Universität Kassel

Bachelorarbeit

Protokollierung und Visualisierung des Laufzeitverhaltens einer Taskpool-Implementierung

von Jan Bingemann

Kassel, 12. März 2020

Gutachter:
Prof. Dr. Claudia Fohry
Prof. Dr. Albert Zündorf

Inhaltsverzeichnis

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, 12. März 2020

Jan Bingemann

Abkürzungen

APGAS	Asynchronous Partitioned Global Address Space Bibliothek [19]
GLB	Global Load Balancing Framework [21]
J_GLB	Kooperative Java- und APGAS-Variante des GLB Frameworks [14]
FT_GLB	Fehlertolerante Weiterentwicklung von J_GLB [13]
GLB_Logger	Neue Logging-Software, die in dieser Arbeit vorgestellt wird
BC	Betweenness Centrality Benchmark [4]
UTS	Unbalanced Tree Search Benchmark [10]

1 Einleitung

Mit der Verwendung immer größerer Computercluster steigt die Wahrscheinlichkeit von Knotenausfällen. Durch Fehlertoleranz wird sichergestellt, dass beim Ausfall von einem oder mehreren Knoten die Zwischenergebnisse der Berechnung nicht komplett verloren gehen. Eine übliche Methode zur Fehlertoleranz ist das Checkpoint-Restart-Verfahren. Dabei wird regelmäßig der Zwischenstand der Programmausführung gespeichert. Nach einem Absturz kann das Programm an dem gespeicherten Punkt weiterlaufen. Da bisher übliche Realisierungen auf System-Ebene alle Daten der Programmausführung speichern, erzeugen sie einen großen Overhead. Daher sind Lösungen auf Anwendungsebene, als effizientere Alternative, Thema der Forschung. Hier werden nur die nötigsten Daten gesichert.

Für den Programmierer bedeutet Fehlertoleranz auf Anwendungsebene jedoch einen hohen zusätzlichen Implementationsaufwand. Das Fachgebiet entwickelte deshalb ein generisches Framework (*FT_ GLB* [13]). Verwendet der Programmierer dieses, ist Fehlertoleranz für sein Programm garantiert und der Implementationsaufwand wird deutlich reduziert. Das Framework ist eine Weiterentwicklung des ebenfalls im Fachgebiet entwickelten, nicht-fehlertoleranten *J_ GLB* [14].

Da beide Frameworks auch weiterhin Thema aktueller Forschung des Fachgebiets sind, wurde eine Möglichkeit gesucht, ihr Laufzeitverhalten besser verständlich zu machen. Im Rahmen dieser Bachelorarbeit habe ich *GLB_ Logger*, eine neue Logging-Komponente für die Frameworks, entwickelt. Sie protokolliert das Laufzeitverhalten der Frameworks und generiert aus diesen Informationen Graphiken zur Visualisierung.

Beiden Frameworks liegt das Taskpool-Pattern zu Grunde. Dabei wird eine große Anzahl von Tasks definiert, die parallel verarbeitet werden. Die Tasks werden auf eine feste Anzahl von Workern abgebildet. Jeder Worker speichert einen Teil der Tasks in einem lokalen Taskpool. In der Berechnungsphase nimmt er die Tasks sukzessive aus seinem lokalen Taskpool und verarbeitet sie. Dabei können weitere Tasks entstehen, die zum lokalen Taskpool hinzugefügt werden. Lastenverteilung wird dadurch erreicht, dass untätige Worker versuchen, Tasks von anderen Workern zu stehlen.

GLB_Logger zeichnet die unterschiedlichen Zustände auf, in denen sich ein Worker

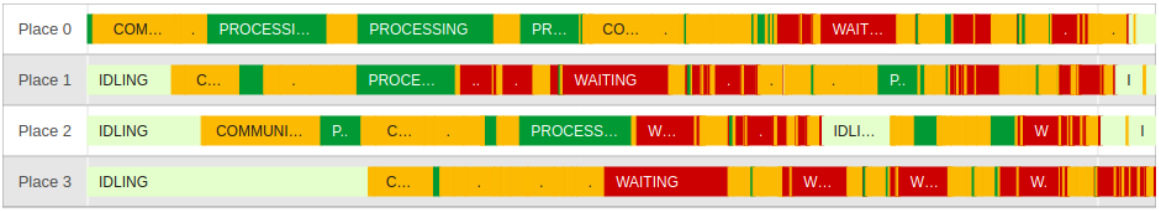


Abbildung 1: Zeitleiste

befinden kann:

- *Processing*: der Worker verarbeitet Tasks.
- *Communication*: der Worker sendet eine Nachricht an einen anderen Worker, um beispielsweise eine Stehl-Anfrage zu tätigen oder Daten in einer globalen Datenstruktur zu sichern.
- *Waiting*: nach dem Senden einer Stehl-Anfrage wartet der Worker auf die Antwort des anderen Workers.
- *Idling*: der Worker wurde noch nicht gestartet oder er ist untätig, weil er keine Tasks mehr hat und seine Stehl-Anfragen abgelehnt wurden.

Auf jedem Worker läuft eine Logger-Instanz, die aufgerufen wird, sobald der Worker in einen neuen Zustand wechselt. Dabei entstehen Zeitabschnitte, bestehend aus Start- und Endzeitpunkt, sowie dem zugehörigen Zustand. Für den Endzeitpunkt wird in der Regel beim Zustandswechsel der Startzeitpunkt des nächsten Zeitabschnitts eingetragen. Nach der Berechnung verarbeitet GLB_Logger diese Daten und erzeugt Textdateien. Diese dienen einerseits dem Benutzer unmittelbar als Informationsquelle, andererseits werden sie als Input für zwei graphische Darstellungen verwendet. Wie in Abbildung 1 zu sehen, ist die erste Graphik eine Zeitleiste, in der die Zeitabschnitte jedes Workers (Places) dargestellt werden. Die zweite Graphik, die in Abbildung 2 exemplarisch dargestellt wird, ist ein Histogramm. Hier werden zu jedem Zeitpunkt die Anteile der verschiedenen Zustände bezogen auf alle Worker dargestellt.

FT_GLB realisiert Fehlertoleranz auf Anwendungsebene, indem es regelmäßig Backups schreibt. Dabei sammelt GLB_Logger Daten wie bspw. die Zeitpunkte und Dauer des Backup-Schreibens, die Größe des Backups oder die Anzahl der Tasks,

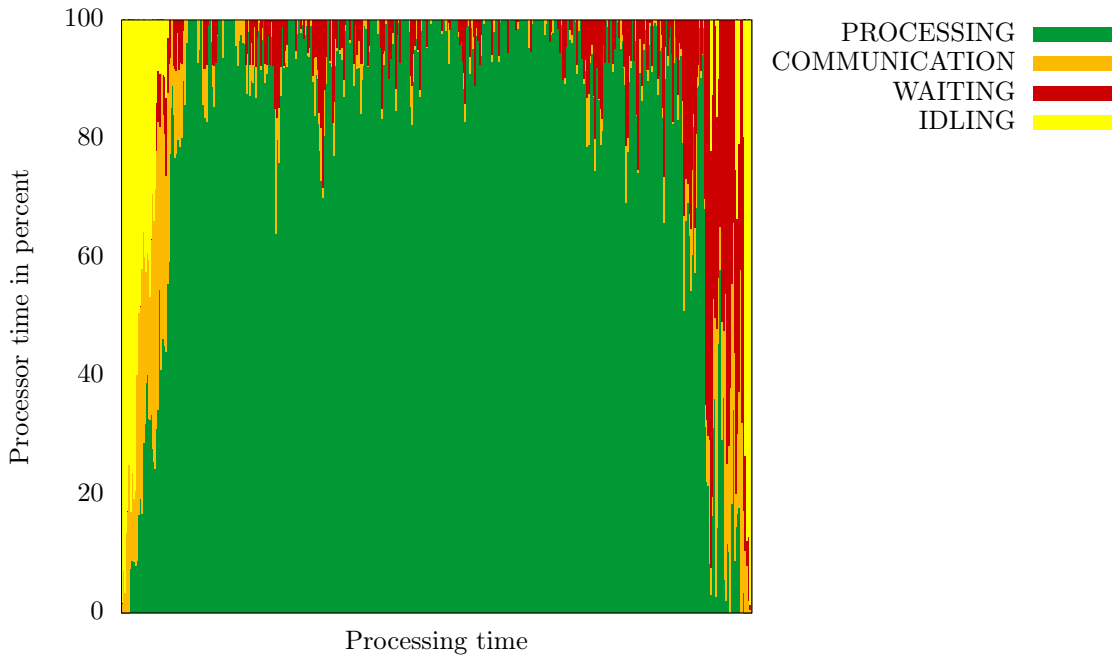


Abbildung 2: Histogramm

die im Backup gespeichert wurden. Der Benutzer erhält als Ausgabe eine Textdatei, aus der er alle Informationen zu den aufgezeichneten Backups entnehmen kann. Da J_GLB keine Backups schreibt, werden diese Daten nur für FT_GLB erhoben.

Die Zeitleiste, das Histogramm, sowie die Informationen der Textdateien gewähren Einblicke in die Funktionsweise der Frameworks. Mit ihrer Hilfe ist es mir unter anderem gelungen, die charakteristischen Laufzeitverhalten zweier Benchmarks visuell darzustellen (siehe Abschnitt 6.1). Auch die Unterschiede der beiden Frameworks J_GLB und FT_GLB konnten sichtbar gemacht werden (siehe Abschnitt 6.2). Des Weiteren konnte ich zeigen, wie die Graphiken dabei helfen, die Performance der Frameworks auf spezifische Anwendungen zu optimieren (siehe Abschnitt 6.3). In einer weiteren praktischen Anwendung zeigte ich die Nützlichkeit des GLB_Loggers durch das Aufdecken mehrerer potenzieller Performance-Probleme (siehe Abschnitt 6.4).

Die folgende Arbeit gliedert sich in acht Abschnitte. In Abschnitt 2 werden J_GLB und FT_GLB vorgestellt. Danach wird in Abschnitt 3 das grundlegende Konzept von GLB_Logger erklärt. In Abschnitt 4 wird die Implementierung der Logging-Komponente vorgestellt und in Abschnitt 5 die darauf aufbauende Generierung der

beiden Graphiken. Danach werden in Abschnitt 6 die Ergebnisse beschrieben, die bei der Ausführung der beiden Frameworks mit der neuen Logging-Komponente aufgezeichnet wurden, sowie die daraus gewonnenen Erkenntnisse. Im Anschluss nennen wir in Abschnitt 7 verwandte Arbeiten und vergleichen GLB_Logger mit dem Visualisierungswerkzeug des parallelen Programmiersystems Legion. Abschließend werden in Abschnitt 8 die Ergebnisse der Arbeit zusammengefasst und bewertet.

2 Grundlagen

2.1 APGAS Bibliothek

Die *APGAS* Bibliothek [19] wurde benutzt, um J_GLB und FT_GLB zu parallelisieren.

Ihr liegt das APGAS-Modell zu Grunde, welches den verteilten Speicher eines Clusters zu einem globalen Adressraum abstrahiert, der in Partitionen aufgeteilt ist. Als *Place* bezeichnet es eine dieser Speicher-Partitionen und die dieser zugeordneten Prozessoren. In den meisten Fällen entspricht ein Place einem Knoten des Clusters. Recheneinheiten eines Places können auf alle anderen Places zugreifen, Zugriff auf den lokalen Speicher ist jedoch schneller.

Im APGAS-Modell werden asynchrone Tasks, genannt *Activities*, lokal oder auf einem entfernten Place gestartet. Die Berechnung beginnt auf Place 0. Von dort werden sukzessive weitere Tasks erzeugt und so die anderen Places mit Tasks versorgt.

2.2 J_GLB Framework

J_GLB ist die Java-Implementierung des Taskpool-Frameworks *GLB* [21], welches ein Teil der Standard-Bibliothek der parallelen Programmiersprache X10 ist und der Lastenbalancierung dient.

J_GLB implementiert das Taskpool-Pattern, das bereits in der Einleitung erklärt wurde. Das Stehl-Verfahren hat einige Besonderheiten, da J_GLB die Lifeline-Variante des Taskpool-Patterns anwendet. Deshalb wird es im Folgenden genauer ausgeführt.

Jeder Worker besitzt einen lokalen Taskpool, aus dem er Tasks nimmt, um sie zu verarbeiten. Ist sein Taskpool leer, versucht der Worker, Tasks von anderen Workern zu stehlen. Der Worker, der den Stehl-Versuch tätigt, wird dabei *Thief* genannt. Er wählt zufällig einen anderen Worker aus und sendet diesem eine Stehl-Anfrage. Dafür trägt er sich in die Liste der registrierten Thieves des ausgewählten Workers ein, der auch als *Victim* bezeichnet wird. Stehl-Versuche in J_GLB sind kooperativ. Der Thief wartet auf die Antwort zu seiner Anfrage und hat keinen direkten Zugriff auf den

lokalen Taskpool des Victims. Die Antwort ist entweder eine Absage oder eine Menge von Tasks, genannt *Loot*. Der Thief versucht auf diese Art, von bis zu w zufälligen Victims Tasks zu stehlen.

Schlugen diese Stehl-Versuche fehl, tätigt er z weitere Stehl-Versuche mit so genannten *Lifeline-Buddies*. Letztere sind seit Beginn festgelegt, sodass jeder Worker z andere Worker als Lifeline Buddies hat. Diese unidirektionale Verbindung von einem Worker zu einem anderen Worker ist eine *Lifeline* [15]. Jede Lifeline besitzt ein Flag, durch den sie aktiviert oder deaktiviert werden kann. Nutzt ein Worker eine seiner Lifelines für einen der z Lifeline-Stehl-Versuche, wird sie durch das Setzen des Flags aktiviert (er tut dies nur, wenn die Lifeline nicht bereits aktiviert ist). Hat der Lifeline Buddy zum Zeitpunkt der Anfrage keine Tasks, bleibt die Lifeline bestehen und er versucht später, Tasks zu schicken. Ist Letzteres eingetreten, wird die Lifeline deaktiviert.

J_GLB startet auf jedem Place genau einen Worker. Es ist erlaubt, mehrere Activities auf demselben Place laufen zu lassen. Durch Synchronisation ist der Zugriff auf den lokalen Taskpool eines Workers deshalb abgesichert. Wie in Listing 1 zu sehen, wird dafür als Lock das Objekt des Workers verwendet. In Zeile 2 wird geprüft, ob der Worker noch Tasks hat. Dies geschieht über den *Empty*-Flag, der mit der Java-Datenstruktur `AtomicBoolean` ebenfalls geschützt wurde. Im Listing ist nicht erkennbar, dass einzig die Registrierung eines Thiefs beim Victim parallel zu dessen Berechnung erfolgen kann. Dafür wird nämlich lediglich der Zugriff auf die Liste der registrierten Thieves benötigt. Diese ist mittels der Java-Datenstruktur `ConcurrentLinkedQueue` vor Nebenläufigkeitsproblemen geschützt.

