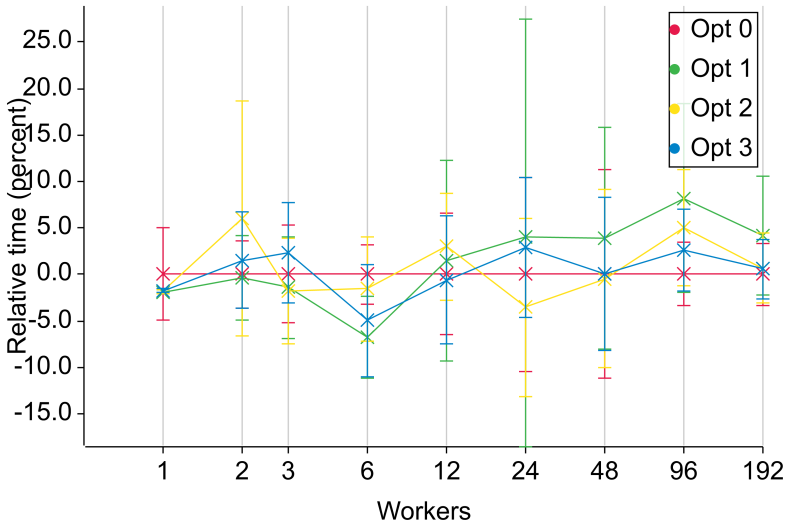

Abbildung 5.1: Vergleich von Optimierungsstufen mit **Lifeline/Kooperativ** (in Prozent)

jeweils die durchschnittliche Abweichung von der Dauer für die Optimierungsstufe 0, während die x-Achse die Anzahl der Worker angibt.

Für die Fast-Reject-Variante zeigt Abbildung 5.2 die Auswirkungen der Optimierungsstufen.

Bei der Interpretation der Ergebnisse für Lokalitätsoptimierungen ist zu bedenken, dass ein Place bis zu 24 Worker besitzen kann, sodass für die niedrigeren Worker-Anzahlen die Optimierungen nur Overhead hinzufügen. Es ist möglich, dass bei höheren Anzahlen von Places im Vergleich zu den hier maximal verwendeten 8 die Optimierungen durch Skalierungseffekte deutliche Verbesserungen erzeugen könnten.

Überraschenderweise und trotz der hohen Varianz der Ergebnisse lässt sich bei keiner der beiden Varianten eine eindeutige Verbesserung durch die Lokalitätsoptimierungen feststellen, die durchschnittlich besser ist als die unoptimierte Variante. Tatsächlich sind für höhere Workeranzahlen (48-192) sämtliche Optimierungsstu-

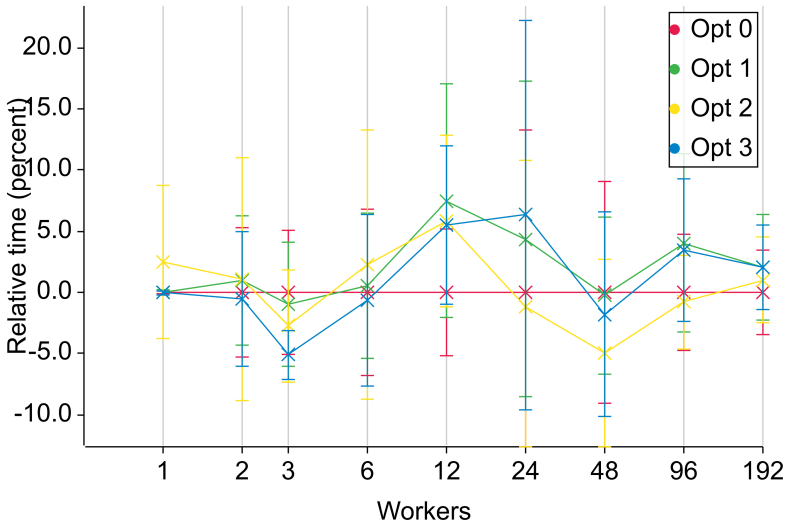

Abbildung 5.2: Vergleich von Optimierungsstufen mit **Lifeline/Fast-Reject** (in Prozent)

fen langsamer als Stufe 0, sogar um bis zu 9 % (Stufe 1 bei 96 Workern mit der kooperativen Variante).

Bei der **Fib**-Benchmark ist für beide Varianten die Optimierungsstufe 0 am schnellsten. Für **Synthetic** liegt für Fast-Reject knapp die Optimierungsstufe 3 vorne, für die kooperative Variante ist wieder 0 am schnellsten. Bei **UTS** liegen für die kooperative Variante knapp die Optimierungen 2 und 3 vorne, bei Fast-Reject jedoch nur Stufe 2. Trotzdem ist Optimierungsstufe 0 im Durchschnitt über alle Benchmarks am schnellsten.

