UNI KASSEL
VERSITÄT

Universität Kassel

Bachelorarbeit

Anwendung einer SplitQueue Datenstruktur auf Work Stealing in Laufzeitsystemen taskbasierter paralleler Programmiersysthemen

vorgelegt vor
Fachbereich Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik

Tobias Werner
35613833

Kassel, 12. September 2022

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

Inhaltsverzeichnis

Verzeichnis der Abbildungen Tabellen und Codeabschnitte II

Selbstständigkeitserklärung III

1 Einleitung 1

2 Hintergrund 4
2.1 APGAS 4
2.2 Kooperatives Global Load Balancing 6

3 SplitQueue 11

4 Koordiniertes GLB mit SplitQueue 15
4.1 Implementation und Integration der SplitQueue 15
4.2 Änderungen an GLB 17

5 Ergebnisse 19
5.1 Benchmarks 19
5.2 Konfiguration und Ausführung der Benchmarks 20
5.3 Ergebnisse der Benchmarks 21
5.4 Diskussion der Ergebnisse 23

6 Verwandte Arbeiten 25

7 Fazit 26

Literatur VII

Verzeichnis der Abbildungen Tabellen und Codeabschnitte

2.1 Ablauf eines kooperativen GLB Workers 10

3.1 SplitQueue Datenstruktur, grauer Bereich ist freier Speicher . . . 11
3.2 SplitQueue Metadaten 12

4.1 Ablauf eines koordinierten GLB Workers 17

5.1 Parameter der Benchmarks 20
5.2 UTS Benchmark. 21
5.3 BC Benchmark. 22
5.4 Synthetic Benchmark, Overhead des Laufzeitsystems. 23

2.1 APGAS Beispielprogramm. 5
2.2 APGAS Beispiel Ausgabe 5

3.1 steal Operation 13

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. September 2022

Tobias Werner

1 Einleitung

Mit dem Fortschritt der Digitalisierung steigt die Anzahl und Komplexität von Problemen in der Informatik. Komplizierte Probleme werden im High Performance Computing mit einem Supercomputer gelöst. Heutige Supercomputer nutzen die Clusterarchitektur. Ein Cluster verbindet leistungsstarke Rechner über einem Netzwerk zu einem Computer mit hoher Rechenleistung den Supercomputer. Auf diesem Supercomputer werden Probleme mit konstanten und dynamische Arbeitsaufwand gelöst. Konstanter Arbeitsaufwand ist vorhanden, wenn vor der Berechnung die gesamte Anzahl von Arbeit fest ist und während der Ausführung die Anzahl sich nicht verändert. Supercomputer eignen sich gut diese Probleme zu lösen. Probleme mit dynamischen Arbeitsaufwand zeichnen sich durch eine variable Anzahl an Arbeit aus mit einer Veränderung der Anzahl zur Laufzeit. Besonders komplizierte Probleme besitzen einen dynamischen Arbeitsaufwand und effiziente Lösungen dafür sind nötig.

Manche Rechner sind bei dynamischen Probleme schneller fertig als andere und daher schlecht ausgelastet. Dynamische Lastenbalancierung verteilt die Arbeit während der Laufzeit und versucht eine möglichst hohe Auslastung der vorhandenen Rechner zu erreichen. Dafür wird die Arbeit in kleinteilige Arbeitspakete Tasks zerlegt und zwischen den Rechnern getauscht. Tasks können weitere Tasks erzeugen und werden in einer verteilten Datenstruktur auf mehreren Rechnern gespeichert sowie verarbeitet.

Der Tausch von Arbeit findet über Work Stealing statt. Ist ein Worker ohne Arbeit, wählt er zufällig einen anderen Worker aus und versucht Arbeit von diesem zu stehlen. Der Stehlende heißt Dieb und der Bestohlene heißt Opfer. Für das Stehlen gibt es die Ansätze des kooperativen und koordinierten Work Stealing.

Beim kooperativen Work Stealing arbeiten Dieb und Opfer zusammen. Der Dieb stellt eine Anfrage an das Opfer, das Opfer unterbricht seine Arbeit periodisch und behandelt die Anfrage. Das Opfer bei dem koordinierten Work Stealing legt ein Teil seiner Arbeit für den Dieb zurück und der Dieb stiehlt diesen Teil. Ein möglicher Vorteil vom koordinierten Work Stealing wäre das Stehlen während der Abarbeitung von Tasks.

Diese Arbeit implementiert das koordinierte Work Stealing mit der verteilten Datenstruktur SplitQueue. Diese SplitQueue wurde für dieses Work Stealing entwickelt und vor kurzem von Cartier, Dinan und Larkins verbessert [1]. Die SplitQueue ist eine Warteschlange unterteilt in den private und public Bereich. Diebe stehlen aus dem public Bereich und Worker verarbeiten Tasks aus dem private Bereich. Die SplitQueue ist wie von Cartier beschrieben implementiert worden. Der SplitQueue wurden neue Operationen für die Anwendung in der Lastenbalancierung hinzugefügt. Die Lastenbalancierung basiert auf der Implementierung von Hardenbicker [3] und wurde für das koordinierte Work Stealing angepasst. Bei der Anpassung für koordiniertes Work Stealing konnte Synchronisation entfernt werden.

Für beide Varianten sind drei Benchmarks implementiert worden. Die Benchmarks UTS, BC und ein synthetischer Benchmark wurden auf dem Cluster der Universität Kassel ausgemessen. Bei einem synthetischen Benchmark kann der Arbeitsaufwand genau eingestellt werden und der Overhead, der Verwaltungsaufwand des Laufzeitsystems, der Lastenbalancierung dadurch besser gemessen werden. Als Ergebnis ist das koordinierte Work Stealing leicht besser in den UTS und BC Benchmarks. Bei dem synthetischen Benchmark ist koordiniertes Work Stealing deutlich besser.