2.3 FT_GLB Framework

Aus J_GLB entstand FT_GLB. Dafür wurde eine Komponente des Java-Frameworks Hazelcast verwendet, das bereits für die Verbindung der JVMs in APGAS verwendet wurde. Bei der Komponente handelt es sich um die `IMap`-Datenstruktur, die Backups lokaler Taskpools und andere Informationen speichert. Die `IMap` erreicht Fehlertoleranz, indem sie Replikationen der enthaltenen Daten anfertigt und automatisch auf

```
while (tasks available) {
    while (local pool is not empty) {
        synchronized (worker object) {
            process up to n tasks;
            send tasks to recorded thieves;
        }
    }
    synchronized (worker object) {
        try to steal from up to w+z victims;
    }
}
```

Listing 1: Verhalten des J_GLB-Workers (übernommen von [14])

die Knoten des Clusters verteilt. Alle Places schreiben ihre Backups in eine globale `IMap`.

Es gibt vier Arten von Backups:

- *Regular*: wird in regelmäßigen Zeitabständen geschrieben
- *Final*: wird geschrieben, kurz bevor der Worker untätig wird
- *Restore*: wird geschrieben, nachdem ein Place abgestürzt ist
- *Steal*: wird bei einem Stehl-Versuch geschrieben

Im Fall eines Place-Absturzes werden die anderen Places benachrichtigt und übernehmen die übrig gebliebenen Tasks nach einem festgeschriebenen Protokoll.

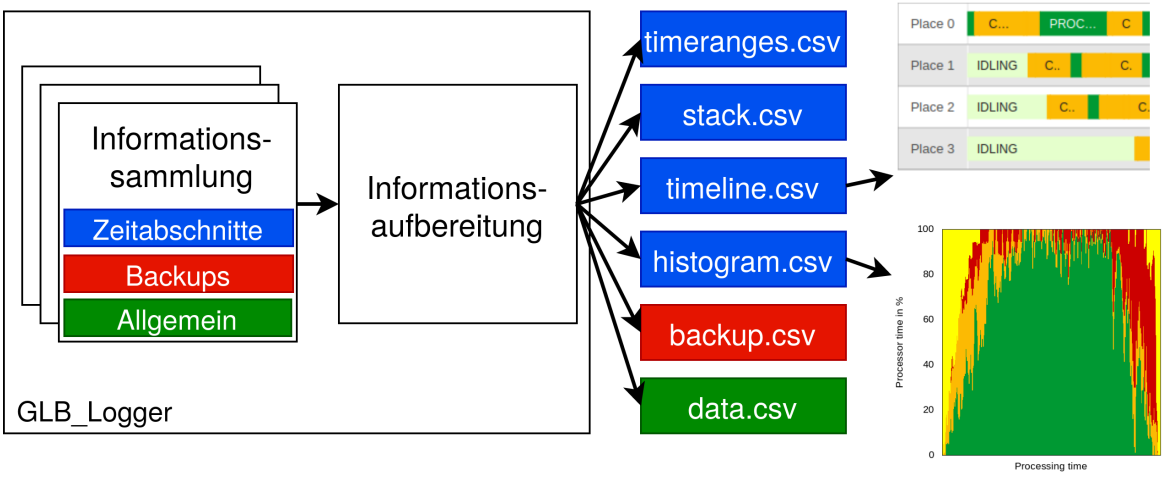


Abbildung 3: Konzept von GLB_Logger

3 Konzept

GLB_Logger besteht zum einen aus Klassen zum Aufzeichnen von Daten und zum anderen aus Datenstrukturen, um diese Daten zu speichern. Diese Komponenten befinden sich in einem Package der gemeinsamen Codebasis von J_GLB und FT_GLB, und können deshalb von beiden Frameworks benutzt werden. Die zentrale Klasse ist `GLBLogger`. Damit GLB_Logger Daten eines Frameworks aufzeichnet, wird auf jedem Worker ein globales Objekt dieser Klasse initialisiert. Wenn der Worker in einen neuen Zustand wechselt oder einen Stehl-Versuch tätigt, werden entsprechende Methoden des Objekts aufgerufen. Am Ende der Berechnung liefert GLB_Logger Ergebnisse, die zur Visualisierung weiterverwendet werden können.

Wie Abbildung 3 zeigt, funktioniert die Protokollierung von GLB_Logger in zwei Phasen. In der Informationssammlungs-Phase, während der Ausführung des Programms, zeichnen die `GLBLogger`-Objekte viele unterschiedliche Daten auf:

- *Daten der Zeitabschnitte*: Zustände, in denen sich ein Worker befindet, mit Start- und Endzeitpunkt.
- *Daten der Backups*: Backup-Typen mit Start- und Endzeitpunkt.
- *Allgemeine Daten*: Anzahl der Stehl-Versuche; Anzahl der vier Backup-Typen; Anzahl der gesendeten Nachrichten; etc.

Die Informationsaufbereitungs-Phase beginnt nach dem Abschluss des eigentlichen Programms. Zuerst werden die `GLBLogger`-Objekte der einzelnen Places gesammelt. Dann werden die Daten verarbeitet und in sechs Textdateien im CSV-Format ausgegeben (Abbildung 3 zeigt sechs Beispiele). Zu den oben genannten Daten der Backups und den allgemeinen Daten wird jeweils eine Datei ausgegeben (`backup.csv` und `data.csv`). Die Backup-Informationen aller Worker werden hintereinander in die Datei geschrieben, sodass jede Zeile in `backup.csv` die Daten eines Backups enthält. In `data.csv` werden zusätzlich zu den Informationen jedes Workers der Durchschnitt und die Summe berechnet. Die restlichen vier Dateien haben ihren Ursprung in den aufgezeichneten Daten der Zeitabschnitte:

Die Datei `timeranges.csv` stellt die Zeitabschnitte mit dem Zustand und der Dauer dar. Jedem Zeitabschnitt ist außerdem eine Identifikationsnummer (ID) zugeordnet, die sich ebenfalls in der Datei befindet. Die ID findet zusätzlich Verwendung in `stack.csv`, in der jeder ID eine Zeile zugeordnet ist. Jede Zeile der Datei enthält detailliertere Informationen zu einem Zeitabschnitt. Da diese Informationen für die Zeitabschnitte aller Worker hintereinander geschrieben wurden, findet sich zu jedem Zeitabschnitt eine Zeile in `stack.csv`. Der Anwender kann die ID aus `timeranges.csv` benutzen, um `stack.csv` weitere Informationen über den Zeitabschnitt zu entnehmen. In Abbildung 4 kann dieses Verfahren nachvollzogen werden. In Zeile 7 der `timeranges.csv`-Tabelle gibt es einen Zeitabschnitt mit der ID 1. Diese ID findet sich auch in der Tabelle `stack.csv` in der Spalte *TimeRange ID*. Die Klasse `TimeRange` modelliert einen Zeitabschnitt. Der Anwender erfährt den Stacktrace zum Zeitpunkt, als das Logging-Objekt zur Laufzeit aufgerufen wurde, um den betrachteten Zeitabschnitt aufzuzeichnen. Der Stacktrace enthält alle verschachtelten Methoden-Aufrufe seit dem Beginn eines Threads bis zur Generierung des Stacktrace [20]. Das erste Element ist dabei die zuletzt aufgerufene Methode. Der Worker antwortete demnach auf eine Stehl-Anfrage mit dem Senden von Loot. Über die Spalte *Comments* zeigt sich, dass es sich um einen Lifeline-Stehl-Versuch handelte.

Die bisherigen Dateien waren ausschließlich Informationsquellen. Die beiden letzten Dateien `timeline.csv` und `histogram.csv` verarbeiten die Daten der Zeitabschnit-

te, sodass sie als Input für die Visualisierungsprogramme verwendet werden können. Zum einen erwartet ein, zu Beginn der Bachelorarbeit vorliegendes, Skript die Datei `histogram.csv`. Das Skript basiert auf Gnuplot [5] und generiert das Histogramm. In Abbildung 4, ist beispielhaft der Inhalt von `histogram.csv` zu sehen. Die Datei besteht aus vier Spalten für jeden Worker-Zustand. Jede Zeile gibt für ein Intervall an, welchen prozentualen Anteil die Zustände auf allen Workern hatten. In Zeile 2 der Tabelle sieht man, dass ein Viertel der Worker den Zustand Processing hatten und sich die restlichen Worker im Zustand Idling befanden. Zeile 2 beschreibt das erste Intervall der Berechnungszeit, die folgenden Zeilen liefern Informationen über die weiteren Intervalle bis zur Beendigung des Rechenvorgangs. Eine Webseite, die die Google-Charts-Bibliothek [6] verwendet, benutzt die `timeline.csv`, um die Zeitleiste zu generieren. Diese enthält dieselben Informationen wie `timeranges.csv`, jedoch so formatiert, dass sie mit der Webseite kompatibel sind.

Die Dateien `backup.csv` und `data.csv` wurden bereits im oberen Textabschnitt erläutert.

Da die beschriebenen Funktionalitäten von GLB_Logger die Performance der Frameworks beeinflussen, wurden fünf Logging-Levels implementiert, die akkumulativ Funktionalität ergänzen:

- *0*: GLB_Logger ist deaktiviert.
- *1*: allgemeine Daten werden in der Konsole ausgegeben.
- *2*: allgemeine Daten werden in `data.csv` ausgegeben.
- *3*: alle sechs Textdateien werden generiert.
- *4*: Größe der Backups wird aufgezeichnet (Für die Begründung des Logging-Levels siehe Abschnitt 4.2).

Das Logging-Level wird als Kommandozeilenargument übergeben.

timeranges.csv

Place 0	Place 1	Place 2
PROCESSING	IDLING	IDLING
7.91	87.01	340.38
0	269	493
COMMUNICATION	COMMUNICATION	COMMUNICATION
9.83	13.78	25.23
1	270	494
COMMUNICATION	COMMUNICATION	COMMUNICATION
70.39	137.3	223.68
2	271	495

stack.csv

Place ID	Timerange ID	Comments	Stack	...
0	0		FTTimedGLB.FTTimedWorker.processStack ...	...
0	1	{steal=lifeline}	FTTimedGLB.FTTimedWorker.give ...	...
0	2	{backup-id=1}	FTTimedGLB.FTTimedWorker.writeBackup ...	...

timeline.csv

Place 0	0	PROCESSING	25	30	1	COMMUNICATION	30	39	...
Place 1	269	IDLING	25	106	270	COMMUNICATION	106	119	...
Place 2	493	IDLING	25	135	494	COMMUNICATION	135	138	...

histogram.csv

PROCESSING	COMMUNICATION	WAITING	IDLING
0.25	0	0	0.75
0.16520615283	0	0	0.83479384716

backup.csv

Place ID	Backup ID	Start (ms)	Type	#Tasks	Size (kB)	Writing Time
0	0	-17.77	INIT_BACKUP	0	37.69	15.81
0	1	39.76	STEAL_BACKUP	23	38.017	65.89
0	2	116.2	STEAL_BACKUP	20	38.017	25.29

data.csv

PlaceID	Tasks Processed	Total Steal Requests	Total Backups	...
0	866273	27	38	...
1	4145988	22	37	...
2	345581	23	35	...
	5357842	72	110	...
	1785947.33	24.0	36.67	...

Abbildung 4: Exemplarische Ausschnitte aus Ausgabe-Dateien

4 Protokollierung

4.1 Allgemeine Daten

4.1.1 Informationssammlung

Im Bereich der allgemeinen Daten zeichnet GLB_Logger diverse Informationen auf jedem Worker auf:

- *Zeit, bis das `GLBLogger`-Objekt benutzt wurde.*
- *Laufzeitinformationen des Taskpool-Patterns*: z.B. Zeit, die der Worker inaktiv war; Anzahl der Stehl-Versuche; Anzahl der verarbeiteten Tasks; Anzahl der gesendeten Nachrichten; u.v.m.
- *Informationen der Backups*: Anzahl der vier Backup-Typen; Zeit, die verwendet wurde, um Backups zu schreiben; Zeit, für den durchschnittlichen Schreibvorgang eines Backups.

Bei den Informationen handelt es sich um einfache `long`-Werte, die im `GLBLogger`-Objekt gespeichert werden.

4.1.2 Informationsaufbereitung

Nach dem Zusammentragen der `GLBLogger`-Objekte werden die Daten verarbeitet, um `data.csv` auszugeben.

Die Werte der Logger-Objekte werden nicht direkt in eine Zelle geschrieben, sondern meist mit simplen Formeln berechnet. Die Berechnung der einzelnen Zellen, sowie die Berechnung des Durchschnitts der Spalten erfordert viel redundanten Code. Es wurde deshalb nach einer Lösung gesucht, eine Spalte von `data.csv` mit einer generischen Klasse zu modellieren. Zu diesem Zweck wurde die Klasse `CsvCol` entwickelt. Mit dieser kann die Spaltenüberschrift und die Formel zur Berechnung einer Spalte definiert werden. Außerdem können mit ihr einfach die Summe und der Durchschnitt berechnet werden.

```
1 glbLogger.startRecordingBackup(BackupType.STEAL_BACKUP);
2 ...
3 int numOfTasks = this.queueWrapper.queue.size();
4 long size = ObjectSizeFetcher.getObjectSize(this.queueWrapper
    , this.logger.verbose);
5 ...
6 this.logger.endRecordingBackup(numOfTasks, size);
```

Listing 2: GLB_Logger zeichnet ein Backup auf

Durch `CsvCol` werden die Spalten einheitlich und auf eine einfach lesbare Art erstellt. Dadurch sind sie besser wartbar und in Zukunft einfacher zu erweitern. Die Implementierung ist zwar langsamer; dies hat jedoch keine Auswirkung auf Messergebnisse, da diese vor der Informationsaufbereitungsphase erhoben werden.