Da keine eindeutige Verbesserung gefunden werden kann, wird der Vergleich der Lifeline-Varianten mit Optimierungsstufe 0 stattfinden.

Abbildung 5.3 zeigt den Vergleich zwischen der kooperativen Variante und der Fast-Reject-Variante.

Es zeigt sich eine Verschlechterung der Laufzeitperformance durch Fast-Reject

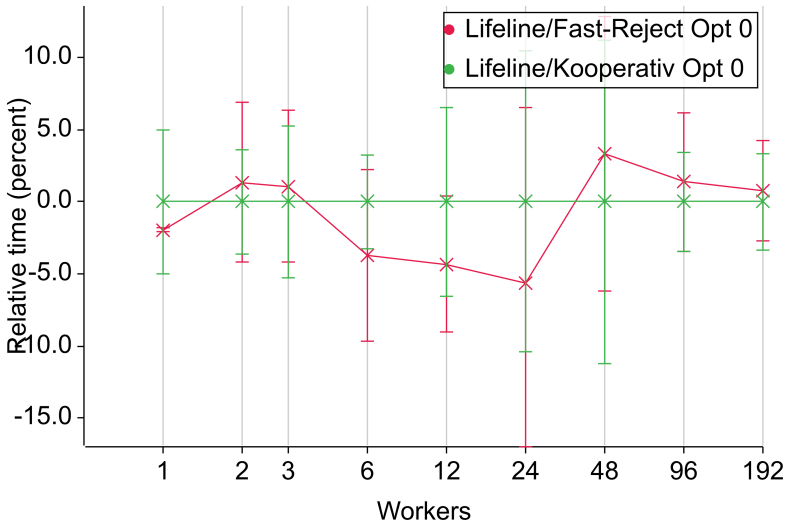

Abbildung 5.3: Vergleich der schnellsten Lifeline-Varianten

um durchschnittlich ca. 1-4 % bei höheren Anzahlen (48-192) von Workern, wobei für kleinere Workeranzahlen (6-24) Fast-Reject bei einigen Benchmarks (Synthetic, UTS) schneller ist.

Wir stellen somit fest, dass bei Einsatz des Lifeline-Schemas die Verwendung der originalen kooperativen Variante mit Optimierungsstufe 0 innerhalb unserer Testreihe die beste Performance lieferte.

In Anbetracht der recht hohen relativen Standardabweichungen von bis zu 5 % wäre möglicherweise eine Durchführung der Benchmarks mit häufigeren Iterationen sinnvoll, um genauere Ergebnisse zu erhalten.

5.3.2 Random-Verfahren

Abbildung 5.4 zeigt die Ergebnisse bei der Verwendung des Random-Verfahrens in der kooperativen Variante mit Binärbaum-Verteilung. Bei dieser Variante ist die

Optimierungsstufe 3 schneller als die anderen Stufen (3 % im Vergleich zu Stufe 0 bei 192 Workern).

Abbildung 5.5 zeigt die Ergebnisse, wenn Fast-Reject zur Verarbeitung der Steal-Anfragen eingesetzt wird. Hier zeigt sich die Optimierungsstufe 0 als am schnellsten für hohe Workeranzahlen.

In Abbildung 5.6 wird Fast-Reject mit reinem Workstealing zur Verteilung kombiniert. Optimierungsstufe 2 ist hier für Workeranzahlen am performantesten.

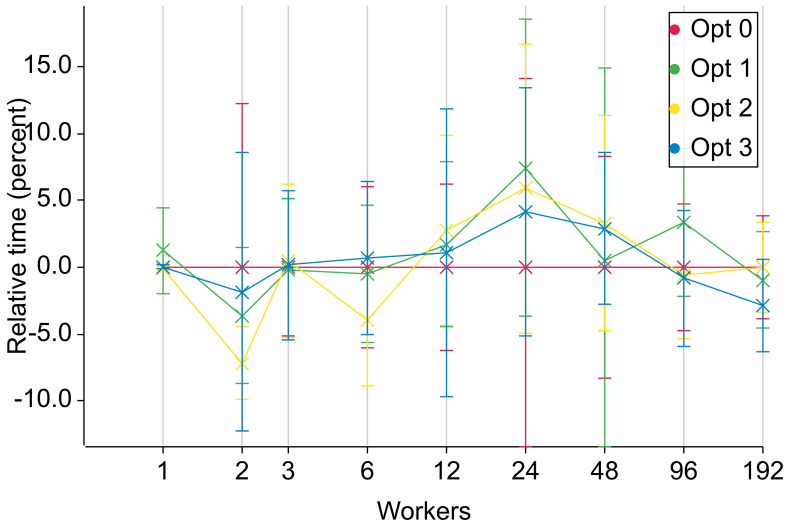

Abbildung 5.4: Vergleich von Optimierungen bei **Random/Kooperativ mit Binärbaum-Verteilung** (in Prozent)

Bei allen drei Varianten zeigt sich jedoch eine hohe Varianz der Ergebnisse, die eindeutige Aussagen über Performanceverbesserungen erschwert.