Diese Arbeit ist in 7 Abschnitte aufgeteilt. Anfangs beschreibt der 2. Abschnitt das Programmiermodell APGAS und die kooperative GLB Variante. Darauf wird die SplitQueue nach Cartier im 3. Abschnitt eingeführt. Abschnitt 4 beschreibt die Implementation der koordinierten GLB Variante mit der SplitQueue und

anschließend wird im Abschnitt 5 beide Varianten über Benchmarks auf dem Cluster der Universität Kassel ausgemessen und die Ergebnisse präsentiert. Die Arbeit wird mit einem Fazit in Abschnitt 7 abgeschlossen und vorher wird kurz auf verwandte Arbeiten eingegangen.

2 Hintergrund

In diesem Kapitel wird das Programmiermodell APGAS vorgestellt und aufbauend GLB, global load balancing with lifelines, eingeführt. Die GLB Variante nutzt kooperatives Work Stealing und die Variante mit koordinierten Work Stealing wird in Abschnitt 4 beschrieben.

2.1 APGAS

Das Programmiermodell APGAS[7] erweitert das Programmiermodell PGAS, was für für Partitioned Global Address Space steht. In PGAS wird der Speicher auf den Rechnern des Clusters in einen globalen Speicher zusammen gefasst. Jeder Rechner hat seinen lokalen Speicherteil und kann den Speicher von anderen Rechnern global adressieren. PGAS arbeitet mit einer konstanten Anzahl an Threads auf jedem Rechner und jeder Thread führt das gleiche Programm aus. PGAS findet breite Anwendung in parallelen Programmiersprachen wie Fortran oder UPC. APGAS erweitert die Idee von PGAS mit asynchronen Konstrukten für Probleme mit dynamischen Arbeitsaufwand.

In APGAS wird ein Rechner und sein Speicher zu einem place zusammen gefasst Auf jedem place läuft eine feste Anzahl an Threads. Jeder place hat Zugriff auf alle anderen places und verarbeitet Tasks, welche den Threads zur Laufzeit zugeteilt werden. Die Berechnung startet auf place 0 mit einem Task und alle anderen places sind im Leerlauf. Mit den Konstrukten können Tasks auf anderen places erzeugt und synchronisiert werden. Jeder Task ist eine Mischung von Programmcode und Konstrukten. Das Konstrukt **at** sendet einen Task zu einem anderen place und blockiert die Ausführung bis der Task auf dem anderen place ausgeführt wurde. Mit **finish** und **asyncAt** können asynchrone, nicht blockierende, Tasks

```
1  class HelloAPGAS{
2      public static void main(String[] args){
3          System.out.println("Hello World!\n");
4
5          finish(() -> {
6              for(Place place : places()){
7                  asyncAt(() -> System.out.println("place " + here()));
8              }
9          });
10
11         System.out.println("\nBye World!");
12     }
13 }
```

Listing 2.1: APGAS Beispielprogramm.

erstellt werden. **finish** blockiert die Ausführung bis alle im finish ausgeführten von **asyncAt** erstellten Tasks fertig sind. **asyncAt** sendet einen Task zu einem anderen place und blockiert **nicht** die Ausführung von weiteren Code. Zuletzt gibt es noch das **uncountedAsyncAt**, welches wie **asyncAt** funktioniert und nicht von **finish** erfasst wird. Jedes Konstrukt kann auch Tasks lokal auf dem eigenen place erzeugen.

```
Hello World!

place 1
place 3
place 0
place 2

Bye World!
```

Listing 2.2: APGAS Beispiel Ausgabe

Die Codebeispiele 2.1 und 2.2 demonstrieren ein einfaches Beispiel von der APGAS Implementierung in Java. Auf place 0 wird die Ausführung mit einem **finish** synchronisiert. Jeder place gibt asynchron seine Nummer mit **asyncAt** aus und dadurch sind die Nummern nicht geordnet. places ist ein Array mit allen places und here() gibt die aktuelle place Nummer zurück.

2.2 Kooperatives Global Load Balancing

Global Load Balancing, GLB nutzt die Konstrukte von APAGAS, um Tasks mittels Work Stealing auf verschiedene Worker zu verteilen. Jeder place besitzt mehrere Worker und diese verarbeiten Tasks. Die Anzahl der Worker pro place ist konstant für jeden place und am Start der Berechnung festgelegt. Wenn bei einem Worker keine Tasks vorhanden sind, dann versucht der Worker zufällig von anderen Worker zu stehlen. Das Stehlen von Tasks ist kooperativ und koordiniert zwischen den Workern möglich. Das koordinierte Work Stealing wurde in dieser Arbeit implementiert. Diese Implementierung wird in Abschnitt 4 erläutert und modifiziert die Implementierung vom kooperativen Work Stealing von Hardenbicker[3], welche in diesem Abschnitt erläutert wird. Beide Implementierungen basieren auf dem Konzept von GLB[6], welches für Multicorerechner erweitert wurde[2].

Work Stealing nennt den bestohlenen Worker Opfer und der stehlende Worker Dieb. Im kooperativen Ansatz senden Diebe Anfragen für Tasks an das Opfer und warten auf eine Antwort. Das Opfer unterbricht die Verarbeitung von Tasks nach einer vorgegebenen Anzahl n und behandelt diese Anfragen. Erhält ein Dieb als Antwort Tasks vom Opfer, dann hört der Dieb mit dem Stehlen auf und fängt an die Tasks zu verarbeiten. Wird die Anfrage abgelehnt bestiehlt der Dieb das nächste Opfer. Hat das Opfer keine Arbeit wird die Anfrage sofort abgelehnt.