4.2 Backups

Das Aufzeichnen eines Backups, geschieht wie in Listing 2 zu sehen. In Zeile 1 wird `startRecordingBackup` aufgerufen, sobald der Schreib-Vorgang beginnt. Im Hintergrund legt `GLBLogger` ein Objekt der `Backup`-Klasse an und speichert es in einer Liste. Darin wird die Start-Zeit des Schreib-Vorgangs eingetragen und der Typ des Backups über die Enumeration gesetzt. Am Ende des Schreibvorgangs wird in Zeile 6 `endRecordingBackup` aufgerufen und dadurch der Endzeitpunkt des Schreibvorgangs gespeichert. Dafür wird auf das letzte `Backup`-Objekt der Liste zugegriffen und die Endzeit entsprechend gesetzt. Außerdem werden die in der Zwischenzeit berechnete Anzahl der Tasks, sowie die Größe des Backups übergeben. Das Objekt `queue` ist in FT_GLB und J_GLB der lokale Taskpool eines Workers.

Wie in Abschnitt 2.2 beschrieben, ist der Zugriff auf den lokalen Taskpool über einen Lock geschützt. Da das Schreiben eines Backups diesen Zugriff benötigt, kann währenddessen kein anderes Backup geschrieben werden. Deshalb kann bei dem Aufruf von `stopRecordingBackup` angenommen werden, dass bisher kein anderes Backup angefangen wurde. Das letzte `Backup`-Objekt der Liste der aufgezeichneten Backups ist damit auch immer das letzte aufgezeichnete Backup.

Die Berechnung der Backup-Größe geschieht mit der Utility-Klasse `ObjektSize-`

`Fetcher`. In Java ist die Berechnung der Größe eines Objekts nicht trivial. Java stellt zwar eine entsprechend Funktion in der `Instrumentation`-Klasse bereit, die diese Aufgabe übernehmen kann. Dafür muss aber zunächst eine Jar-Datei generiert werden, was die bisherige Nutzung der Frameworks verändert hätte [2]. Unsere Implementierung verwendet einen Byte-Stream, mit der die Größe des Backups bestimmt wird. Das Objekt wird auf den Byte-Stream geschrieben, dabei serialisiert und danach ausgegeben, wie viele Bytes geschrieben wurden. Diese Lösung wurde übernommen von [9] und ist sehr performance-aufwendig. Außerdem ist sie ungenau, da die Größe eines serialisierten Objekts nicht gleich der Größe des Objekts auf dem Heap ist [11]. Sie wurde dennoch implementiert, weil sie einfacher zu implementieren war als alternative Lösungen. Außerdem liefert unsere Lösung zumindest einen guten ersten Eindruck über die Größe der Backups. Der hohe Performance-Aufwand ist weiterhin ein Problem, weshalb das Berechnen der Backup-Größe nur im höchsten Logging-Level aktiviert ist.

Die Ausgabe von `backup.csv` ist trivial; die Backup-Informationen aller Worker werden Zeile für Zeile in die Datei geschrieben.

4.3 Zeitabschnitte

4.3.1 Communication-Zeitabschnitte in der Informationssammlungsphase

Zum besseren Verständnis wird zunächst nur die Aufzeichnung des Zustandstyps, sowie des Start- und Endzeitpunkts der Zeitabschnitte beschrieben. Wie die anderen Informationen aufgezeichnet werden, behandelt der nächste Abschnitt.

Es wird im Folgenden außerdem angenommen, dass bei der Aufnahme Objekte der Klasse `TimeRange` einer Liste hinzugefügt werden. Zu Beginn des Zeitabschnitts wird das Objekt mit der Startzeit initialisiert und am Ende des Zeitabschnitts wird ihm eine Endzeit zugewiesen.

Wie in Abschnitt 2.2 beschrieben, können, während der Berechnung eines Workers, ausschließlich Stehl-Anfragen auftreten. Die Stehl-Anfragen gehören zum Zustandstyp Communication. Zeitabschnitte von diesem Typ nehmen bei der Aufnahme deshalb eine besondere Stellung ein.

Zum einen kann sich der Worker dadurch in zwei verschiedenen Zuständen gleichzeitig befinden. Es ist jedoch weitaus einfacher, sowohl für die Ausgabe der Textdateien, als auch für die spätere visuelle Darstellung, wenn sich der Worker immer nur in einem Zustand befindet. Außerdem sollten wir bedenken, dass die Abschnitte der Stehl-Anfragen sehr viel kürzer sind, als die der anderen Zeitabschnitte. Deshalb ist es von geringem Nachteil, wenn wir davon ausgehen, dass der Worker am Anfang der Stehl-Anfrage in den Zustand Communication wechselt und danach wieder in den vorherigen Zustand zurückkehrt.

Zum anderen bedarf das Setzen des Endzeitpunkts deshalb einer genaueren Betrachtung. Während der Worker Tasks verarbeitet und sich damit im Zustand Processing befindet, können eine oder mehrere Stehl-Anfragen eintreffen, in denen der Worker in den Zustand Communication wechselt. Am Ende des Processing-Abschnitts könnte dadurch das letzte Element der Liste auch immer ein, in der Zwischenzeit gestarteter, Communication-Abschnitt sein. Um einen Abschnitt eindeutig zu beenden, müsste von der Logger-Methode zum Starten eines Zeitabschnitts bis zur Methode zum Beenden eine eindeutige ID weitergegeben werden. Dann könnte über diese ID der Zeitabschnitt in der Liste gefunden und der Endzeitpunkt gesetzt werden. Könnte man stattdessen davon ausgehen, dass seit einem angefangenen Zeitabschnitt kein anderer begonnen wurde, wäre der gesuchte Zeitabschnitt immer das letzte Element der Liste. Damit erschweren Communication-Zeitabschnitte die Zuweisung der Endzeitpunkte zu angefangenen Zeitabschnitten. Um zumindest die Aufnahme der Zeitabschnitte der Zustände Idling, Waiting und Processing einfach zu gestalten, wurde sich dafür entschieden, die Communication-Abschnitte getrennt von den anderen aufzuzeichnen. Zeitabschnitte vom Typ Communication müssen, wie im folgenden Abschnitten deutlich wird, weiterhin mit IDs beendet werden; alle anderen Zeitabschnitte können automatisch mit dem Beginn des nächsten Zeitabschnitts beendet werden.

Dafür werden zwei getrennte Listen gepflegt, denen Objekte der Klasse `TimeRange` hinzugefügt werden. Eine Liste speichert die Zeitabschnitte vom Typ Communication, die andere Liste speichert die Zeitabschnitte der Zustandstypen Idling, Processing und Waiting.

```
1 logger.startRecordingTimeRange(TaskType.PROCESSING, queue.
    size());
2 ...
3 logger.startRecordingTimeRange(TaskType.IDLING, -1);
```

Listing 3: GLB_Logger zeichnet mehrere Zeitabschnitte auf

In Listing 3 sieht man das Aufnehmen mehrerer Zeitabschnitte, die nicht vom Typ Communication sind. Wie man sieht, werden diese lediglich gestartet mit `startRecordingTimeRange`. In dieser Methode wird ein `TimeRange`-Objekt angelegt mit der gemessenen Startzeit. Der angefangene Zeitabschnitt muss nicht beendet werden. Stattdessen wird der nächste Abschnitt mit derselben Funktion begonnen und in Folge dessen, der letzte Zeitabschnitt automatisch mit der Startzeit des neuen Abschnitts beendet. Wie oben beschrieben treten Abschnitte vom Zustandstyp Idling, Processing und Waiting nur sequentiell auf. Deshalb ist der letzte Zeitabschnitt immer der Zeitabschnitt aus Zeile 1 und kann ohne Nebenläufigkeitsprobleme beendet werden.

Das zweite Argument, das der Funktion übergeben wird, ist eine zustandsspezifische Information, die näher im nächsten Abschnitt beschrieben wird. Im Fall des Processing-Zeitabschnitts sind das die Anzahl der Tasks im lokalen Taskpool; für Idling-Zeitabschnitte existieren keine solche Informationen, weshalb *-1* übergeben wird.

In Listing 4 wird dagegen ein Communication-Zeitabschnitt aufgezeichnet. Beim Aufrufen von `startRecordingTimeRange` in Zeile 1 wird eine ID ausgegeben, die dem Zeitabschnitt eindeutig zugeordnet ist (die Funktion `startRecordingTimeRange` liefert immer eine ID, sie muss jedoch nur im Fall von Communication-Zeitabschnitten verwendet werden). Darüber kann der Zeitabschnitt in Zeile 3 beendet werden. Obwohl Communication-Zeitabschnitte in einer separaten Liste aufgezeichnet werden, benötigen sie weiterhin IDs. Dies liegt daran, dass auch Stehl-Anfragen von verschiedenen Workern gleichzeitig eintreffen können. Der letzte Zeitabschnitt der Liste ist somit nicht zwingend der Zeitabschnitt, der in Zeile 1 erstellt wurde. Bei der spezifischen Zustandsinformation handelt es sich diesmal um die ID des Places, mit dem der Worker kommunizierte. Diese wird über die Variable `p` übergeben, die außerhalb des

```
1 int id = logger.startRecordingTimeRange(TaskType.
    COMMUNICATION, p);
2 ...
3 logger.endRecordedTimeRange(id);
```

Listing 4: GLB_Logger zeichnet einen Communication-Zeitabschnitt auf

Listings initialisiert wurde.

Wie am Anfang des Abschnitts beschrieben, hätte die Beschränkung auf eine Liste dazu geführt, dass die Idling-, Waiting- und Processing-Zeitabschnitte ebenfalls über IDs beendet werden müssen. Dies würde die Verwendung von `GLBLogger` komplizierter machen, da die IDs über weite Code-Stücke weitergegeben werden müssten. Durch unsere Implementierung kann GLB_Logger also weitaus einfacher verwendet werden. Dafür ist die Verarbeitung der gesammelten Informationen in der Informationsaufbereitungsphase aufwendiger. Die Listen müssen dann wieder zusammengeführt werden, was in Abschnitt 5.1 thematisiert wird.

4.3.2 Stacktrace und zustandsspezifische Informationen

Momentan werden über einen Zeitabschnitt lediglich der Zustand, sowie Start- und Endzeitpunkt aufgezeichnet. Das Ziel sollte jedoch sein, dass der Benutzer mit Hilfe der Informationen eines Zeitabschnitts nachvollziehen kann, wo sich der Worker in einem der GLB-Frameworks befindet und was er dort tut. Beispielsweise könnte sich ein Zeitabschnitt mit dem Zustand Communication an vielen Stellen des Frameworks befinden. Er könnte ein Backup schreiben, eine Stehl-Anfrage senden oder beantworten. Es sollten deshalb zu jedem Zeitabschnitt zusätzliche Informationen gespeichert werden, die dem Benutzer dieses Verhalten nachvollziehbar machen.

Beim Aufruf des Logger-Objekts in Zeile 1 der Listings 3 und 4 wird im Hintergrund der Stacktrace gespeichert. Dafür wird mittels `Thread.currentThread().getStackTrace()` der Stacktrace des aktuell laufenden Threads ermittelt. Damit kann der Benutzer später in `stack.csv` erkennen, an welcher Codestelle sich der Worker zu Beginn eines Zeitabschnitts befand. Beispielsweise könnte sich der Worker in einem Zeitabschnitt vom Typ Communication am Anfang einer Funktion zum Verteilen

von Loot befinden. Es wäre nun in diesem Fall interessant, an welchen Worker das Loot gesendet wird. Deshalb werden zu jedem Zustandstyp spezifische Informationen gespeichert:

- *Communication*: Place-ID des Workers mit dem kommuniziert wird (-1, wenn Backup getätigt wird).
- *Processing*: Größe des lokalen Taskpools.
- *Waiting*: Place-ID des Workers, auf den gewartet wird.
- *Idling*: keine spezifischen Informationen vergeben.

In Zeile 1 in Listing 3 werden, dem Zustandstyp entsprechende, Informationen übergeben. Der Benutzer kann nun die Stelle im Code finden, in der der Worker Tasks prozessierte und wie viele Tasks zu diesem Zeitpunkt im lokalen Taskpool waren.

Im Fall der Communication-Zeitabschnitte sind die Informationen noch ungenügend. Es kann zwar herausgefunden werden, mit welchem Worker kommuniziert wird. Da Stehl-Versuche ein zentraler Bestandteil der Frameworks sind, sollten darüber weitere Informationen aufgezeichnet werden können. An dieser Stelle entschieden wir uns dafür, Anwendern des GLB_Loggers die Möglichkeit zu geben, beliebig viele Informationspaare einem Zeitabschnitt hinzuzufügen. Diese werden in einer `Map` des `TimeRange`-Objekts gespeichert. Im Fall eines Stehl-Versuchs kann dem Zeitabschnitt die Information *steal=lifeline* und *steal-status=success* hinzugefügt werden. Dies würde bedeuten, dass es sich um einen Lifeline-Stehl-Versuch handelte, der erfolgreich durchgeführt wurde. Auf diese Weise können auch in Zukunft einfach spezifische Informationen einem Zeitabschnitt hinzugefügt werden.

Alle in diesem Abschnitt genannten Informationen werden in `stack.csv` Zeile für Zeile ausgegeben. Sie werden, wie in Abschnitt 3 beschrieben, verwendet, um detailliertere Informationen zur Zeitleiste oder zur Datei `timeranges.csv` zu erhalten.

4.4 Synchronisation

Da Communication-Abschnitte während der Berechnung auftreten, können mehrere Threads Zugriff auf die Methoden des Loggers einfordern. Damit besteht die Gefahr von Nebenläufigkeitsproblemen, beispielsweise bei der Vergabe der IDs der Zeitabschnitte. Alle Funktionen, die die Variablen verändern, die während eines Communication-Zeitabschnitts verändert werden könnten, mussten deshalb geschützt werden. Dafür wurden die betroffenen Funktionen der `GLBLogger`-Klasse mit `synchronized`-Blöcken geschützt. Da die Funktionen des Loggers nur wenig Zeit in Anspruch nehmen, haben die `synchronized`-Blöcke geringen Einfluss auf die Performance. Durch die Einstellung eines entsprechenden Logging-Levels kann das Betreten der `synchronized`-Blöcke bei Bedarf verhindert werden.