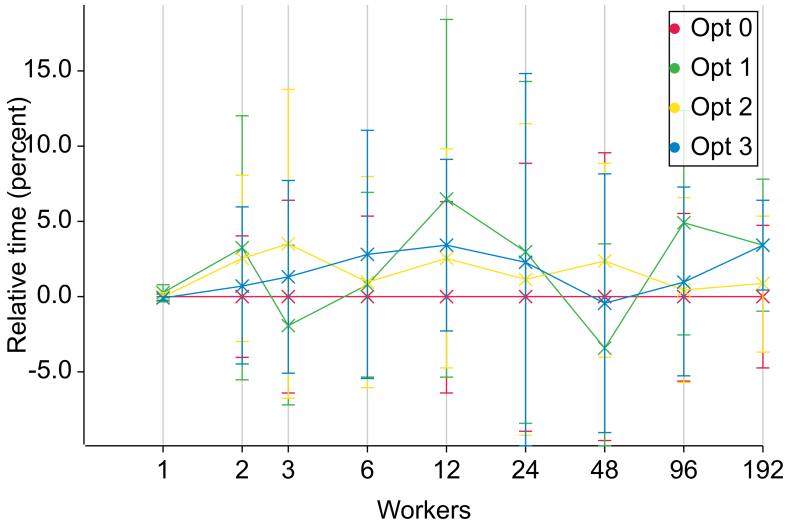

Abbildung 5.5: Vergleich von Optimierungen bei **Random/Fast-Reject mit Binärbaum-Verteilung** (in Prozent)

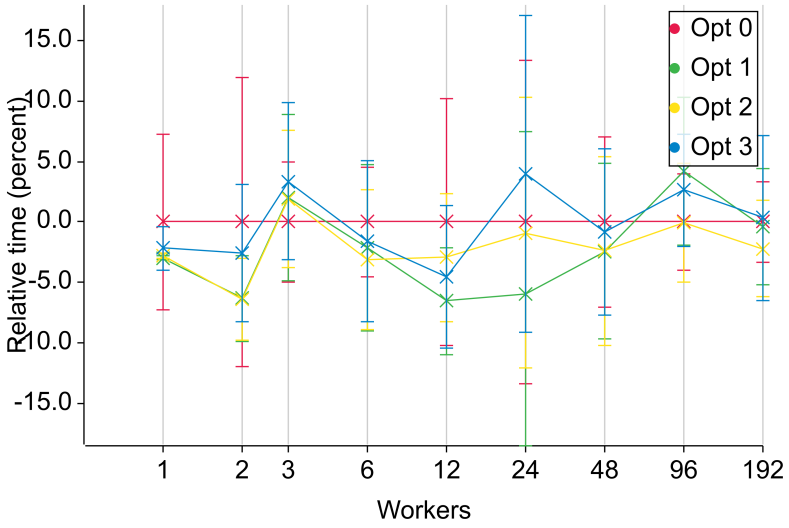

Abbildung 5.6: Vergleich von Optimierungen bei **Random/Fast-Reject mit Random-Verteilung** (in Prozent)

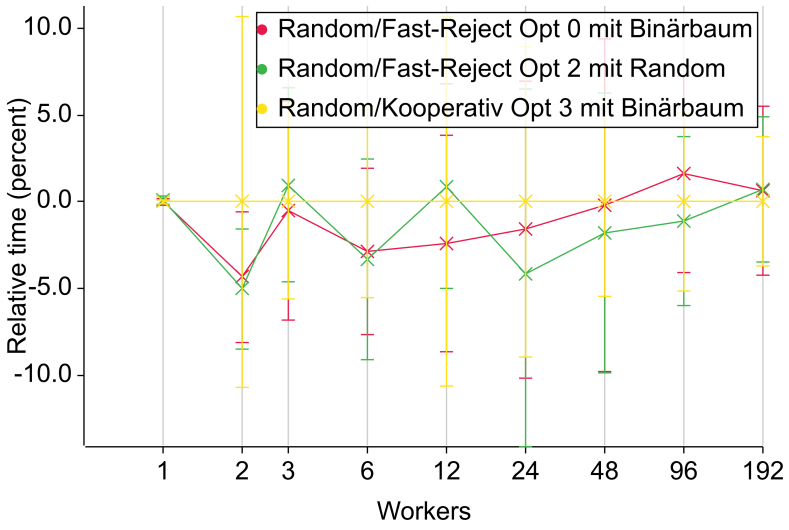

Abbildung 5.7: Vergleich der schnellsten Random-Varianten (in Prozent)