Aus diesem Ansatz entsteht das Problem das Ende der Berechnung zu erkennen. Zum Ende der Berechnung würden nahe zu alle Worker zu Dieben werden und

das Netzwerk mit Anfragen belasten. Das Ende der Berechnung zu erkennen wird dadurch schwierig. GLB erweitert das Modell vom Work Stealing mit dem Konzept von lifelines und löst damit das Problem. Vor der Ausführung erhält jeder Worker lifeline buddies und diese buddies verbinden alle Worker zu einem zusammenhängenden Graphen. Mit lifelines werden erfolglose Diebe nach k Stehlanfragen deaktiviert und vor dieser Deaktivierung senden Diebe ihren lifeline buddies eine Rettungsanfrage. Bei der Behandlung der Stehlanfragen werden auch Rettungsanfragen behandelt. Eine beantwortete Rettungsanfrage aktiviert den Worker und dieserverarbeitet die neuen Tasks. Nun deaktivieren sich erfolglose Diebe zum Ende der Berechnung und belasten das Netzwerk nicht mehr. Sind alle Worker deaktiviert, dann sind alle Tasks im System verarbeitet und das Ergebnis kann ermittelt werden. Der beschriebene Ablauf eines Workers der kooperativen GLB Variante ist in Abbildung 2.1 dargestellt. In diesem Ablauf gibt es zwei Schleifen, die Verarbeitungsschleife und die Diebesschleife. Ohne Tasks geht der Worker zu der Diesebesschleife über und lehnt alle Anfragen ab. Mit vorhandene Tasks werden Anfragen beantwortet und erneut Tasks verarbeitet. Die Diebesschleife arbeitet nach dem Konzept von lifelines.

Ein Worker verwaltet seine Tasks in einem Bag und Bags werden vom Anwender der Lastenbalancierung implementiert. In der Implementierung von Kai ist ein Bag ein Interface und der Anwender implementiert eine Klasse mit Datenstrukturen für Tasks sowie die Verarbeitung von Tasks. Das Interface fordert folgende Operationen mit B als Bag und R als Ergebnis.

- int **process**(int workAmount): der Worker verarbeitet maximal workAmount Tasks und gibt die Anzahl der verarbeiteten Tasks zurück
- void **submit**(R result): am Ende der Berechnung werden die Ergebnisse zu einem Ergebnis result zusammen gesfasst
- int **getCurrentTaskCount**(): gibt die Anzahl Tasks im Bag zurück

- B **split**(boolean takeAll): entfernt die Hälfte der Tasks aus dem Bag und gibt einen Bag mit diesen Tasks zurück. Falls die Flag takeAll gesetzt ist werden alles Tasks entfernt.
- void **merge**(B toMerge): fügt alle Tasks in toMerge dem Bag hinzu
- boolean **isSplittable**(): gibt an ob genügend Tasks für **split** vorhanden sind
- boolean **isEmpty**(): gibt an ob Tasks im Bag vorhanden sind

Bei der Bearbeitung von einer Anfrage wird zuerst mit isSplittable auf entbehrliche Tasks überprüft und darauf ein Bag mit split erstellt. Der erstellte Bag wird dem Worker aus der Anfrage gesendet und dieser speichert die Tasks mit merge. Das Laufzeitsystem greift auf Worker Bags nie gleichzeitig zu und der Anwender braucht keine Synchronisation im Bag zu implementieren. Bereits erwähnt ist die Synchronisation vom Work Stealing, wo ein Dieb immer nur ein Opfer bestiehlt und auf eine Antwort von diesem wartet. Über Work Stealing wird dadurch die Operationen auf den Bag immer einmalig ausgeführt. Hingegen kann ein Worker von mehreren lifeline buddies zu unterschiedlichen Zeiten aktiviert werden. Dadurch entsteht ein gleichzeitiger Zugriff wie ein aktivierter Worker verarbeitet Tasks mit process und neue Tasks werden mit merge dem Bag hinzugefügt. Dieser Zugriff wird über kritische Abschnitte mit synchronised-Blöcken gelöst, indem jeder Zugriff auf den Bag in einem synchronised-Block liegt. Alle Operationen auf Worker Bags schließen sich nun gegenseitig aus. Mit der Synchronisation des Work Stealing und lifeliens arbeitet die Implementierung der Berechnung korrekt.

Die Berechnung startet mit allen Worker deaktiviert mit vorhandenen Rettungsanfragen und einem Bag mit allen Tasks. Ein Worker erhält diesen Bag, wird aktiviert und aktiviert andere Worker über die lifelines. Aktivierung und Deaktivierung wird mit APGAS umgesetzt, indem die Berechnung in einem finish gekapselt wird. Ein finish wartet auf alle asyncs und ein async stellt einen

aktiven Worker da. Hat der Worker keine Tasks mehr und erhält keine Tasks über Work Stealing, dann deaktiviert er sich und das async endet. Wird das finish verlassen ist die Berechnung fertig. Work Stealing erfolgt über uncounted asncs und beeinflusst die Erkennung daher nicht.

Für die zufälligen Stehlanfragen gibt es zwei Optimierungen in der Implementierung. Die erste Optimierung ist es die Stehlanfragen möglichst zuerst lokal zu senden. Das bedeutet erst alle Worker auf dem gleichen place zu bestehlen und bei Misserfolg erst Worker von anderen places zu bestehlen. Diese Optimierung spart aufwendige Kommunikation über das Netzwerk. Eine weitere Optimierung ist es nicht ganz zufällig zu stehlen. Bei dem Stehlen von einem place wird jeder Worker nach seinen vorhandenen Tasks mit getCurrentTaskCount gefragt und von dem Worker mit den meisten Tasks gestohlen. Optimierungen, Anzahl der verarbeiteten Tasks n und maximale Stehlanfragen k sind Parameter von GLB.