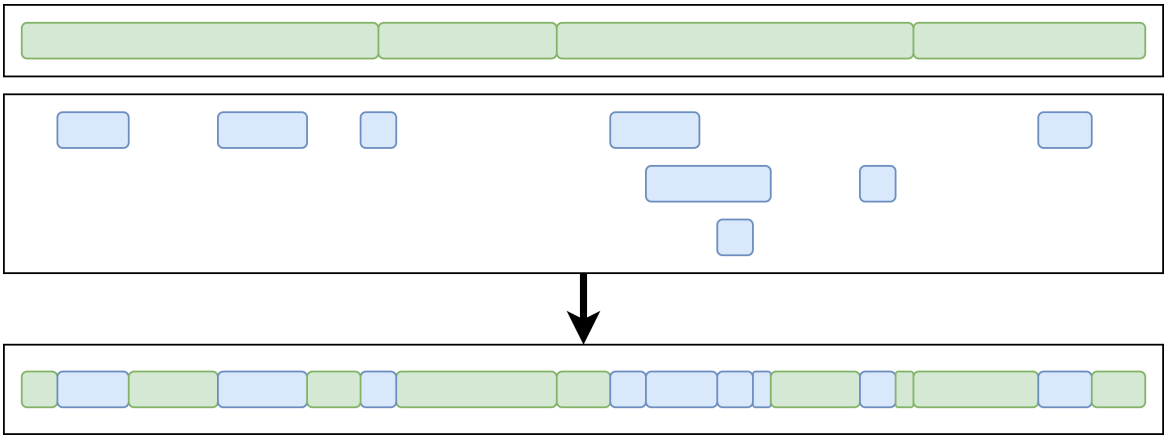


Abbildung 5: Von Oben: 1. Rechteck: Waiting-, Idling- und Processing-Zeitabschnitte, 2. Rechteck: Communication-Zeitabschnitte, 3. Rechteck: Ergebnisliste

5 Visualisierung

5.1 Zeitleiste

Nach der Informationssammlungsphase gibt es eine Liste der Communication-Zeitabschnitte und eine Liste der restlichen Zeitabschnitte. Bevor sie in Textdateien ausgegeben werden können, müssen die beiden zu einer Liste zusammengeführt werden. Dafür wurde ein Algorithmus entwickelt, der im Folgenden vorgestellt wird.

5.1.1 Flatten-Algorithmus

Die Situation nach der Informationssammlungsphase zeigt sich in Abbildung 5. Die Balken sind Zeitabschnitte, deren Länge die Dauer des Zeitabschnitts darstellt. Anfang und Ende des Balkens im Bild stellen jeweils den Start- bzw. Endzeitpunkt des Zeitabschnitts dar.

Im ersten Rechteck ist die Liste der Waiting-, Idling- und Processing-Zeitabschnitte. Durch die im letzten Abschnitt beschriebene Implementierung des Loggers gehen die Zeitabschnitte fortlaufend ineinander über. Im Rechteck darunter ist die Liste der Communication-Zeitabschnitte. Diese wurden beim Aufzeichnen explizit beendet. Folglich sind zwischen den Abschnitten Lücken. Da mehrere Stehl-Anfragen zugleich eintreffen können, sind die Zeitabschnitte teilweise überlappend.

Im letzten Rechteck ist die Zielliste, die durch den Algorithmus erhalten wird und

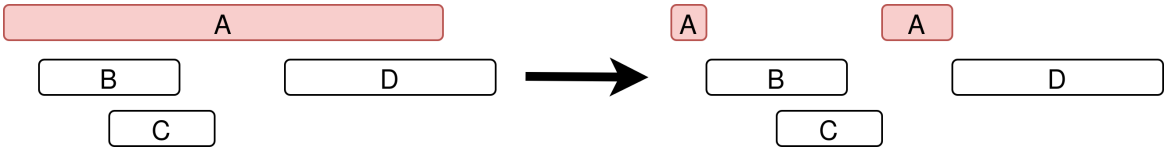


Abbildung 6: Input und Output des Flatten-Algorithmus

die beiden Listen vereint. Es sollte daran erinnert werden, dass die Zielliste als Basis für die Ausgabe jener Textdateien dient, die später für die Visualisierung verwendet werden. Dadurch muss die Zielliste bestimmte Kriterien erfüllen. Am wichtigsten ist, dass sie keine Zeitabschnitte enthält, die sich überlappen. Diese Eigenschaft ist wichtig für die visuelle Darstellung, da sonst die Zeitabschnitte in der Zeitleiste nicht in einer Zeile dargestellt werden könnten. Erfüllt die Zielliste diese Anforderung, können die Dateien `timeline.csv` und `timeranges.csv` auf triviale Weise ausgegeben werden. In `timeline.csv` wird die Zielliste eines Workers in je einer Zeile ausgegeben. In `timeranges.csv` werden die Zeitabschnitte eines Workers in einer Spalte ausgegeben, damit sie besser in Tabellen-Programmen geöffnet werden können. Auch in diesem Fall kann die Zielliste unverändert ausgegeben werden.

Die Tatsache, dass die Communication-Zeitabschnitte bereits überlappen, verkompliziert das Problem. Deshalb wurde sich dafür entschieden statt einem, zwei Algorithmen zu entwerfen. Die Aufgabe des ersten Algorithmus ist, die Communication-Abschnitte zusammenzufassen, sodass sie sich nicht mehr überlappen. Dieser wird im Folgenden *Flatten-Algorithmus* genannt. Der zweite Algorithmus kann dann von zwei Listen ausgehen, in denen sich keine Zeitabschnitte überlappen. Diesen nennen wir *Merge-Algorithmus*; er erzeugt schließlich die Zielliste und wird im nächsten Abschnitt behandelt.

In Abbildung 6 sieht man an einem Beispiel, wie der Flatten-Algorithmus funktioniert. Aus Gründen der Übersichtlichkeit wird aber zunächst nur der rot markierte Zeitabschnitt A betrachtet. Dieser wird in zwei kleinere Zeitabschnitte aufgeteilt und überlappt damit mit keinem anderen Zeitabschnitt.

Der Algorithmus sucht also die Lücken des betrachteten Zeitabschnitts bezüglich der folgenden Zeitabschnitte. Der Flatten-Algorithmus erwartet, dass die Zeitabschnitte nach ihrer Anfangszeit geordnet sind. Außerdem wird nur der betrachtete Zeitab-

```
foreach (timeRange in communicationTimeRanges) {
    foreach (successor of timeRange) {
        Process timeRange and next successor timeRange.
    }
}
```

Listing 5: Flatten-Algorithmus

schnitt verändert, während Zeitabschnitte mit späteren Anfangszeiten vom momentanen Zeitabschnitt unangetastet bleiben. Deshalb müssen die Zeitabschnitte vor A nicht betrachtet werden, da sie keinen Einfluss auf ihn haben können. Damit muss nur dafür gesorgt werden, dass A nicht mit den folgenden Zeitabschnitten überlappt. Iteriert man auf diese Weise über alle Zeitabschnitte, kann man das Problem lösen. In Listing 5, ist das Vorgehen des Flatten-Algorithmus in Pseudo-Code zu sehen. Wie man sieht, iteriert er in zwei Schleifen einmal über alle Zeitabschnitte und jeweils über deren Nachfolger.

Man kann den Vorgang beschleunigen, indem man ein Abbruchkriterium hinzufügt. Bei den folgenden Zeitabschnitten müssen nur jene betrachtet werden, mit denen der betrachtete Abschnitt überlappt; da die Zeitabschnitte geordnet sind, kann der Algorithmus die innere Schleife abbrechen, wenn er einen Abschnitt findet, der sich nicht mit dem momentan Betrachteten überschneidet.

Die innere Schleife des Algorithmus wird in Abbildung 7 beispielhaft gezeigt. Wieder befindet sich Zeitabschnitt A in der äußeren Schleife. Der Algorithmus iteriert also über die folgenden Zeitabschnitte, die jeweils blau markiert sind. Im ersten Bild findet er eine Lücke zwischen A und B. Diese wird der Zielliste hinzugefügt, die durch das gestrichelte Rechteck dargestellt ist. Da der Algorithmus folgende Abschnitte nicht verändern darf, kann er B überspringen und setzt seinen neuen Startpunkt auf das Ende von B. Bei C findet er keine Lücke und muss seinen Startpunkt wieder anpassen. Bei D findet er eine Lücke und fügt sie der Zielliste hinzu. Da es keinen weiteren Abschnitt gibt, geht der Algorithmus zum nächsten Element B über.

5.1.2 Merge-Algorithmus

Nun können die beiden Listen als Input für den Merge-Algorithmus verwendet werden.

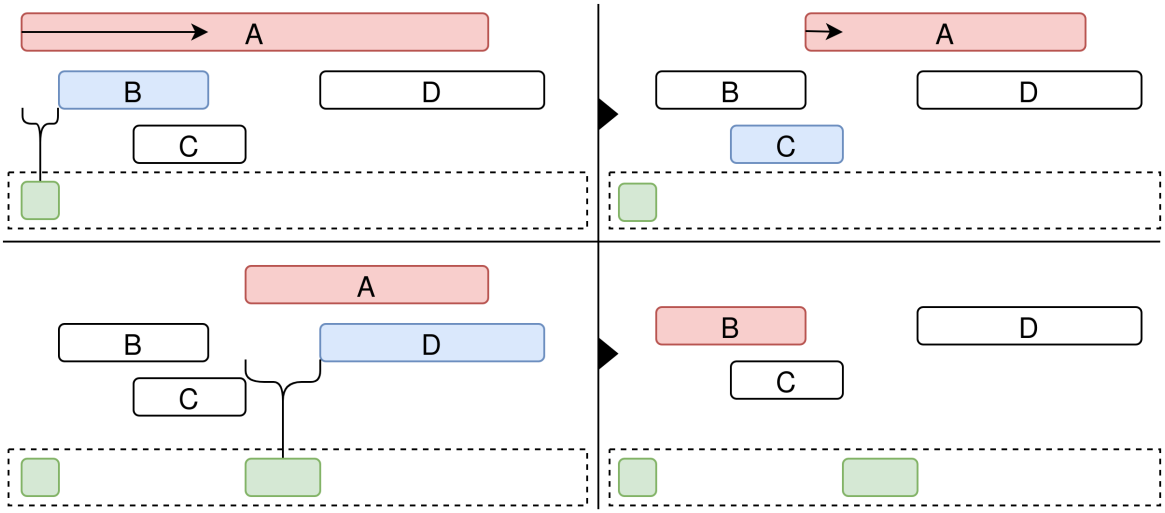


Abbildung 7: Flatten-Algorithmus mit Beispiel-Eingabe

Wie in Abbildung 5 zu sehen, werden die Communication-Zeitabschnitte zwar untereinander durch den Flatten-Algorithmus verändert; danach werden sie jedoch ohne weitere Anpassungen in die Zielliste übernommen. Die Zeitabschnitte der anderen Zustandstypen werden dagegen verkleinert und aufgeteilt.

Die Communication-Zeitabschnitte werden bevorzugt, weil sie im Gegensatz zu den anderen Zeitabschnitten exakt aufgezeichnet werden: Sie werden explizit beendet und beschreiben damit genau den Zeitraum, in dem sie im Worker auftreten. Die Zeitabschnitte der anderen Zustandstypen werden automatisch beendet, sobald ein neuer Zeitabschnitt vom Typ Waiting, Idling oder Processing begonnen wird. Sie werden jedoch nicht beendet, wenn der Worker in den Zustand Communication wechselt. Wenn beispielsweise der Worker zuerst im Zustand Processing ist und im Anschluss ein Regular-Backup schreibt, wird der Processing-Abschnitt nicht beendet. Dies geschieht erst, sobald der Worker keine Tasks mehr hat. Dann tätigt er eine Stehl-Anfrage und wechselt in den Zustand Waiting. Dadurch umfasst ein Zeitabschnitt von Typ Idling, Waiting oder Processing potenziell Zeiträume, in denen sich der Worker im Zustand Communication befand.

In Abbildung 8 sehen wir das gewünschte Verhalten des Algorithmus an einem Beispiel. Der Merge-Algorithmus übernimmt die Communication-Zeitabschnitte unverändert in die Zielliste. Die anderen Zeitabschnitte passt er so an, dass sie sich nicht

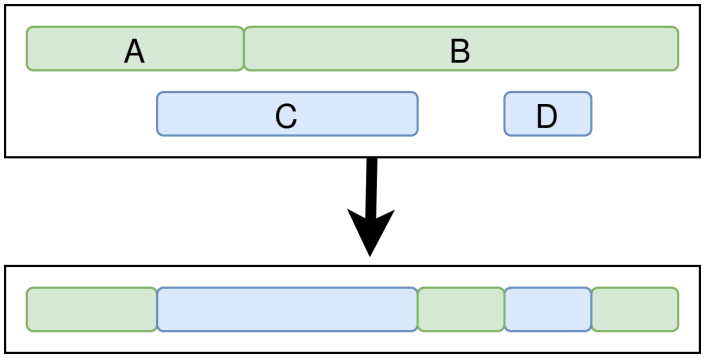


Abbildung 8: Input und Output des Merge-Algorithmus (blaue Rechtecke: Communication-Zeitabschnitte, grüne Rechtecke: restliche Zeitabschnitte)

mit den Communication-Abschnitten überschneiden.

Wie in Listing 6 zu sehen, iteriert der Algorithmus über beide Listen. Während die Schleife über die Communication-Abschnitte iteriert, werden die Zeitabschnitte der anderen Zustandstypen mit einem `Iterator`-Objekt durchlaufen. Im Gegensatz zum Flatten-Algorithmus werden damit beide Listen nur einmal durchlaufen. Um den Grund für diesen Unterschied zu verstehen, betrachten wir zunächst einen Communication-Abschnitt in der Schleife des Algorithmus. Dieser muss alle Idling-, Waiting- oder Processing-Zeitabschnitte verarbeiten, mit denen er sich überschneidet. Der erste Zeitabschnitt, der sich mit dem aktuellen Communication-Abschnitt überlappen könnte, ist der letzte Abschnitt, den der vorherige Communication-Abschnitt bearbeitete. Da beide Listen frei von Überlappungen sind, kann sich keiner der Zeitabschnitte davor mit dem aktuellen Communication-Abschnitt überschneiden. In Abbildung 8 ist B der letzte, von C bearbeitete, Zeitabschnitt. In dieser Situation überschneidet sich C mit A und B, sowie D mit B. Dass sich D zusätzlich mit A überschneidet, ist offensichtlich nicht möglich. Deshalb muss A von D nicht betrachtet werden. Der letzte Abschnitt, der vom aktuellen Communication-Abschnitt betrachtet wird, ist der erste, den der nächste Communication-Abschnitt verarbeitet. Damit reicht jeweils eine Iteration über beide Listen aus. Dafür zentral ist, dass beide Listen frei von Überlappungen und geordnet vorliegen.