Abbildung 5.7 zeigt den Vergleich zwischen den Varianten. Hier lässt sich keine eindeutig schnellste Variante identifizieren. Bei 192 Workern dominiert die Variante **Random/Kooperativ Opt 3 mit Binärbaum** knapp (ca. 1 % schneller als die anderen Varianten), für geringere Anzahlen ist **Random/Fast-Reject Opt 2 mit Random-Verteilung** durchschnittlich schneller. Bei der **Fib**-Benchmark ist **Random/Kooperativ Opt 3 mit Binärbaum** jedoch die langsamste Variante, gewinnt aber dafür bei den **Synthetic**- und **UTS**-Benchmarks deutlich.

Der Vergleich mit dem Lifeline-Schema ist durch die ähnliche Performance aller Varianten ohnehin repräsentativ. Wir verwenden für den Vergleich die beiden Varianten **Random/Kooperativ Opt 3 mit Binärbaum** und **Random/Fast-Reject Opt 2 mit Random-Verteilung**, um den besten Fall sowohl für kleine als auch für große Workeranzahlen jeweils mit dem Lifeline-Schema vergleichen zu können.

5.3.3 Vergleich der Verfahren

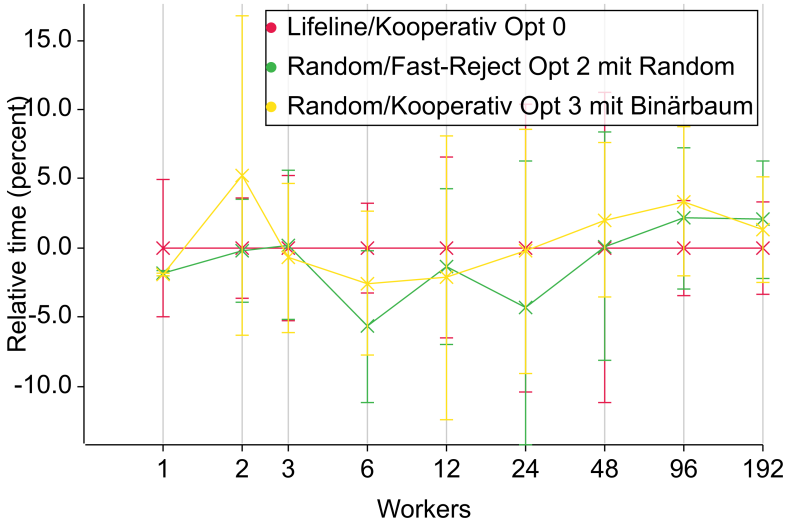

Abbildung 5.8: Vergleich der schnellsten Verfahren (in Prozent)

Abbildung 5.8 zeigt den Vergleich zwischen der Verwendung der schnellsten Lifeline-Variante **Lifeline/Kooperativ Opt 0** und den beiden Varianten des Random-Verfahrens.

Für höhere Workeranzahlen (48-192) sind die Random-Varianten im Durchschnitt zwischen 1 und 4 % langsamer als die Lifeline-Variante. Bei den niedrigeren Workeranzahlen (3-24) hingegen sind die Random-Varianten bis zu 5 % schneller (**Random/Fast-Reject Opt 2 mit Random** bei 6 Workern). Bei Betrachtung der einzelnen Benchmarks ist zu erkennen, dass die Lifeline-Variante für hohe Workeranzahlen bei allen Benchmarks die beste Performance liefert.

5.4 Schlussfolgerungen

Anhand der Ergebnisse der Benchmarks können wir feststellen, dass innerhalb der Testreihe die Lokalitätsoptimierungen aus Abschnitt 3.3 vor allem für das Lifeline-Schema keine hohe Relevanz hatten oder sogar die Performance verschlechtert haben. Bei der Verwendung des Random-Verfahrens hingegen sorgten die Optimierungen für Performance-Verbesserungen.

Die neu eingeführte Fast-Reject-Variante hingegen sorgte bei höheren Workeranzahlen weder bei dem Lifeline-Schema noch beim Random-Verfahren für Verbesserungen, sondern erhöhte die Laufzeiten sogar. Der Overhead durch häufigere Steal-Versuche scheint höher zu sein als der Leistungsgewinn durch schnellere Ablehnung und Versuchsdurchführung. Insofern kann diese Variante nicht als eine erfolgreiche Weiterentwicklung betrachtet werden.