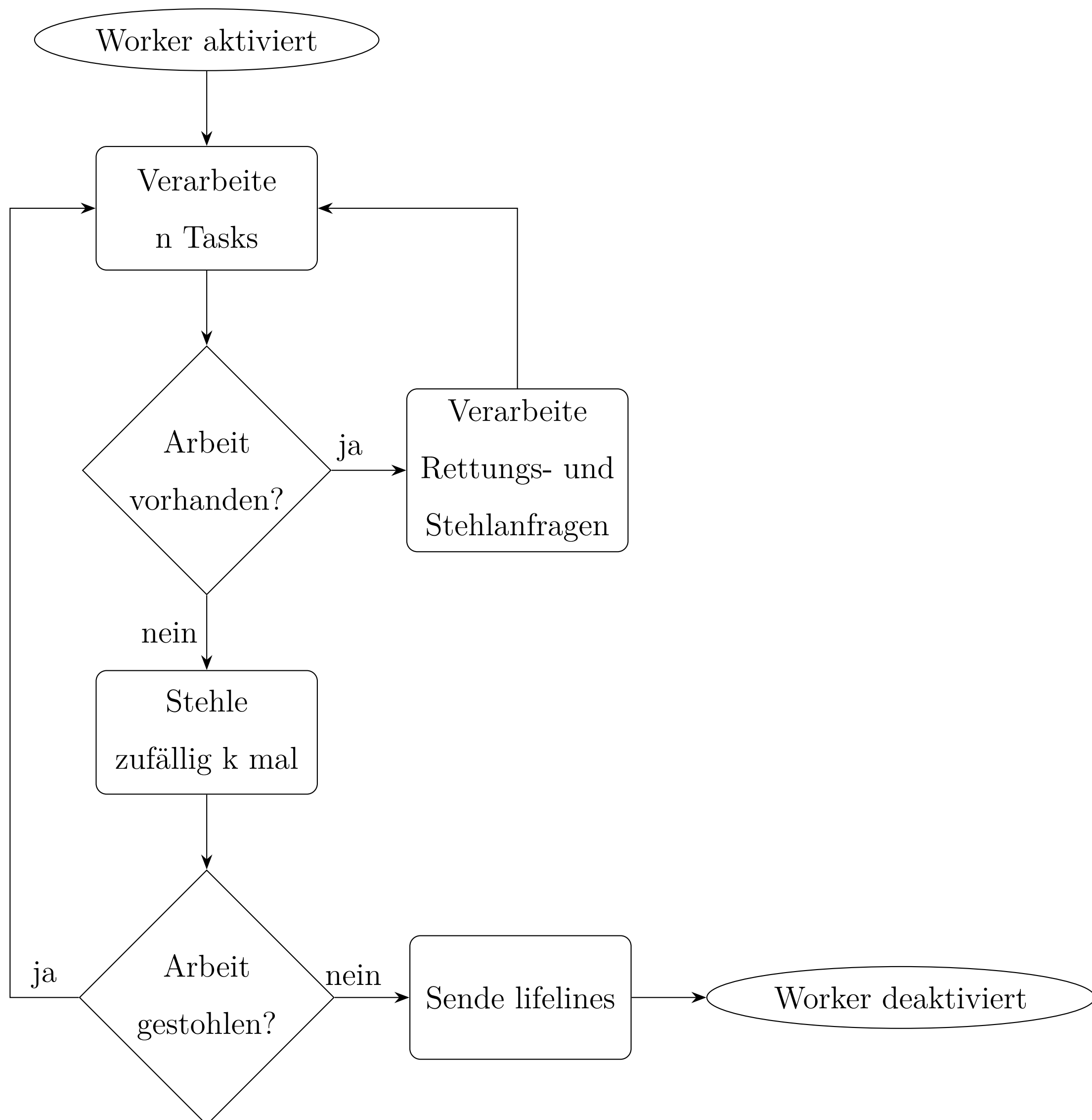

Abbildung 2.1: Ablauf eines kooperativen GLB Workers

3 SplitQueue

In Abschnitt 2.2 wurde das kooperative Work Stealing erläutert. Dieser Abschnitt beschäftigt sich mit koordinierten Work Stealing mit der Datenstruktur SplitQueue. der darauf folgende Abschnitt modifiziert die kooperative Variante von GLB zu einer koordinierten Variante mit SplitQueue.

Zuerst ist koordiniertes Work Stealing anders aufgebaut wie kooperatives. Beim koordinierten Work Stealing arbeiten die Worker zusammen, um Tasks im System zu verteilen. Der Dieb greift direkt auf die Tasks vom Opfer zu und das Opfer garantiert diesen Zugriff. Im Vergleich zum kooperativen Work Stealing ist eine Synchronisation dieser Zugriffe nötig und diese wird über die Datenstruktur SplitQueue ermöglicht. Folgend wird die Variante der SplitQueue[1] erläutert.
Der Aufbau der SplitQueue ist in der Abbildung 3.1 dargestellt. Die SplitQueue ist ein Ringbuffer mit einer festen Größe size. An der head Position werden Tasks eingefügt und alle Tasks in der SplitQueue liegen zwischen head und tail. Die split Position unterteilt die SplitQueue in die public und private Bereiche. Aus dem public Bereich werden Tasks von Dieben parallel gestohlen und der private Bereich ist für den Worker da. Bei dem Stehlen von Tasks werden die Hälfte der Tasks ausgehend von dem tail gestohlen. Tasks in dem public Bereich besitzen die Zustände Available, Claimed und Finished. Diese Zustände werden in einem separaten Array gespeichert. Die SplitQueue besitzt folgende Operationen mit T als Task:

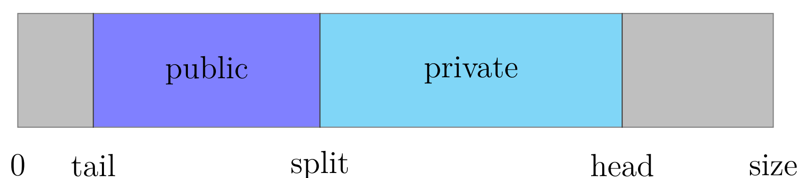

Abbildung 3.1: SplitQueue Datenstruktur, grauer Bereich ist freier Speicher

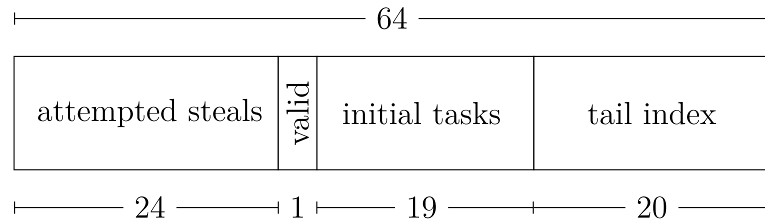

Abbildung 3.2: SplitQueue Metadaten

- push(T): fügt Tasks zu dem privaten Teil hinzu.
- T get(): entfernt 1 Task aus dem privaten Teil und gibt ihn zurück.
- void release(): verschiebt die Hälfte der Tasks von dem privaten Bereich in den public Bereich. Darf nur bei leerem public Bereich ausgeführt werden.
- void acquire(): Verschiebt die Hälfte der Tasks von dem public Bereich in den privaten Bereich. Darf nur bei leeren privaten Bereich ausgeführt werden.
- T[] steal(): stiehlt die Hälfte der verfügbaren Tasks im public Teil. Vor dem Stehlen werden zuerst die Metadaten überprüft. Diese Operation kann parallel zu anderen steal Operationen ausgeführt werden und wird durch acquire blockiert.

Der Ablauf von steal ist im Codeabschnitt 3.1 dargestellt. Das Stehlen mittels steal wird mit einem Atomic Integer, dargestellt in Abbildung 3.2, synchronisiert. Dieser Atomic Integer speichert Metadaten wichtig für das Stehlen. Gespeichert wird in attempted steals die bisherigen Stehlversuche, eine valid Flag, die initialen Tasks im public Bereich und der Index der tail Position. Ist die valid Flag nicht gesetzt wird jeder Stehlversuch abgebrochen. Mit dem Rest der Metadaten berechnen Diebe welche Tasks aus dem public Bereich zu stehlen sind. Berechnet wird über das Steal Half Prinzip, welches den public Bereich in Abschnitte unterteilt. Diese Abschnitte ergeben sich aus der Anzahl der inital Tasks und werden über wiederholte Halbierung dieser Anzahl gebildet. Aus 21

```
1 steal() {
2         get metadata and increment attempted steals
3         check valid flag
4         calculate section to steal
5         mark Tasks as Claimed
6         copy Tasks
7         mark Tasks as Finished
8         return Tasks
9     }
```

Listing 3.1: steal Operation

inital Tasks würden sich die Abschnitte mit Anzahl von Tasks gleich 10,5,3,1,1,1 ergeben. Jeder Abschnitt wird von einem Dieb gestohlen und ein Dieb errechnet seinen Abschnitt, indem die Folge von Tasks gebildet wird. Über die bisherigen Stehlversuche sieht der Dieb bereits gestohlene Abschnitte. Mit 3 bisherigen Stehlversuchen würden die Abschnitte 10,5,3 gestohlen sein und daher insgesamt 18 Tasks gestohlen und der Dieb würde den nächsten Abschnitt stehlen. Sind alle Abschnitte bereits gestohlen wird der Stehlversuch abgebrochen und der public Abschnitt ist leer. Ein Dieb beansprucht seinen Abschnitt, indem er die attempted steals inkrementiert und dadurch andere Diebe seinen Abschnitt als gestohlenen sehen. Die Tasks im Abschnitt werden als Claimed markiert, in einem Array kopiert und als Finished markiert. In dem Array sind die gestohlenen Tasks gepeichert und diese wird am Ende zurück gegeben. Die Markierungen der Tasks ist wichtig für die Operation acquire.

Mit release werden die Hälfte der Tasks aus dem privaten Bereich in den public Bereich verschoben und mit acquire werden die Hälfte der Tasks aus dem public Bereich in den private Bereich verschoben. Beide Operationen dürfen nur bei leerem Zielbereich ausgeführt werden. Für das Work Stealing müssen beide Bereiche gefüllt sein und daher ruft der Worker gelegentlich bei leeren Bereichen diese Operation auf. Bei leerem public Bereich sehen Dieben keine zu stehlenden

Tasks und brechen ihre Stehlversuche ab. So muss die release Operation nicht synchronisiert werden und der Worker verschiebt die Tasks mit einer Änderung der split Position. Nach dieser Änderung werden die Metadaten aktualisiert und Diebe können die neuen Tasks stehlen. acquire benötigt Synchronisation, da Diebe aus dem public Bereich stehlen. Mit dem Entfernen der valid Flag greife neue Diebe nicht mehr auf den public Bereich zu. Jedoch können Diebe diese Flag vor der acquire Operation überprüft haben und aktiv stehlen. Solche Diebe markieren das Ende ihres Stehlversuches, indem alle gestohlenen Tasks als Finished markiert werden. acqurie errechnet wie Diebe die Abschnitt und zählt die Anzahl von Tasks in gestohlenen Abschnitten. Diese Anzahl wird mit den tatsächlich als Finished markierten Tasks verglichen. Entsteht eine Gleichheit sind alle Diebe fertig mit dem Stehlen. Bei Ungleichheit wird erneut gezählt bis eine Gleichheit entsteht. Bei Gleichheit werden Tasks wie bei release verschoben und die Metadaten aktualisiert.