In Abbildung 9 sieht man die innere Schleife des Algorithmus an einem Beispiel. Die Zeitabschnitte mit breiteren Rändern sind jeweils das Communication-Objekt in

```
nonComIterator = nonCommunicationTimeRanges.iterator
Get next nonComTimeRange from nonComIterator.

foreach (comTimeRange in communicationTimeRanges) {
    Process comTimeRange and current nonComTimeRange.
    Get next nonComTimeRange from nonComIterator if necessary
        .
    Add comTimeRange to result.
}

Add remaining nonComTimeRanges to result.
```

Listing 6: Merge-Algorithmus

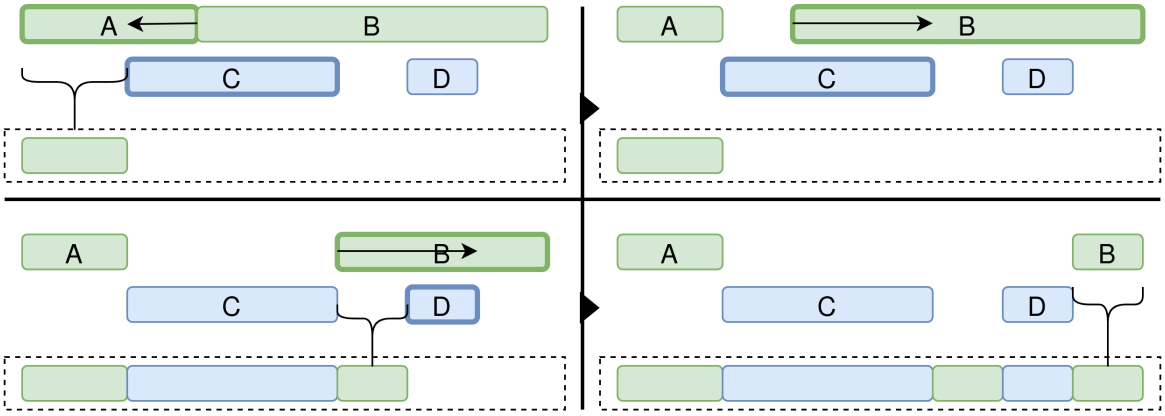


Abbildung 9: Merge-Algorithmus mit Beispiel-Eingabe

der äußeren Schleife und das momentane Objekt aus dem `Iterator`.

Als erstes werden die Zeitabschnitte A und C betrachtet. A überschneidet sich mit C. Deshalb wird A verkleinert und zur Zielliste hinzugefügt. B überschneidet sich ebenfalls mit C und wird deshalb verkleinert. Damit keine Lücken doppelt hinzugefügt werden, geschieht dies nur bei Zeitabschnitten, die kleiner sind als der betrachtete Communication-Abschnitt. Deshalb wird B zunächst nicht zur Zielliste hinzugefügt. Da der Endzeitpunkt über C hinausgeht, wird er zur Zielliste hinzugefügt und als nächstes Zeitabschnitt D betrachtet. In Folge dessen wird B angepasst und an die Zielliste angehängt. Auch D ist damit abgeschlossen und wird zur Zielliste hinzugefügt. Die Schleife der Communication-Abschnitte ist damit beendet. Danach werden die restlichen Abschnitte der Idling-, Processing- und Waiting-Zeitabschnitte in die Zielliste eingefügt. So gelangt auch B in die Zielliste.

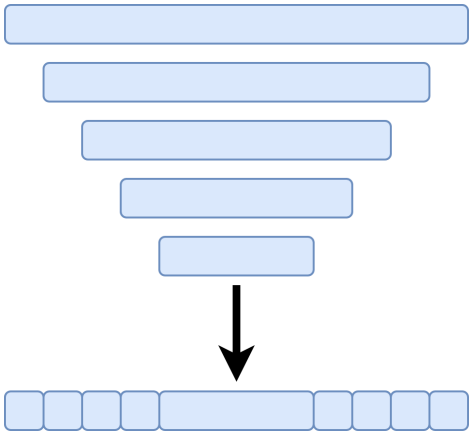


Abbildung 10: Worst-Case Eingabe des Flatten-Algorithmus

5.1.3 Laufzeitanalyse

Im nachfolgenden soll die Laufzeit des Flatten- und Merge-Algorithmus analysiert und bewertet werden.

Die betrachtete Eingabegröße n des Flatten-Algorithmus ist die Länge der Liste der Communication-Zeitabschnitte. Eine Beispieleingabe für den schlechtesten Fall sehen wir in Abbildung 10. Der Flatten-Algorithmus iteriert einmal über alle Zeitabschnitte der Liste. Da ein frühzeitiger Abbruch nicht möglich ist, iteriert er bei jeder Iteration noch einmal über alle nachfolgenden Abschnitte. Damit durchläuft der erste Zeitabschnitt n Iterationen. Diese Zahl nimmt bei jedem folgenden Zeitabschnitt um eins ab. Wie in der nachfolgenden Formel zu sehen, hat der Flatten-Algorithmus daher eine Laufzeit von $O(n^2)$:

$$n + (n-1) + \cdots + 1 = \sum_{k=1}^{n} k \overset{\text{Gauß}}{=} \frac{n \cdot (n+1)}{2} = \frac{n^2 + n}{2}$$

In der Praxis ist dies kein großer Nachteil, weil das schlechteste Szenario nur selten auftritt. Die Communication-Abschnitte sind sehr kurz; Überlappungen sind daher unwahrscheinlich. In den meisten Fällen wird ein Communication-Abschnitt also keine Überlappungen haben und der Algorithmus kann sofort zum nächsten Abschnitt übergehen. Realistischer ist also der Best-Case mit einer linearen Laufzeit von $\Theta(n)$. Daher ist der Flatten-Algorithmus für den gegebenen Anwendungsfall ausreichend.

Der Merge-Algorithmus erhält die Liste der Communication-Zeitabschnitte als Eingabe. Deshalb ist es interessant, wie dessen Größe durch den Flatten-Algorithmus verändert wird. In Abbildung 10 sieht man auch ein Beispiel für den Worst-Case im Bezug auf die Größe der Ausgabe-Liste. Betrachtet man die Eingabe-Zeitabschnitte von oben nach unten, fällt auf, dass jeder Zeitabschnitt zwei Teilstücke zur Ausgabe-Liste beiträgt. Eine Ausnahme bildet der mittlere Zeitabschnitt, der komplett übernommen wird. Dies ergibt im Worst-Case offensichtlich eine Größe der Ausgabe-Liste von $2n - 1$. Wie oben diskutiert, ist der Fall, in dem sich die Abschnitte nicht überschneiden, wahrscheinlicher. Dann bleibt die Größe der Liste konstant.

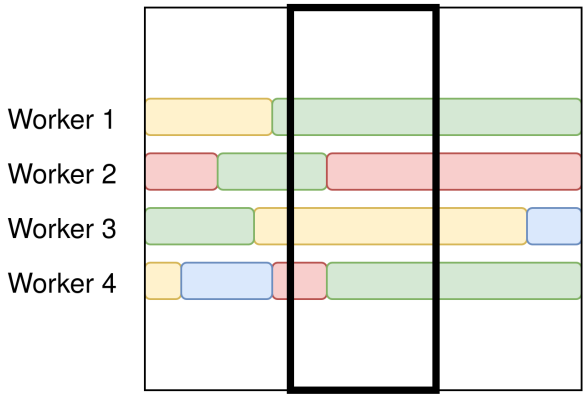
Im Fall des Merge-Algorithmus sind die Eingabegrößen jeweils die Länge der Liste der Communication-Zeitabschnitte $2n - 1$ und die Länge der Liste der Zeitabschnitte anderer Zustandstypen m. Wie im vorherigen Abschnitt beschrieben, durchläuft der Algorithmus beide Listen genau einmal und führt Operationen von konstanter Laufzeit auf diesen aus. Es handelt sich deshalb offensichtlich um eine lineare Laufzeit von $O(n + m)$.

5.2 Histogramm

Die Datei `histogram.csv` basiert auf den Zeitabschnitt-Listen aller Worker, die mit den, in den letzten Abschnitten beschriebenen, Algorithmen berechnet werden. Abbildung 11 zeigt an einem Beispiel, wie die Generierung funktioniert.

Die Listen werden in eine Anzahl von Intervallen aufgeteilt, die über die Kommandozeile übergeben wird. Im Beispiel werden sie in drei Abschnitte unterteilt. Die Intervalle werden der Reihe nach berechnet und in `data.csv` geschrieben. Das Gnuplot-Skript, welches das Histogramm generiert, zeichnet für jeden Eintrag in `data.csv` einen Balken im Histogramm. Damit kann über die Anzahl der Intervalle die Feinheit des Histogramms eingestellt werden. Zur Vereinfachung wird im Beispiel lediglich Interval 2 berechnet.

In einem Intervall wird die prozentuale Anzahl eines Zustandstyps auf allen Workern berechnet. Dieser Anteil bezieht sich nur auf jene Teile der Zeitabschnitte, die sich im Bereich des Intervalls befinden. Zur Berechnung werden die Dauer dieser Teil-



PROCESSING	COMMUNICATION	WAITING	IDLING
0.5	0.0	0.25	0.25

Abbildung 11: Berechnung von `histogram.csv`

Zeitabschnitte eines Zustandstyps von allen Workern addiert. Für den Zustandstyp Processing wird der Durchschnitt über die folgende Formel berechnet:

$$durchschnittProcessing = \frac{processingDauerImIntervall}{(dauerDesIntervalls \cdot anzahlDerWorker)}$$

Im Fall der Processing-Zeitabschnitte ergibt sich im Beispiel ein Anteil von fünfzig Prozent. Für alle anderen Zustandstypen wird die Formel entsprechend angepasst. Da kein Communication-Abschnitt im Intervall vorkommt, wird ein Anteil von null Prozent berechnet. Der Rest entfällt auf Waiting- und Idling-Zeitabschnitte mit jeweils einem Anteil von fünfundzwanzig Prozent.

Abschnitt	Benchmark	Parameter	Places	Berechnungszeit
6.1 - 6.2	UTS	d = 15, b = 4, n = 511, s = 10, r = 19, w = 3, l = 4, m = 1024, z = 3, v = 3	60	FT_GLB: 17.85 s J_GLB: 14.33 s
6.1 - 6.2	BC	N = 2^{16}, g = 511, s = 10, seed = 2, l = 4, m = 1024, z = 3, a = 0.55, b = 0.1, c = 0.1, w = 3, d = 0.25, v = 3	60	FT_GLB: 16.45 s J_GLB: 14.76 s
6.3	UTS (J_GLB)	d = 14, b = 4, **n = 5 bzw. n = 1000**, r = 19, w = 3, l = 4, m = 1024, z = 3, v = 3	60	n = 5: 5.17 s n = 1000: 4.62 s
6.4.1	UTS (FT_GLB)	d = 12, b = 4, n = 511, s = 10, r = 19, w = 3, l = 4, m = 1024, z = 3, **v = 4**	4	5.20 s
6.4.2	BC (FT_GLB)	N = 2^{14}, g = 511, s = 10, seed = 2, l = 4, m = 1024, z = 3, a = 0.55, b = 0.1, c = 0.1, w = 3, d = 0.25, v = 3	4	9.74 s

Abbildung 12: Konfiguration und Berechnungszeiten der Benchmarks

6 Exemplarische Ergebnisse

Die Benchmarks *UTS* [10] und *BC* [4] werden vom Fachgebiet verwendet, um die Performance von J_GLB und FT_GLB zu testen. Im Folgenden werden sie benutzt, um die Funktionen von GLB_Logger praktisch zu zeigen und im Zuge dessen, die Frameworks besser zu verstehen. Die Konfiguration und Berechnungszeiten der Versuche können in Abbildung 12 nachvollzogen werden.

6.1 Benchmark Charakteristika

Beide Benchmarks haben ein charakteristisches Laufzeitverhalten, welches über die generierten Histogramme sichtbar gemacht werden kann. UTS und BC wurden jeweils mit FT_GLB auf der Partition *public2* des Clusters des ITS der Universität Kassel ausgeführt.

6.1.1 UTS

Die UTS Benchmark zählt die Anzahl der Knoten in einem unbalancierten Baum, der dynamisch generiert wird. Die Berechnung beginnt an der Wurzel des Baums mit einem Task. Beim Durchlaufen des Baums werden an Abzweigungen neue Tasks erstellt. Da zu Beginn der Berechnung noch nicht alle Tasks feststehen, handelt es sich

um einen dynamischen Arbeitsaufwand.

In Abbildung 13 sieht man das generierte Histogramm.

Wir sehen, dass am Anfang ein großer Teil der Worker im Zustand Idling ist und nur langsam aktiv wird. Auf Grund des dynamischen Arbeitsaufwands wird nämlich zunächst nur der Worker auf Place 0 gestartet. Während er den ersten Task berechnet, entstehen neue Tasks. Nachdem der Worker eine festgelegte Anzahl Tasks berechnet hat, schickt er Tasks zu seinen Lifeline-Buddies. Die Lifeline-Buddies werden dabei gestartet und setzen das Verhalten des Workers auf Place 0 fort.

Wurden auf diese Weise alle Worker gestartet, beginnt die nächste Phase, in der das Framework eine stabile Lastenverteilung herstellt. Um diese zu erreichen, müssen Stehl-Versuche getätigt werden, weshalb der Anteil von Waiting und Communication zunächst steigt. Mit zunehmender Stabilität der Lastenverteilung haben Worker seltener einen leeren Taskpool, weswegen die Anzahl der Stehl-Versuche und Steal-Backups zurückgeht. Damit nimmt auch der Anteil von Waiting und Communication wieder ab.

Nach etwa einem Viertel der Laufzeit ist eine stabile Lastenverteilung erreicht. Die Worker wechseln vereinzelt in den Zustand Communication und Waiting für einen gelegentlichen Stehl-Versuch. Der Zustandstyp Communication tritt außerdem regelmäßig wegen Regular-Backups auf. Zum Großteil der Zeit befinden sich die Worker jedoch im Zustand Processing. Die Worker können ungestört arbeiten, weshalb hier die effektivste Phase der Berechnung ist.

Etwa im letzten Fünftel haben einzelne Worker keine Tasks mehr und die Effektivität sinkt allmählich. Die Worker unternehmen wieder vermehrt Stehl-Versuche, um die Lastenbalancierung stabil zu halten. Dabei müssen sie mehr miteinander kommunizieren und Backups schreiben. Damit drängen die Zustandstypen Waiting und Communication Processing zurück. Am Ende ist der Anteil von Workern im Zustand Processing minimal, weil nur noch wenige Worker Tasks bearbeiten.

6.1.2 BC

Die BC Benchmark berechnet für jeden Knoten eines Graphens einen Wert. Da die Knoten des Graphens bekannt sind, können alle Tasks am Anfang initialisiert werden. Die Tasks sind damit statisch.

In Abbildung 15 sieht man das erzeugte Histogramm.