Insgesamt zeigt sich, dass das Lifeline-Verfahren bei der Auswahl der besten Variante durchschnittlich schneller ist als die beste Random-Variante. Jedoch ist der Unterschied mit <4 % geringer als die Unterschiede zwischen verschiedenen Lifeline-Varianten. Folglich ist die Auswahl der Parameter für die Performance des Lifeline-Verfahrens sehr wichtig.

Jedoch ist zu Bedenken, dass die Ergebnisse über 7 Iterationen entstanden sind und maximal 192 Worker auf 8 Knoten verwendet wurden. Weitere Testreihen mit höheren Workeranzahlen und mehr Iterationen könnten hilfreich sein, um auch Skalierungseffekte genauer betrachten zu können.

6 Zusammenfassung

Der Vergleich zwischen zwei Verfahren zur Lastenbalancierung sowie verschiedener Varianten und Optimierungen ergab, dass das Lifeline-Schema für eine hohe Anzahl von Workern eine bessere Performance bietet als das rein zufallsbasierte Random-Verfahren.

Eine Variante (Fast-Reject), die eine schnellere Verarbeitung von Anfragen auf Kosten einer höheren Fehlerrate zum Ziel hatte, hat für keines der beiden Verfahren eine überzeugende Verbesserung erzielt.

Jedoch ist auch festzustellen, dass die Versuche einer Optimierung zur Ausnutzung von Lokalitätseffekten keine Verbesserung beim Lifeline-Verfahren zur Folge hatten, sondern dass eine suboptimale Auswahl sogar eine schlechtere Performance als einige zufallsbasierte Varianten verursachte.

Das im originalen Lifeline-Paper [10] erklärte Ziel des Lifeline-Schemas, die Anzahl der fehlschlagenden Anfragen zu minimieren, um eine gleiche bis bessere Performance als die zufallsbasierten Verfahren zu bieten, kann hier also für alle eingesetzten Benchmarks bestätigt werden (und nicht nur für UTS, wie darin erwähnt).

Die Lokalitätsoptimierungen wurden aus der Publikation von Reitz u. a. [12] übernommen, auf der auch die Implementierung basiert, und waren ursprünglich für eine GLB-Variante, die DIT als Taskmodell einsetzte, entwickelt worden. Die festge-

stellten Verbesserungen durch die Optimierungen konnten hier nicht experimentell bestätigt werden, wobei eine umfangreichere Datenlage und weitere Untersuchungen hinsichtlich der Ursachen von Interesse wären.

Ebenfalls auf dieser Publikation basierend vergleicht die Bachelorarbeit von Stitz [12] das Lifeline-Schema mit zufallsbasierten Verfahren sowie verschiedenen Varianten, jedoch ebenfalls im Kontext des DIT-Taskmodells. Wir können somit feststellen, dass das Lifeline-Schema unabhängig vom unterliegenden Taskmodell schneller ist als die zufallsbasierten Verfahren, auch wenn die Resultate zu Lokalitätsoptimierungen sich unterscheiden.

Eine weitere Methode zur Lastenbalancierung stellen koordinierte Workstealing-Verfahren dar, wie die auf einer SplitQueue basierende GLB-Variante [11]. Ein Vergleich dieser Verfahren mit den vorgestellten wäre ebenfalls von Interesse.

Zudem wären weitere Untersuchungen mit mehr Workern interessant, um Informationen über Skalierungseffekte zu erhalten.

Literaturverzeichnis

[1] Charles Leiserson und Aske Plaat. „Programming Parallel Applications in Cilk“. In: *Siam news* (Juli 1997).

[2] Olivier Tardieu. „The APGAS Library: Resilient Parallel and Distributed Programming in Java 8“. In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10 2015. Portland, OR, USA: Association for Computing Machinery, 2015, S. 25–26. ISBN: 9781450335867. DOI: 10.1145/2771774.2771780. URL: https://doi.org/10.1145/2771774.2771780.

[3] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky und Olivier Tardieu. „The asynchronous partitioned global address space model“. In: *The First Workshop on Advances in Message Passing*. 2010, S. 1–8.

[4] *X10: Performance and Productivity at Scale*. URL: http://x10-lang.org/ (besucht am 05. 05. 2023).

[5] *Hazelcast | Real-Time Stream Processing Platform*. URL: https://hazelcast.com/ (besucht am 05. 05. 2023).

[6] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat und Mikio Takeuchi. *GLB: Lifeline-based Global Load Balancing library in X10*. 2013. arXiv: 1312.5691 [cs.DC].

[7] Jonas Posner und Claudia Fohry. „Cooperation vs. Coordination for Lifeline-Based Global Load Balancing in APGAS“. In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. X10 2016. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, S. 13–17. ISBN: 9781450343862. DOI: 10.1145/2931028.2931029. URL: https://doi.org/10.1145/2931028.2931029.

[8] Lukas Reitz. „Load Balancing Policies for Nested Fork-Join“. In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 2021, S. 817–818. DOI: 10.1109/Cluster48925.2021.00075.