4 Koordiniertes GLB mit SplitQueue

In diesem Abschnitt wird die Implementierung von der koordinierten GLB Variante mit SplitQueue beschrieben. Die Implementierung modifiziert die Variante aus Abschnitt 2.2 mit der SplitQueue, welche aus einer vorherigen Projektarbeit stammt. Die Änderungen an der SplitQueue Variante aus Abschnitt 3 werden auch erläutert.

4.1 Implementation und Integration der SplitQueue

Die koordinierte Variante erwartet, dass die Implementierung von Bags die SplitQueue zur Speicherung von Tasks benutzt. Bags wurden um die Operation manageQueue ergänzt, welche vom Laufzeitsystem aufgerufen wird für die Balancierung der SplitQueue Bereiche über acquire und release. Die Implementierung der SplitQueue hat die Operationen put, push, release und acquire wie schon beschrieben. Dazugekommen sind folgende Operationen mit T als Task:

- pushArray(T[] toAdd): fügt das Array toAdd dem privaten Bereich wie push hinzu
- isLocalQueueEmpty(): gibt true zurück gdw. der private Bereich leer ist
- isRemoteQueueEmpty(): gibt true zurück gdw. der public Bereich leer ist
- remoteQueueSize(): gibt die Anzahl nicht gestohlener Tasks aus dem public Bereich zurück
- extend(int newSize): erhöht die Kapazität der SplitQueue auf newSize

- T[] deplete(): gibt alle Tasks aus beiden Bereichen gesammelt in einem Array zurück.

Ein leerer privater Bereich zeichnet sich durch split == tail aus und wird darüber von isLocalQueueEmpty erkannt. Ein leerer public Bereich wird durch den Status des letzten Tasks im public Bereich erkannt. Ist der Task nicht in dem public Bereich, dann ist der public Bereich garantiert leer. Ist der Task als Finished markiert, dann wurden alle Abschnitte gestohlen und der Bereich ist leer. Die Anzahl verfügbaren Tasks in dem public Bereich wird über einen Zugriff auf die Metadaten ermittelt. Mit diesen können, wie bei steal, die bereits gestohlenen Tasks ermittelt werden und die verfügbaren Tasks ergeben sich als Differenz mit den initial Tasks.

Bei der Integration der SplitQueue in die Lastenbalancierung gab es das Problem die Größe der SplitQueue am Anfang der Berechnung zu wählen. Es ist schwer abzuschätzen wie viele Tasks maximal in der SplitQueue bei einem Worker gespeichert werden. Da die SplitQueue einen begrenzten Speicher hat, muss der Speicher vor der Ausführung festgelegt werden. Vorsichtshalber wird dann die Größe der SplitQueue größer als nötig gewählt. Mit der extend Operation ist es möglich zur Laufzeit die Kapazität der SplitQueue zu erhöhen. Diese Operation kommt mit den Kosten der acquire Operation, da alle Tasks in ein größeres Array kopiert werden müssen und daher keine Diebe auf die SplitQueue zugreifen dürfen. Die dynamische Erhöhung der SplitQueue kostet daher Performance.

Bei der Implementierung der merge Operation müssen alle Tasks von einem Bag in den anderen integriert werden. Mit deplete werden zuerst alle Tasks von dem public Bereich in den private Bereich verschoben wie in acquire und darauf der private Bereich in ein Array kopiert. Dieser Array wird zurückgegeben und die SplitQueue ist nach deplete leer. Mit pushArray kann dieser Array dann dem Bag hinzugefügt werden.