Im Gegensatz zu UTS starten alle Worker gleichzeitig mit dem Berechnen von Tasks und sind damit im Zustand Processing. Da die Tasks statisch sind, ist die Lastenverteilung der Worker bereits zu Beginn stabil. Es müssen deshalb weitaus weniger Stehl-Versuche und damit Steal-Backups durchgeführt werden. Das Framework ist von Anfang an in seiner effektivsten Phase. In der Mitte der Abbildung sieht man einen Communication-Ausschlag, welcher auf Regular-Backups zurückzuführen ist. Da die Worker gleichzeitig starten, führen sie auch gleichzeitig dieses Backup durch.

Etwa im letzten Viertel erhöhen sich die Anteile von Communication und Waiting, während Processing langsam abnimmt. Die Worker haben häufiger keine Tasks mehr und tätigen Stehl-Versuche, die Lastenbalancierung wird zunehmend instabiler. Dieser Prozess setzt sich fort, bis am Schluss, ähnlich wie bei UTS, der Anteil von Processing beinahe verschwindet.

6.2 Vergleich zwischen J_GLB und FT_GLB

Beide Benchmarks wurden außerdem jeweils mit J_GLB getestet, um das Laufzeitverhalten beider Frameworks zu vergleichen. Insbesondere wollen wir herausfinden, welche Auswirkungen die Backups von FT_GLB auf die Berechnung der Benchmarks haben. Die Histogramme dieser Tests können in Abbildung 14 und 16 betrachtet werden. Die Versuche wurden ebenfalls auf dem Cluster des ITS der Universität Kassel ausgeführt.

Im Fall von UTS verlassen die Worker in J_GLB zu Beginn deutlich schneller den Zustand Idling. Wie in Abschnitt 6.1.1 beschrieben, werden bei UTS die Worker beginnend von Place 0 durch das Verteilen der generierten Tasks gestartet. Bei FT_GLB wird nach jedem Verteil-Vorgang ein Steal-Backup geschrieben, wodurch diese Phase verzögert wird. Zusätzlich dazu wird sie bei FT_GLB aus technischen Gründen ver-

langsamt, die hier nicht weiter ausgeführt werden. Diese technischen Gründe führen dazu, dass das Phänomen bei BC in abgeschwächter Form ebenfalls auftritt.

Im weiteren Verlauf wird bei UTS in J_GLB deutlich früher eine stabile Lastenbalancierung erreicht. In FT_GLB wird nach jedem erfolgreichen Stehl-Versuch ein Steal-Backup geschrieben. Die Stehl-Versuche, die für die Lastenbalancierung verantwortlich sind, verlängern sich dadurch. Da bei BC für einen großen Teil der Berechnung selten Stehl-Versuche tätigt werden, sind die Verläufe bei beiden Frameworks anfangs nahezu gleich.

Ist eine stabile Lastenbalancierung erreicht, entfallen bei J_GLB bei beiden Benchmarks die Communication-Anteile, die bei FT_GLB wegen den Regular-Backups auftraten.

Bei beiden Benchmarks zeigt sich, dass im Fall von FT_GLB die ineffiziente Schluss-Phase deutlich länger anhält. Dies liegt erneut an den Steal-Backups, da sie die Dauer der Stehl-Versuche verlängern, die am Schluss gehäuft vorkommen. Auch die, bereits oben erwähnten, technischen Gründe tragen zur Verlangsamung der Stehl-Versuche bei.

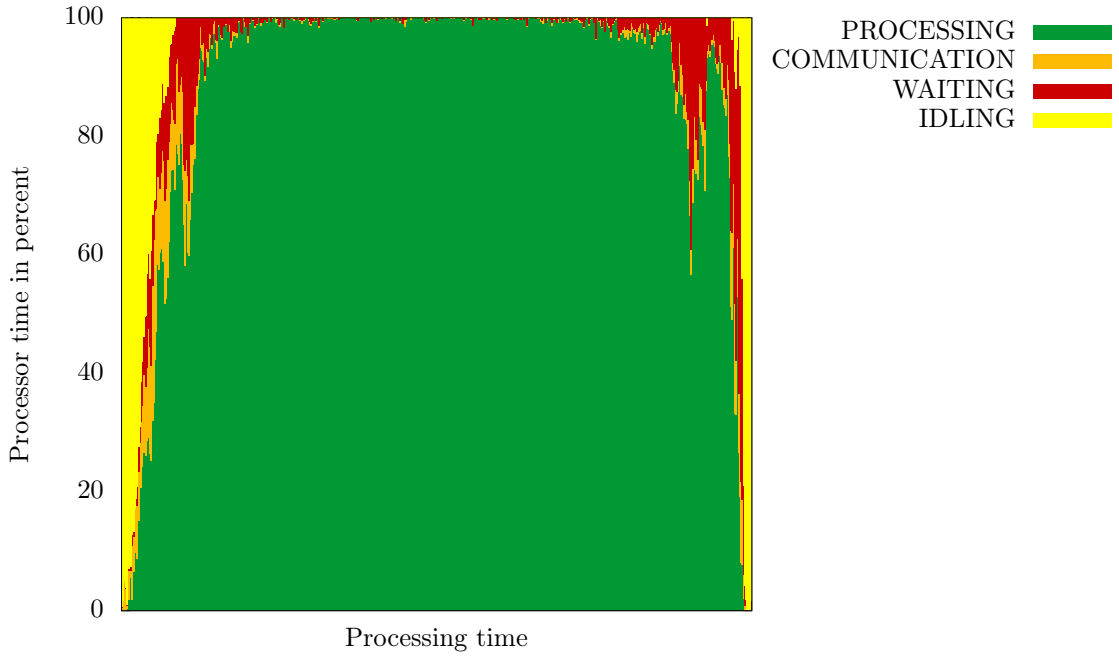


Abbildung 13: UTS Benchmark (FT_GLB)

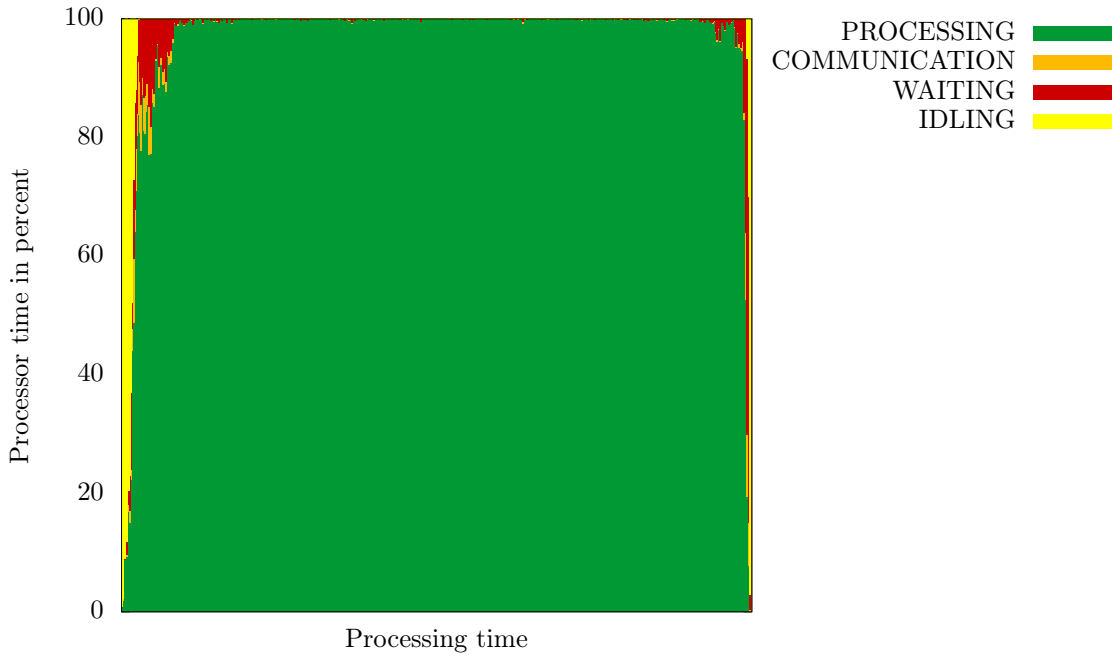


Abbildung 14: UTS Benchmark (J_GLB)

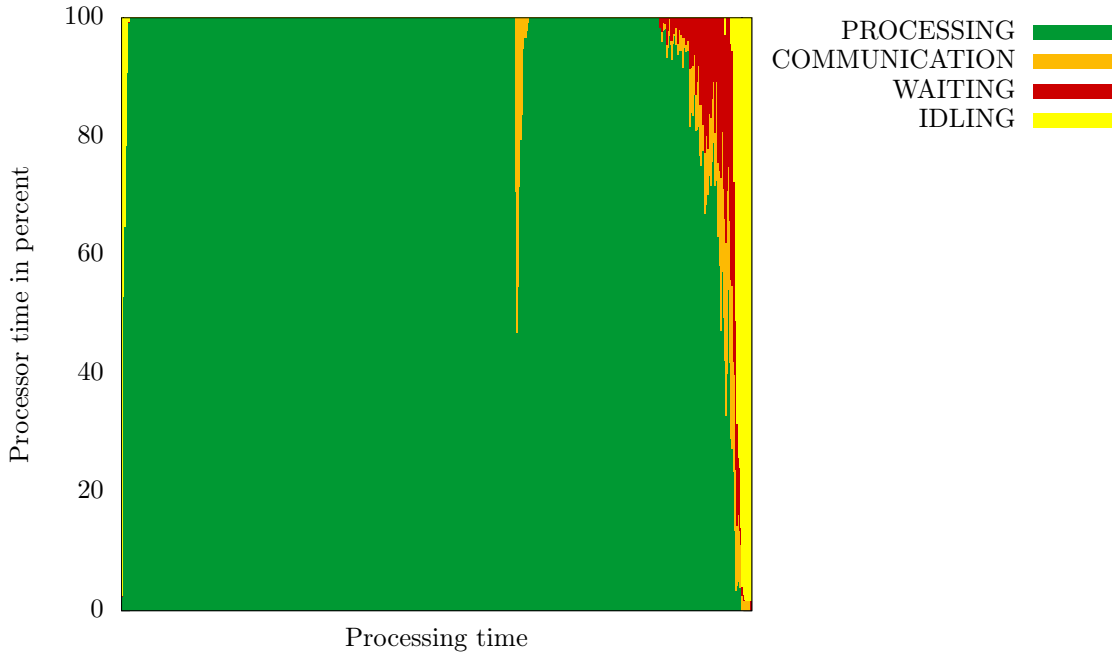


Abbildung 15: BC Benchmark (FT_GLB)

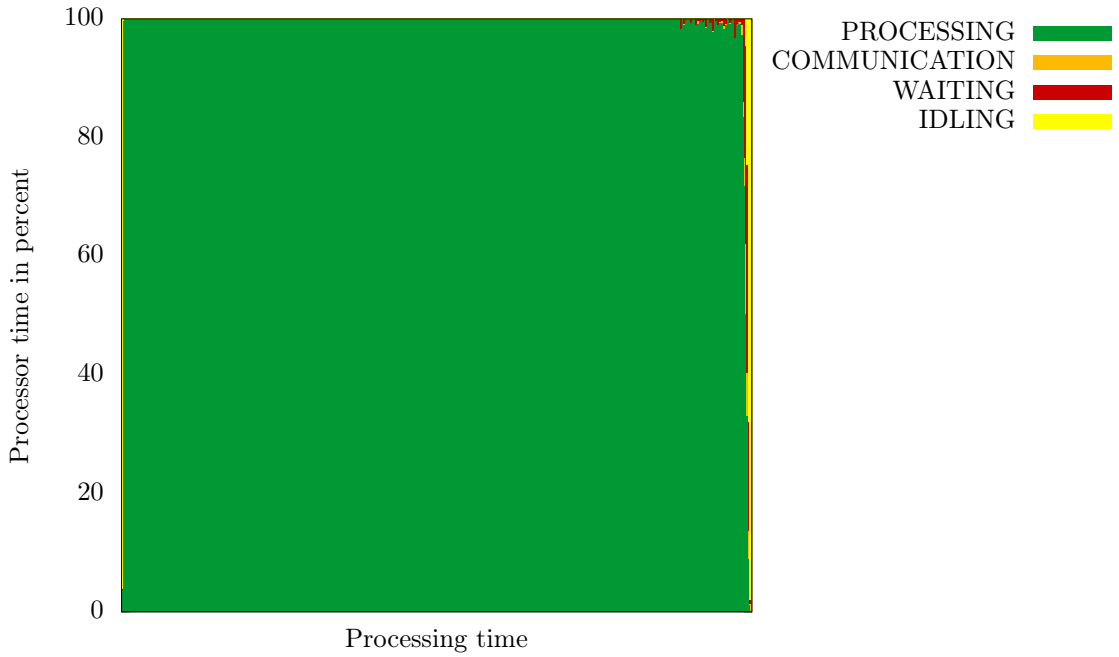


Abbildung 16: BC Benchmark (J_GLB)

6.3 Performance Optimierung

Über die Kommandozeilen-Argumente kann das Verhalten der Frameworks verändert werden. Für einen Anwendungsfall sollen die Parameter in der Regel so eingestellt werden, dass die Berechnungszeit möglichst gering ist. Dafür werden die Berechnungszeiten von Versuchen mit verschiedenen Parametereinstellungen verglichen und auf diese Weise eine optimale Einstellung herausgefunden. Die visuellen Darstellungen können bei diesem Prozess helfen, zu verstehen, wie sich die Veränderung eines Parameters auf das Laufzeitverhalten der Frameworks auswirkt. Dadurch erfährt man auch die Begründung, wieso eine Parametereinstellung im Anwendungsfall optimal ist.

Im Fall von UTS mit J_GLB gibt es den Parameter n. Dieser gibt die Anzahl der Tasks an, die der Worker verarbeitet, bevor er in der Anfangsphase einen Lifeline-Buddy startet oder registrierten Thieves antwortet. In Abbildung 17 wurde dieser Wert auf 5, in Abbildung 18 dagegen auf 1000 gesetzt. Alle anderen Einstellungsparameter wurden nicht verändert. Die Benchmark wurden wieder auf dem Cluster des ITS der Universität Kassel berechnet. Wie in Abbildung 12 zu sehen, war der Versuch mit dem Wert 1000 für n signifikant schneller als der andere Durchlauf. Mit Hilfe der visuellen Darstellung des GLB_Loggers können wir nun herausfinden, warum dies der Fall ist.

Vergleicht man die generierten Histogramme, fällt zunächst auf, dass in Abbildung 18 die Worker länger brauchen, um aktiv zu werden. Wie in Abschnitt 6.1.1 beschrieben, wird zunächst nur der Worker auf Place 0 gestartet. Durch das Verteilen der generierten Tasks werden dann auch die anderen aktiv. Da die Worker, durch ein höheres n, länger Tasks berechnen, bevor sie die Tasks verteilen, verzögert sich diese Phase bei einem hohen n.