[9] Lukas Reitz, Kai Hardenbicker, Tobias Werner und Claudia Fohry. „Lifeline-based load balancing schemes for Asynchronous Many-Task runtimes in clusters“. In: *Parallel Computing* 116 (2023), S. 103020. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2023.103020. URL: https://www.sciencedirect.com/science/article/pii/S0167819123000261.

[10] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove und Sriram Krishnamoorthy. „Lifeline-Based Global Load Balancing“. In: *SIGPLAN Not.* 46.8 (Feb. 2011), S. 201–212. ISSN: 0362-1340. DOI: 10.1145/2038037.1941582. URL: https://doi.org/10.1145/2038037.1941582.

[11] Tobias Werner. „Anwendung einer SplitQueue Datenstruktur auf Work Stealing in Laufzeitsystemen taskbasierter paralleler Programmiersysthemen“. Bachelorarbeit. Universität Kassel, 2022.

[12] Konstantin Stitz. „Effizienzvergleich von zwei Lastenbalancierungsverfahren für parallele Programmiersysteme“. Bachelorarbeit. Universität Kassel, 2023.

Anhang

1 Digitale Abgabe

Die digitale Abgabe enthält nebst dem Quellcode der Implementierung die vollständigen Logs und Ergebnisse der Benchmarks sowie die Software zur Auswertung der Benchmarks.

2 Messergebnisse

Die Tabellen in diesem Kapitel enthalten die Messergebnisse aus den Benchmarks. Es wurden der Kürze halber nicht für sämtliche Workeranzahlen Werte eingefügt.

Tabelle 1: Fib Random Kooperativ Binärbaum (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4532.491	5.597	1.000
12	462.608	15.939	9.798
48	154.145	25.982	29.404
192	44.248	2.181	102.433

Tabelle 2: Synthetic Random Kooperativ Binärbaum (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.120	0.017	1.000
12	101.991	0.088	0.991
48	102.447	0.064	0.987
192	103.331	0.135	0.979

Tabelle 3: UTS Random Kooperativ Binärbaum (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6730.022	18.756	1.000
12	794.800	120.273	8.468
48	204.025	16.162	32.986
192	49.540	3.161	135.851

Tabelle 4: Fib Random Fast-Reject Random (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4533.403	3.880	1.000
12	484.312	80.168	9.360
48	181.469	26.466	24.982
192	43.557	3.228	104.080

Tabelle 5: Synthetic Random Fast-Reject Random (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.115	0.017	1.000
12	101.720	0.077	0.994
48	102.369	0.088	0.988
192	103.056	0.122	0.981

Tabelle 6: UTS Random Fast-Reject Random (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	7368.292	1605.319	1.000
12	874.450	120.996	8.426
48	192.763	12.358	38.225
192	50.311	1.193	146.456

Tabelle 7: Fib Random Fast-Reject Binärbaum (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4538.297	6.077	1.000
12	461.615	13.967	9.831
48	169.509	29.378	26.773
192	41.428	4.390	109.546

Tabelle 8: Synthetic Random Fast-Reject Binärbaum (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.103	0.010	1.000
12	101.982	0.115	0.991
48	102.477	0.079	0.987
192	103.240	0.067	0.979

Tabelle 9: UTS Random Fast-Reject Binärbaum (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6728.051	30.600	1.000
12	762.336	121.429	8.826
48	200.143	22.375	33.616
192	49.199	1.752	136.753

Tabelle 10: Fib Lifeline Kooperativ Lifeline (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4534.730	6.009	1.000
12	499.305	12.050	9.082
48	161.471	28.849	28.084
192	39.086	2.637	116.019

Tabelle 11: Synthetic Lifeline Kooperativ Lifeline (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.106	0.017	1.000
12	102.146	0.132	0.990
48	102.337	0.083	0.988
192	102.871	0.089	0.983

Tabelle 12: UTS Lifeline Kooperativ Lifeline (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	7149.104	1052.309	1.000
12	808.909	138.229	8.838
48	199.991	31.467	35.747
192	49.416	1.533	144.672

Tabelle 13: Fib Lifeline Fast-Reject Lifeline (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4532.789	10.174	1.000
12	487.865	10.565	9.291
48	169.730	23.098	26.706
192	39.965	2.278	113.418

Tabelle 14: Synthetic Lifeline Fast-Reject Lifeline (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.100	0.006	1.000
12	102.056	0.259	0.991
48	102.343	0.081	0.988
192	102.899	0.074	0.983

Tabelle 15: UTS Lifeline Fast-Reject Lifeline (Opt 0)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6738.870	12.625	1.000
12	722.888	94.947	9.322
48	209.792	28.472	32.122
192	49.420	2.237	136.358