4.2 Änderungen an GLB

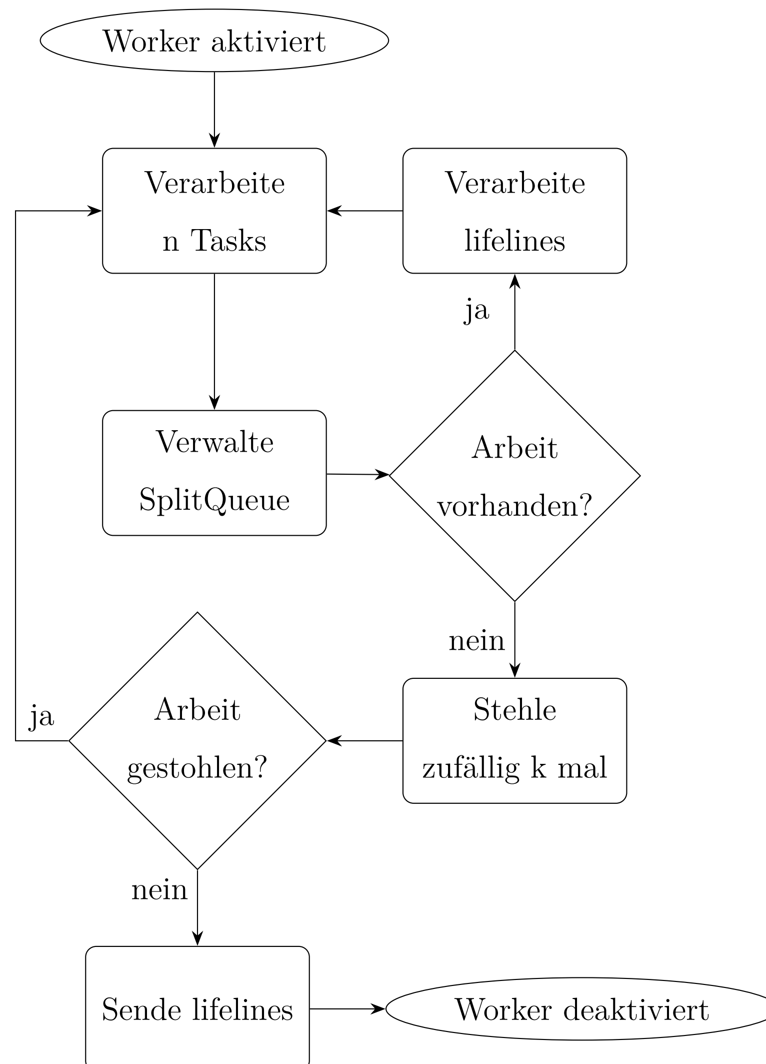

Abbildung 4.1: Ablauf eines koordinierten GLB Workers

In Abbildung 4.1 ist der Ablauf eines Workers im koordinierten Work Stealing dargestellt. Das koordinierte Work Stealing benötigt keine Stehlanfragen und die Mechanismen dafür wurden entfernt. Hinzugefügt wurde die Verwaltung der SplitQueue nach der Verarbeitung von Tasks. Die Verwaltung ruft manageQueue auf und erwartet die Balancierung der beiden Bereiche. Hat der Bag noch Arbeit ist der lokale Bereich nicht leer und Tasks können weiter mit process verarbeitet

werden. Ist der Bag hingegen leer, geht der Worker zu dem Stehlen über. Das Stehlen von Arbeit wird wie in der kooperativen Variante synchron ausgeführt, aber mit koordinierten Stehlen greift der Dieb direkt auf die SplitQueue zu und die Synchronisation wartet nur auf die Tasks vom steal. Ohne gestohlene Arbeit kontaktiert der Worker seine lifeline buddies und deaktiviert sich.

Die kooperativer Variante sichert jeden Zugriff auf Bags mit einem synchronised. Diese Synchronisation wird entfernt und ein gleichzeitiger Zugriff auf den Bag erlaubt. Die SplitQueue erlaubt parallelen Zugriff von Dieben über steal und der Worker kann während steals mit push und get auf dem privaten Bereich arbeiten. Aber Tasks von lifelines werden in den privaten Bereich eingefügt und diese Tasks können von mehreren lifeline buddies während push und get eintreffen. push und get sind nicht für gleichzeitigen Zugriff konzipiert und müssen extern abgesichert werden. Dafür werden Bags von lifeline buddies in einer ConcurrentLinkedQueue gebuffert. Bei Aktivierung von einem Worker und nach der Bearbeitung von Tasks, wo die SplitQueue verwaltet wird, wird dieser Buffer geleert. Der Buffer wird dynamisch allokiert, da die Anzahl an lifeline buddies relativ niedrig ist.

5 Ergebnisse

Auf dem Cluster der Universität Kassel wurden die kooperative und koordinierte Variante von GLB ausgemessen. Die Benchmarks UTS(Unbalanced Tree Search), BC(Betweenes Centrality) und Synthetik werden in diesem Kapitel erläutert und anschließend die Work Stealing Varianten gegenüber gestellt.

5.1 Benchmarks

Der UTS Benchmark basiert auf einer Tiefensuche in einem nicht balancierten Binärbaum. Dabei werden die Blätter des Baumes mit dem Algorithmus SHA1 aus der Kryptographie zufällig bei Ausführung generiert. Die zufällige Verteilung ergibt eine ungleichmäßige Arbeitsverteilung zwischen den Worker und eignet sich daher für den Vergleich der Lastenbalancierung zwischen den beiden Varianten. Zum Start der Berechnung wird der Wurzelknoten einem Worker zugeteilt und alle anderen Worker starten ohne Arbeit.

Der Benchmark ist detailliert mit verschiedenen Parametern konfigurierbar und wichtige Parameter sind die Höhe des Baumes, der Seed für den Zufallsgenerator und die Struktur des Baumes. Die Ausmessung hat diese Parameter fest gewählt und die Anzahl places variiert.

Der BC Benchmark berechnet in einem vorher generierten Graphen für jeden Knoten die Betweenness Centrality. Betweenness Centrality zählt wie oft ein Knoten auf dem kürzesten Weg zwischen zwei Knoten liegt und bei der Berechnung werden alle kürzesten Pfade zwischen allen Knoten errechnet. Zum Start der Berechnung hat jeder Worker einen Teil der Knotenmenge. Der Benchmark wird über die Größe des Graphen und den Seed für die Generierung

eingestellt. Wie bei dem UTS Benchmark werden die Parameter fest gehalten und die Anzahl places variiert.

Der Synthetic Benchmark ist ein künstlicher Benchmark, wo die gesamte Ausführungszeit der Arbeit vor der Ausführung festgelegt wird. Bei dem Benchmark wird eine Zeit wie 30 Sekunden eingestellt und über Messung der Ausführungszeiten einzelner Tasks wird die Dauer eines Tasks dynamisch angepasst. Folglich hat der Benchmark eine Ausführungszeit von 30 Sekunden unabhängig von dem Laufzeitsystem. Die gemessene Laufzeit ist die Summe aus der vorgegebenen Zeit von Synthetik und dem Overhead des Laufzeitsystems. Diese Eigenschaft ist Ideal für einen Vergleich der beiden Varianten vom Work Stealing. Bei dem Benchmark kann die Anzahl der Tasks pro Worker, die gesamte Ausführungszeit und eine Varianz der Taskausführungszeit eingestellt werden. Diese Werte sind fest gewählt und die Anzahl places variiert.