Wenn n den Wert 5 hat, sind alle Worker sehr schnell aktiv. Dafür treten im ersten Viertel immer wieder große Ausschläge der Waiting-Anteile auf, bevor die Lastenbalancierung einen stabilen Zustand erreicht. Dies liegt daran, dass ein Worker zwar schnell Tasks zu seinen Lifeline-Buddies schickt; da er jedoch weniger Zeit mit der Generierung von Tasks verbringt, enthält seine Nachricht eine geringere Anzahl Tasks. Dadurch geschieht es schneller, dass ein gestarteter Worker keine Tasks mehr hat und beginnt, Stehl-Versuche zu tätigen.

Bei einem großen n hat der Worker mehr Zeit, Tasks zu berechnen und verschickt in einer Nachricht mehr Tasks. Dadurch müssen die Worker zwar länger auf die Antwort einer Stehl-Anfrage warten. Eine einzelne Stehl-Anfrage leistet dafür einen länger anhaltenden Beitrag zur stabilen Lastenbalancierung. Deshalb werden weniger Stehl-Anfragen benötigt. Wie in Abbildung 18 zu sehen, wird die Lastenbalancierung außerdem weitgehend gleichmäßig stabiler. In diesem Beispiel ist der Wert von 1000 für n deshalb besser.

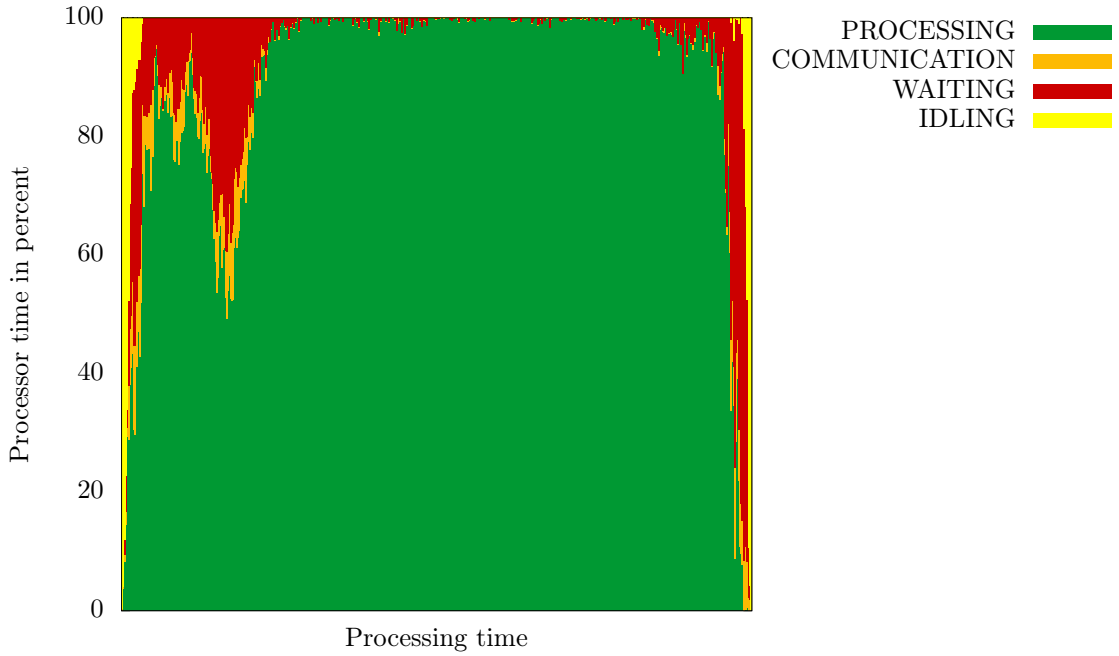


Abbildung 17: UTS Benchmark (J_GLB) mit $n = 5$

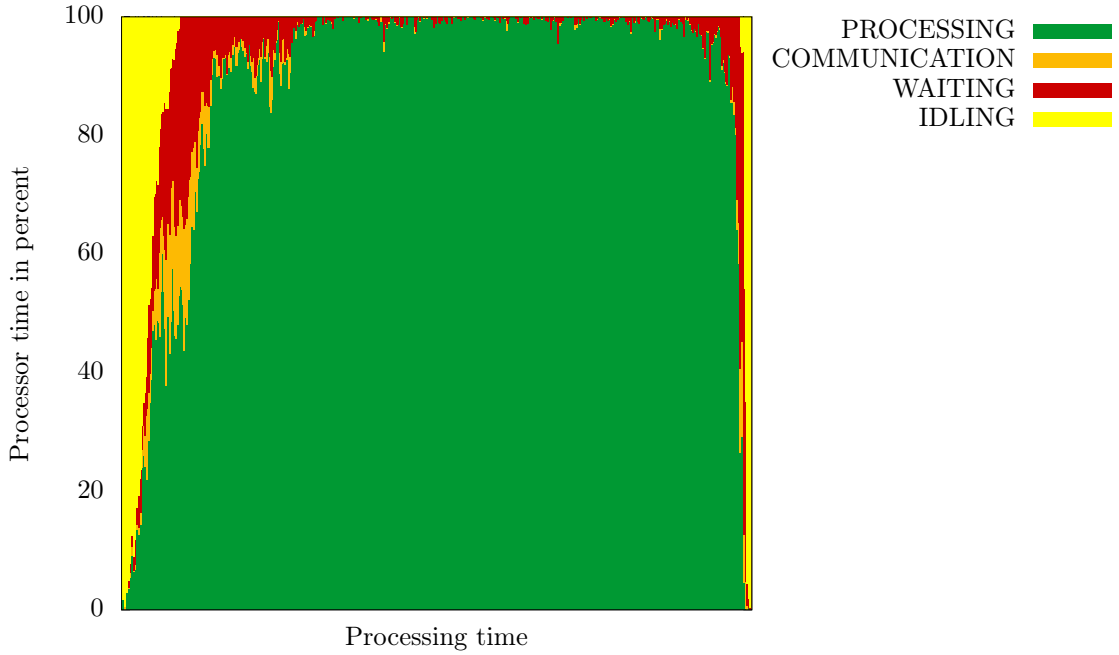


Abbildung 18: UTS Benchmark (J_GLB) mit $n = 1000$

6.4 Potenzielle Performance-Probleme

GLB_Logger kann verwendet werden, um potenzielle Performance-Probleme ausfindig zu machen. Um dies zu zeigen, werden im Folgenden zwei Probleme vorgestellt, die bei der Berechnung der Benchmarks mit den Frameworks entdeckt wurden.

6.4.1 Backups allokieren zu viel Speicher

Das erste Problem tauchte bei der Berechnung von UTS mit FT_GLB auf.

Die Spalte der Größe und der Anzahl der Backups aus `backup.csv` wurden jeweils in Abbildung 19 und 20 graphisch dargestellt. In Abbildung 20 fällt auf, dass die Größe der Backups über den Zeitraum der Berechnung konstant bleibt. Dies geschieht, obwohl sich wie in Abbildung 19 zu sehen, die Anzahl der Tasks im Backup verändert.

In den Backups wird das lokale Taskpool gespeichert. Dieses wird in den Frameworks durch die Interfaces `FTTaskQueue` für FT_GLB und `TaskQueue` für J_GLB modelliert. Für jedes Programm muss eine Klasse implementiert werden, die vom entsprechenden Interface erbt und alle Tasks aufbewahrt. Im Fall von UTS beinhaltet die Klasse dafür ein Array. Reicht die Größe des Arrays nicht mehr aus, um alle Tasks zu fassen, wächst es. Das Problem hatte seinen Ursprung darin, dass die Start-Größe des Arrays mit einem zu großen Wert initialisiert wurde. Das Array war über den kompletten Verlauf der Berechnung groß genug, um alle Tasks aufzubewahren. Dadurch musste es nicht vergrößert werden. Das Backup speicherte damit viele unbesetzte Array-Elemente und behielt eine konstante Größe bei.

Durch das Setzen der Anfangsgröße des Arrays auf einen niedrigeren Wert konnte das Problem behoben werden. Zur neuen Ausgabedatei wurden die Graphiken in Abbildung 21 und 22 erstellt. Vergleicht man sie mit den oben beschriebenen Graphiken, sehen wir zum einen, dass die Größe der Backups mit dem Ansteigen der Tasks steigt. Nachdem die maximale Anzahl im fünften Backup erreicht ist, bleibt die Backup-Größe konstant. Das bedeutet, dass das Wachsen des Arrays nun funktioniert. Außerdem ist die Größe der Backups insgesamt deutlich geringer als im letzten Versuch.

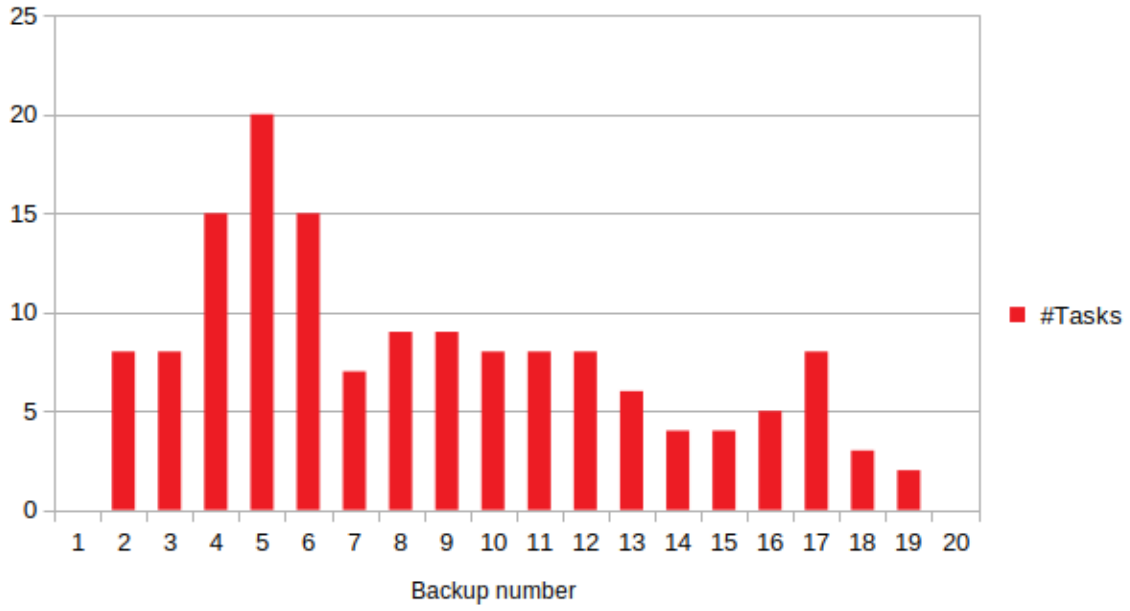


Abbildung 19: UTS (FT_GLB): Anzahl der Tasks in den Backups

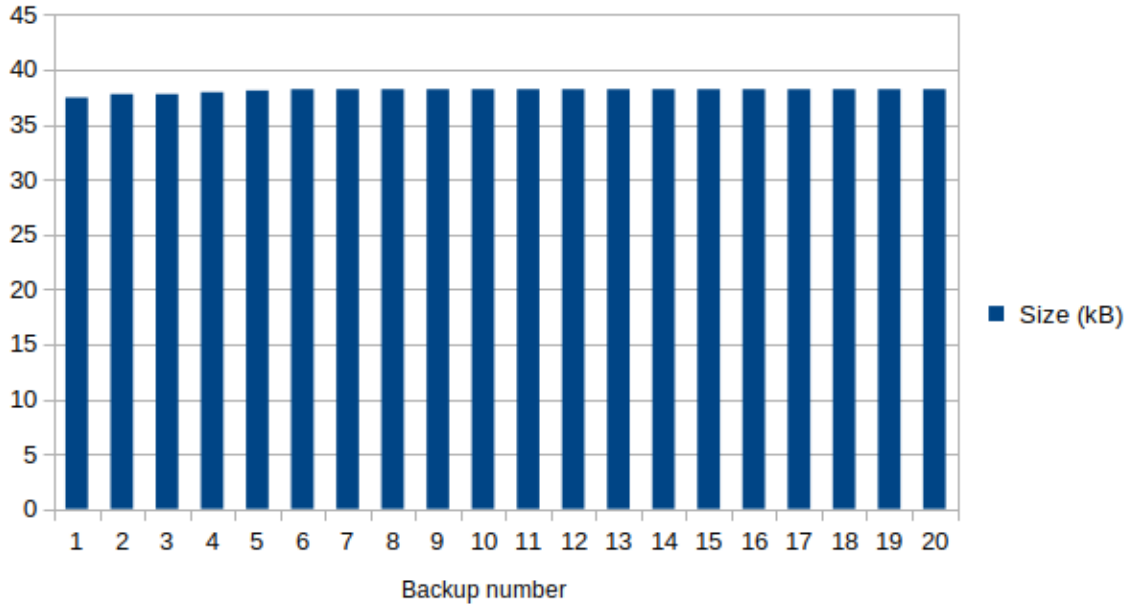


Abbildung 20: UTS (FT_GLB): Größe der Backups

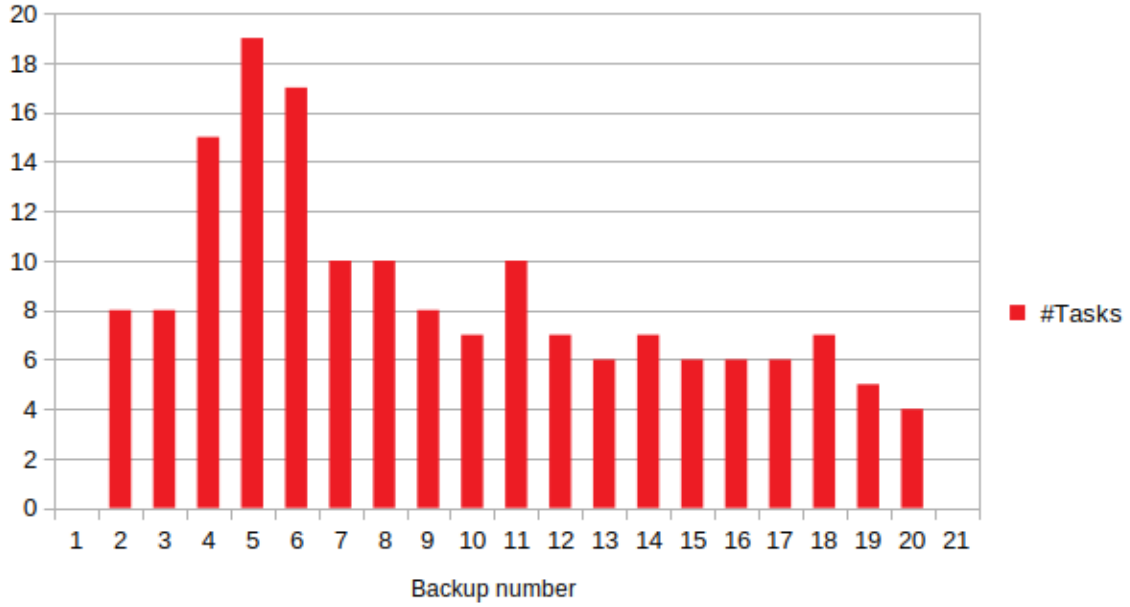


Abbildung 21: UTS mit einer Array-Größe von 1 (FT_GLB): Anzahl der Tasks in den Backups