Tabelle 16: Fib Random Kooperativ Binärbaum (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4562.072	91.400	1.000
12	495.801	11.820	9.201
48	153.201	28.098	29.778
192	42.076	3.361	108.424

Tabelle 17: Synthetic Random Kooperativ Binärbaum (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.113	0.009	1.000
12	101.994	0.060	0.991
48	102.561	0.250	0.986
192	103.246	0.165	0.979

Tabelle 18: UTS Random Kooperativ Binärbaum (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6936.571	518.171	1.000
12	778.580	127.167	8.909
48	207.792	51.085	33.382
192	50.574	1.528	137.157

Tabelle 19: Fib Random Fast-Reject Random (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4535.787	6.536	1.000
12	455.983	3.220	9.947
48	166.506	25.676	27.241
192	42.962	4.588	105.578

Tabelle 20: Synthetic Random Fast-Reject Random (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.120	0.031	1.000
12	101.700	0.100	0.994
48	102.346	0.104	0.988
192	103.014	0.143	0.982

Tabelle 21: UTS Random Fast-Reject Random (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6712.499	25.207	1.000
12	754.044	110.493	8.902
48	194.643	14.358	34.486
192	50.515	1.877	132.880

Tabelle 22: Fib Random Fast-Reject Binärbaum (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4533.581	6.413	1.000
12	532.048	80.722	8.521
48	168.350	26.808	26.929
192	43.652	2.517	103.857

Tabelle 23: Synthetic Random Fast-Reject Binärbaum (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.110	0.014	1.000
12	102.052	0.092	0.991
48	102.609	0.194	0.985
192	103.209	0.186	0.980

Tabelle 24: UTS Random Fast-Reject Binärbaum (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6783.011	106.101	1.000
12	794.592	137.816	8.536
48	180.770	9.655	37.523
192	51.589	3.395	131.481

Tabelle 25: Fib Lifeline Kooperativ Lifeline (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4536.940	9.309	1.000
12	534.242	82.479	8.492
48	172.317	34.659	26.329
192	42.296	4.360	107.268

Tabelle 26: Synthetic Lifeline Kooperativ Lifeline (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.113	0.026	1.000
12	102.126	0.232	0.990
48	102.445	0.121	0.987
192	103.311	0.085	0.979

Tabelle 27: UTS Lifeline Kooperativ Lifeline (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6738.851	30.603	1.000
12	788.409	125.657	8.547
48	209.761	28.668	32.126
192	51.374	3.953	131.173

Tabelle 28: Fib Lifeline Fast-Reject Lifeline (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4534.120	5.808	1.000
12	499.564	21.505	9.076
48	175.061	22.978	25.900
192	42.069	2.605	107.779

Tabelle 29: Synthetic Lifeline Fast-Reject Lifeline (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.117	0.019	1.000
12	101.972	0.198	0.992
48	102.347	0.195	0.988
192	103.269	0.151	0.979

Tabelle 30: UTS Lifeline Fast-Reject Lifeline (Opt 1)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6739.195	28.332	1.000
12	868.546	173.656	7.759
48	201.688	11.543	33.414
192	49.673	3.095	135.670

Tabelle 31: Fib Random Kooperativ Binärbaum (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4531.658	5.452	1.000
12	453.254	3.434	9.998
48	176.662	23.859	25.652
192	44.697	3.752	101.386

Tabelle 32: Synthetic Random Kooperativ Binärbaum (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.129	0.025	1.000
12	102.401	0.071	0.988
48	102.306	0.088	0.988
192	103.169	0.170	0.980

Tabelle 33: UTS Random Kooperativ Binärbaum (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6712.858	18.367	1.000
12	873.893	162.512	7.682
48	194.587	17.359	34.498
192	49.056	0.716	136.840

Tabelle 34: Fib Random Fast-Reject Random (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4538.676	9.172	1.000
12	456.759	8.620	9.937
48	163.443	25.640	27.769
192	40.847	3.270	111.113

Tabelle 35: Synthetic Random Fast-Reject Random (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.133	0.032	1.000
12	101.706	0.061	0.994
48	102.486	0.417	0.987
192	103.079	0.238	0.981

Tabelle 36: UTS Random Fast-Reject Random (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6743.678	24.833	1.000
12	847.940	122.565	7.953
48	197.856	16.998	34.084
192	50.131	2.081	134.520

Tabelle 37: Fib Random Fast-Reject Binärbaum (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4538.034	10.227	1.000
12	464.739	19.590	9.765
48	184.282	16.153	24.625
192	42.375	3.622	107.093

Tabelle 38: Synthetic Random Fast-Reject Binärbaum (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.114	0.023	1.000
12	102.337	0.256	0.988
48	102.347	0.216	0.988
192	103.343	0.176	0.978