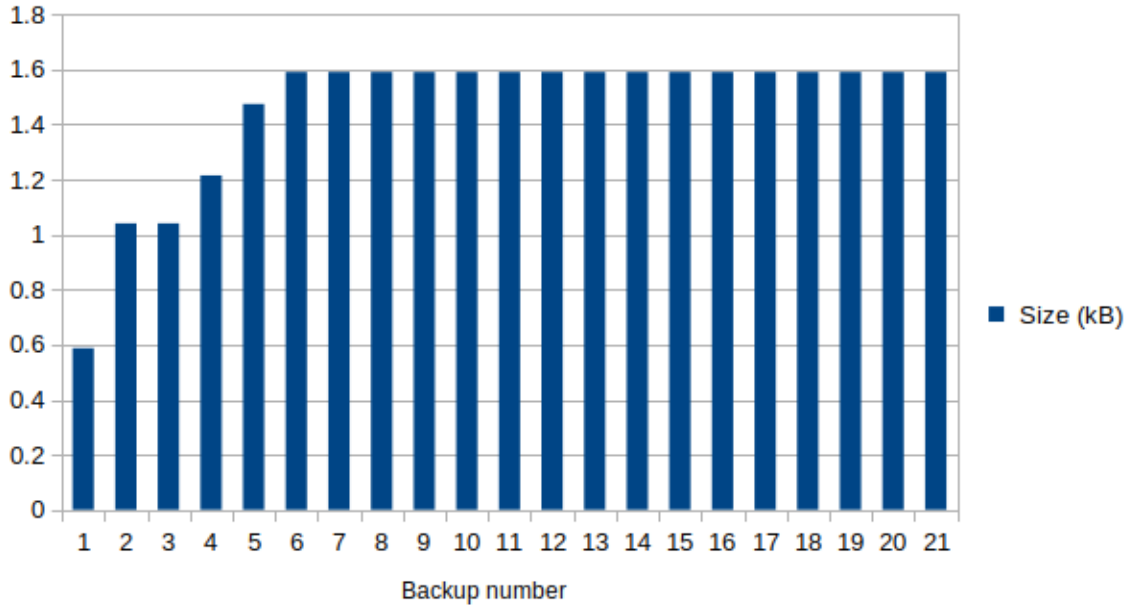


Abbildung 22: UTS mit einer Array-Größe von 1 (FT_GLB): Größe der Backups

Abbildung 23: BC (FT_GLB) Zeitleiste

6.4.2 BC Benchmark: Erstes Backup wird zu früh geschrieben

Das nächste Probleme zeigte sich bei der Berechnung von BC mit FT_GLB.

FT_GLB schreibt nach einer konstanten Dauer regelmäßig Regular-Backups. In diesem Fall war diese Dauer auf 10s gesetzt worden. In Abbildung 23 sieht man einen Ausschnitt der Zeitleiste, die in Folge der Berechnung generiert wurde. Dabei fällt auf, dass die ersten Communication-Zeitabschnitte bereits nach wenigen Millisekunden auftreten. Da BC einen statischen Arbeitsaufwand hat, sind Stehl-Versuche an einem so frühen Zeitpunkt unwahrscheinlich. Deshalb vermuten wir, dass es sich um Regular-Backups handelte, welche zu früh geschrieben wurden.

Um dieser Vermutung Gewissheit zu verschaffen, wird einer der Zeitabschnitte genauer betrachtet. Klickt der Benutzer auf einen Zeitabschnitt der Zeitleiste, erhält er verschiedene Informationen. Wie in Abbildung 23 zu sehen, hat ein verdächtiger Zeitabschnitt die ID 2. In Abbildung 24 sieht man einen Ausschnitt von `stack.csv` und `backup.csv`. In der Zeile von `stack.csv`, die zur ID 2 gehört, sieht man, dass in dem Zeitabschnitt das Backup mit der ID 1 geschrieben wurde. Dieses kann in `backup.csv` gefunden werden und zeigt, dass es sich tatsächlich um ein Regular-Backup handelte.

Als nächstes betrachten wir den Code von FT_GLB, um den Fehler zu finden. In Listing 7 sieht man einen Ausschnitt des Codes der Worker-Klasse. Dieser wird immer ausgeführt, nachdem Tasks berechnet wurden. Die Zeit seit dem letzten Regular-Backup wird wie in Zeile 1 berechnet. Die Variable `timestampLastBackup` gibt dabei den Zeitpunkt an, in dem das letzte Backup geschrieben wurde. Danach wird geprüft, ob die berechnete Zeit über den Wert der globalen Variable s hinausgeht. Diese gibt an, nach welcher Zeit ein Regular-Backup geschrieben wird. Ist dies der Fall, wird in

stack.csv

Place ID	Timerange ID	Comments	Stack	...
0	0		FTTimedGLB.FTTimedGLB.lambda ...	...
0	1		FTTimedGLB.FTTimedWorker.processStack ...	...
0	2	{backup-id=1}	FTTimedGLB.FTTimedWorker.writeBackup ...	...

backup.csv

Place ID	Backup ID	Start (ms)	Type	#Tasks	Size (kB)	Writing Time
0	0	-17.76	INIT_BACKUP	1	0	15.96
0	1	59.46	REG_BACKUP	1	0	103.48
0	2	1163.65	REG_BACKUP	1	0	42.93

Abbildung 24: BC (FT_GLB) Ausschnitt aus `stack.csv` und `backup.csv`

```
1 long period = System.nanoTime() - timestampLastBackup;
2 if ((period / 1E9) >= this.s) {
3     lastTimeRangeId = this.writeBackup(BackupType.REG_BACKUP,
          Integer.MIN_VALUE);
4 }
```

Listing 7: FT_GLB: Berechnung der Dauer seit dem letzten Backup

Zeile 3 ein Backup geschrieben. Die Variable `timestampLastBackup` wird aktualisiert, nachdem ein neues Backup getätigt wurde.

Der Fehler trat zu Beginn der Berechnung auf. Nachdem der Worker die ersten Tasks berechnete, wurde der Code aus Listing 7 ausgeführt. Die Variable `timestampLastBackup` war jedoch noch nicht initialisiert und hatte den Wert 0. Deshalb ergab die Berechnung der Zeit, die seit dem letzten Backup vergangen ist, den Wert aus `System.nanoTime()`. Dieser Wert ist sehr hoch, weshalb ein Regular-Backup geschrieben wurde.

Das Problem konnte einfach behoben werden, indem `timestampLastBackup` in der Methode zum Berechnen der Tasks einmalig mit dem aktuellen Zeitpunkt initialisiert wird.

7 Verwandte Arbeiten

Die Protokollierung und Visualisierung des Laufzeitverhaltens von parallelen Programmiersystemen ist ein großes Thema in der Forschung. Für die parallele Programmiersprache X10 wurde für diesen Zweck X-Eye [8] entwickelt. Nicht immer wird für ein System ein eigenes Visualisierungswerkzeug implementiert. Stattdessen greifen die Programmierer auf generische Frameworks zurück, welche diese Aufgabe für parallele Programmiersysteme übernehmen. Beispielsweise wird für die Analyse des parallelen Frameworks COMPS [1] Paraver [12] verwendet. Weitere Beispiele für solche Werkzeuge sind TAU [16], PPW [18], Vampir [3], und VGV [7].

Im parallelen Programmiersystem Legion lässt sich das Laufzeitverhalten wie bei GLB_Logger mittels einer Zeitleiste visualisieren [17]. Die entsprechende Komponente heißt *Legion Prof* und stellt die Zeitleiste ebenfalls auf einer interaktiven Webseite dar. Sieht man von den grundsätzlichen Unterschieden der verschiedenen Systeme ab, bietet Legion Prof einige interessante zusätzliche Funktionalitäten. Diese könnten in der zukünftigen Entwicklung von GLB_Logger implementiert werden.

Ein Vorteil von Legion Prof ist, dass bei einem Klick auf einen Zeitabschnitt mehr Informationen angezeigt werden. Bei GLB_Logger muss der Benutzer erweiterte Informationen in Textdateien nachschauen. Stattdessen könnten zukünftig Informationen aus `stack.csv` beim Klicken auf einen Zeitabschnitt erscheinen.

In Legion Prof können zudem im Graphen weitaus mehr Informationen visuell dargestellt werden. In einem erweiterten Modus werden Linien zwischen abhängigen Zeitabschnitten gezogen. Die Abhängigkeit der Zeitabschnitte begründet sich zunächst auf Eigenschaften des Legion-Systems, die in den Modellen der GLB-Frameworks nicht vorkommen; es können jedoch einfach eigene Abhängigkeiten in den GLB-Frameworks gefunden werden. So könnten miteinander kommunizierende Worker mit Linien verbunden werden oder Worker, die auf die Antwort eines Workers warten. Die dafür benötigten Informationen befinden sich alle in `stack.csv`.

8 Fazit

In dieser Bachelorarbeit wurde die Logging-Komponente GLB_Logger vorgestellt. Diese ermöglicht es, eine Vielzahl von Informationen über die Frameworks J_GLB und FT_GLB aufzuzeichnen und zu visualisieren.

GLB_Logger lässt sich im Umgang mit den beiden Frameworks in verschiedenen Bereichen einsetzen. Zum einen kann es beim Einstieg in die Frameworks helfen. Über die Graphiken können die komplexen Vorgehensweisen auf eine neue Art erschlossen werden und sprechen damit insbesondere visuelle Lerntypen an. Entwickler können die neuen Funktionen des Loggers zudem nutzen, um Performance-Probleme und Fehler im Code zu finden. Während Benutzer der Frameworks die Parameter für ihren Anwendungsfall optimal einstellen, können sie die Graphiken als Unterstützung verwenden.

Damit kann GLB_Logger sowohl Entwickler, als auch Benutzer der Frameworks bei ihrer Arbeit unterstützen.

Literatur

[1] Rosa M. Badia u. a. „COMP Superscalar, an interoperable programming framework“. In: *SoftwareX* 3 (Nov. 2015).

[2] Baeldung. *How to Get the Size of an Object in Java.* URL: `https://www.baeldung.com/java-size-of-object` (besucht am 26. 02. 2020).

[3] H. Brunst und W. E. Nagel. „Scalable performance analysis of parallel systems: Concepts and experiences“. In: Proc. of the Parallel Computing Conf., Elsevier, 2003.

[4] L. C. Freeman. „A Set of Measures of Centrality Based on Betweenness“. In: Sociometry 40.1, 1977, S. 35–41.

[5] Gnuplot Community. *Calculating histograms.* Gnuplot Version 5.2 Info Webseite, 2019. URL: `http://www.gnuplotting.org/calculating-histograms/` (besucht am 04. 02. 2020).

[6] Google LLC. *Timelines.* Google Charts Webseite. URL: `https://developers.google.com/chart/interactive/docs/gallery/timeline` (besucht am 22. 01. 2020).

[7] J. Hoeflinger u. a. „An Integrated Performance Visualizer for MPI/OpenMP Programs“. In: *Workshop on OpenMP Appl. and Tools WOMPAT 2001* (Juli 2001).

[8] Seisei Itahashi und Yoshiki Sato. „Toward a profiling tool for visualizing implicit behavior in X10“. In: 2014.

[9] jodonnell (Stack Overflow User). *In Java, what is the best way to determine the size of an object?* URL: `https://stackoverflow.com/questions/52353/in-java-what-is-the-best-way-to-determine-the-size-of-an-object#52362` (besucht am 05. 02. 2020).

[10] S. Olivier, J. Liu J. Huan u. a. „TS: An Unbalanced Tree Search Benchmark.“ In: Languages and Compilers for Parallel Computing. Springer LNCS 4382, 2006, S. 235–250.

[11] Oracle. *The Java^{TM} Tutorials - Serializable Objects.* URL: `https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html` (besucht am 26. 02. 2020).

[12] V. Pillet u. a. „Paraver: A Tool to Visualize and Analyze Parallel Code“. In: 18th World OCCAM und Transputer User Group Technical Meeting, 1995. URL: `http://www.bsc.es/paraver`.

[13] Jonas Posner und Claudia Fohry. „A Java Task Pool Framework providing Fault-Tolerant Global Load Balancing“. In: *International Journal of Networking and Computing* 8 (2018), S. 2–31.

[14] Jonas Posner und Claudia Fohry. *Cooperation vs. coordination for lifeline-based global load balancing in APGAS. In Proc. ACM SIGPLAN X10 Workshop*. 2016, S. 13–17.

[15] Vijay Saraswat u. a. *Lifeline-based Global Load Balancing. In Proc. ACM Symp. on Principles and Practice of Parallel Programming.* 2011, S. 201–212.

[16] S. Shende und A. D. Malony. „The TAU Parallel Performance System“. In: Bd. 20. Int. Journal of High Performance Computing Appl., 2006, 2:287–331.

[17] Stanford University. *Legion Programming System - Performance Profiling and Tuning.* URL: `https://legion.stanford.edu/profiling` (besucht am 02. 03. 2020).

[18] H. Su u. a. „Parallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming“. In: *PDSEC-IPDPS 2008* (Apr. 2008).

[19] Olivier Tardieu. *The APGAS library: resilient parallel and distributed programming in Java 8. In Proc. ACM SIGPLAN X10 Workshop*. Juni 2015, S. 25–26.

[20] C. Ullenboom. „Java ist auch eine Insel: Einführung, Ausbildung, Praxis“. In: Rheinwerk Computing. Rheinwerk, 2017. Kap. 7.8 Den Stack-Trace erfragen. URL: `http://openbook.rheinwerk-verlag.de/javainsel/07_008.html`.

[21] Zhang Wei u. a. *GLB: Lifeline-Based Global Load Balancing Library in X10. In Proc. ACM Workshop on Parallel Programming for Analytics Applications.* 2014, S. 31–40. ISBN: 9781450326544.