Tabelle 39: UTS Random Fast-Reject Binärbaum (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6738.058	31.882	1.000
12	812.259	131.956	8.295
48	197.500	19.314	34.117
192	49.299	2.270	136.678

Tabelle 40: Fib Lifeline Kooperativ Lifeline (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4535.967	15.436	1.000
12	500.200	7.626	9.068
48	158.458	26.377	28.626
192	40.079	2.764	113.176

Tabelle 41: Synthetic Lifeline Kooperativ Lifeline (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.120	0.022	1.000
12	102.314	0.235	0.988
48	102.446	0.284	0.987
192	103.190	0.178	0.980

Tabelle 42: UTS Lifeline Kooperativ Lifeline (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6748.149	24.196	1.000
12	878.422	125.748	7.682
48	200.800	24.233	33.606
192	49.002	1.936	137.713

Tabelle 43: Fib Lifeline Fast-Reject Lifeline (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4727.850	482.656	1.000
12	494.920	12.825	9.553
48	166.637	26.958	28.372
192	42.471	3.077	111.319

Tabelle 44: Synthetic Lifeline Fast-Reject Lifeline (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.112	0.013	1.000
12	101.975	0.258	0.992
48	102.346	0.262	0.988
192	103.027	0.041	0.981

Tabelle 45: UTS Lifeline Fast-Reject Lifeline (Opt 2)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6952.999	548.303	1.000
12	839.310	132.094	8.284
48	182.239	14.386	38.153
192	47.774	1.352	145.539

Tabelle 46: Fib Random Kooperativ Binärbaum (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4535.561	4.190	1.000
12	489.516	78.815	9.265
48	163.400	13.208	27.757
192	39.915	3.087	113.632

Tabelle 47: Synthetic Random Kooperativ Binärbaum (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.111	0.011	1.000
12	102.336	0.226	0.988
48	102.253	0.095	0.989
192	103.051	0.142	0.981

Tabelle 48: UTS Random Kooperativ Binärbaum (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6731.306	15.949	1.000
12	771.039	120.045	8.730
48	209.785	16.967	32.087
192	50.249	1.675	133.958

Tabelle 49: Fib Random Fast-Reject Random (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4628.360	224.789	1.000
12	451.917	4.618	10.242
48	166.318	22.406	27.828
192	42.775	4.290	108.202

Tabelle 50: Synthetic Random Fast-Reject Random (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.098	0.013	1.000
12	101.692	0.096	0.994
48	102.434	0.459	0.987
192	102.953	0.214	0.982

Tabelle 51: UTS Random Fast-Reject Random (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6734.395	33.485	1.000
12	813.807	145.925	8.275
48	204.041	15.067	33.005
192	51.781	5.198	130.056

Tabelle 52: Fib Random Fast-Reject Binärbaum (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4529.949	3.880	1.000
12	453.430	2.639	9.990
48	166.149	27.784	27.264
192	45.459	2.370	99.650

Tabelle 53: Synthetic Random Fast-Reject Binärbaum (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.106	0.025	1.000
12	102.466	0.284	0.987
48	102.358	0.077	0.988
192	103.070	0.144	0.981

Tabelle 54: UTS Random Fast-Reject Binärbaum (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6721.154	15.381	1.000
12	851.262	123.919	7.896
48	201.797	18.619	33.307
192	49.571	1.502	135.586

Tabelle 55: Fib Lifeline Kooperativ Lifeline (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4532.921	3.444	1.000
12	503.796	16.790	8.998
48	150.098	19.914	30.200
192	40.426	2.266	112.128

Tabelle 56: Synthetic Lifeline Kooperativ Lifeline (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.117	0.013	1.000
12	102.323	0.190	0.988
48	102.217	0.192	0.989
192	102.895	0.183	0.983

Tabelle 57: UTS Lifeline Kooperativ Lifeline (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6775.491	40.427	1.000
12	785.210	137.899	8.629
48	214.447	24.197	31.595
192	48.554	1.785	139.546

Tabelle 58: Fib Lifeline Fast-Reject Lifeline (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	4533.338	6.320	1.000
12	499.694	9.069	9.072
48	171.368	23.236	26.454
192	41.802	1.605	108.447

Tabelle 59: Synthetic Lifeline Fast-Reject Lifeline (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	101.105	0.016	1.000
12	102.299	0.357	0.988
48	102.387	0.222	0.987
192	102.874	0.249	0.983

Tabelle 60: UTS Lifeline Fast-Reject Lifeline (Opt 3)

Worker	Durchschnittszeit (s)	Standardabweichung (s)	Speedup
1	6731.977	27.092	1.000
12	822.365	124.927	8.186
48	196.374	23.548	34.281
192	50.232	3.023	134.018