5.2 Konfiguration und Ausführung der Benchmarks

Die Konfiguration der Benchmarkparameter ist in Tabelle 5.1 dargestellt und

BC	N=17, seed=2
UTS	depth=17, seed=19
Synthetic	time=100000ns

Tabelle 5.1: Parameter der Benchmarks

jeder Benchmark wurde auf dem Cluster der Universität Kassel auf der Partition "FB16", welche 12 Rechner mit je 2 Intel Xeon 6-Kern Prozessoren und einem InfiniBand-Netzwerk ausgestattet ist, ausgeführt. Jeder place wurde mit 12 Workern zur Auslastung der Prozessoren konfiguriert und die Anzahl der places in 2er Schritten von 1 bis 12 erhöht.

5.3 Ergebnisse der Benchmarks

Von jedem Benchmark wurden die Laufzeiten über 10 Ausführungen gemessen. In den Abbildungen 5.2, 5.3 und 5.4 sind die Mittelwerte und die Standardabweichung von den Laufzeiten dargestellt. Die Mittelwerte sind Punkt mit einer Linie interpoliert und die Standardabweichung wird bei jedem Punkt als vertikale Linie dargestellt. Die x-Achse ist mit der Anzahl an places und die y-Achse ist mit der Zeit in Sekunden beschriftet. Die y-Achse von dem BC und UTS Bechmark ist logarithmisch skaliert. Der UTS Benchmark in

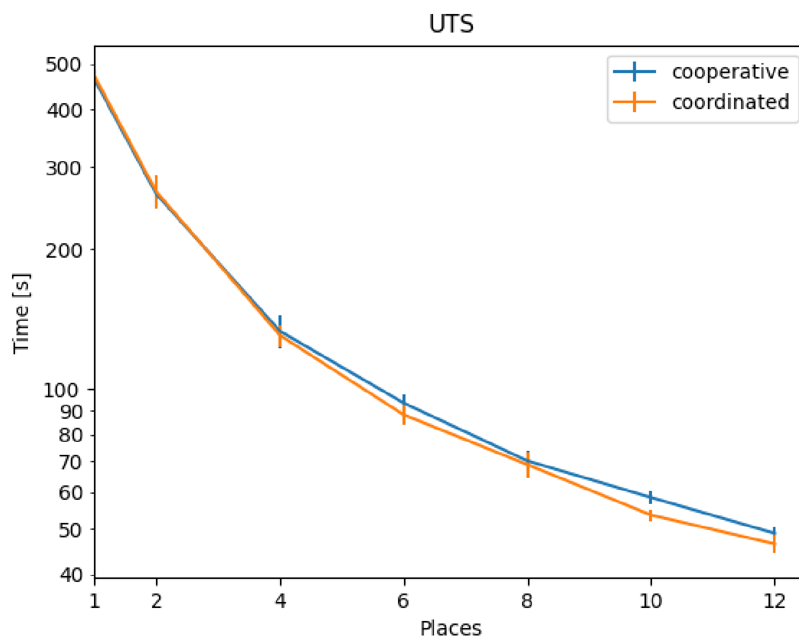

Abbildung 5.2: UTS Benchmark.

Abbildung 5.2 skaliert mit der Anzahl an places unabhängig von der Work Stealing Variante. Beide Varianten weisen eine ähnliche Standardabweichung auf und sind bei niedriger Anzahl an places ebenbürtig. Bei größerer Anzahl an places ist koordiniertes Work Stealing besser, besonders bei 10 places. Der BC Benchmark in Abbildung 5.3 skaliert ähnlich dem UTS Benchmark mit der

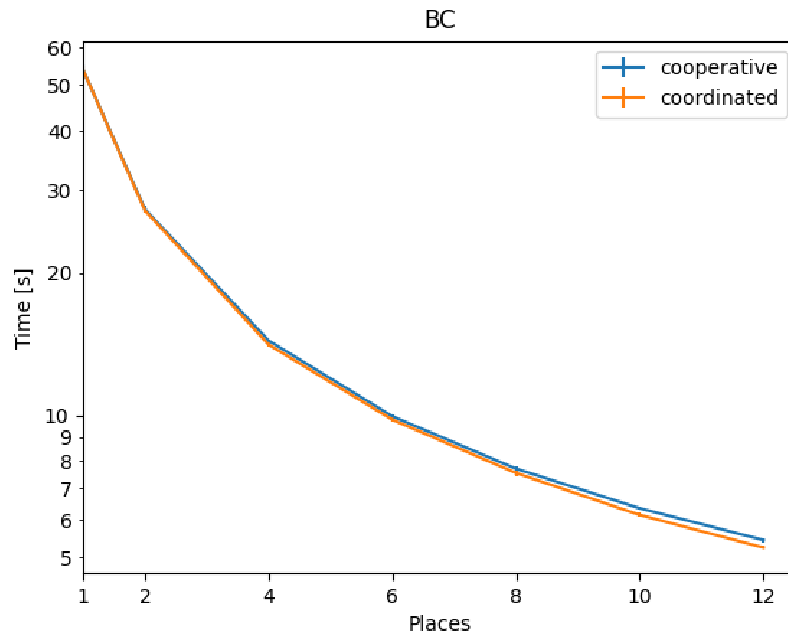

Abbildung 5.3: BC Benchmark.

Anzahl der places. Beide Varianten weisen eine niedrige Standardabweichung auf und liegen nahe beieinander. Koordiniertes Work Stealing ist immer leicht besser als kooperatives Work Stealing, aber der Unterschied ist marginal. In der Abbildung 5.4 von Synthetic Benchmark stellt den Overhead von GLB da. Die angegebene Zeit ist die tatsächliche Laufzeit minus der 100 Sekunden Laufzeit von dem Benchmark. Auffallend ist der Unterschied zwischen den beiden Work Stealing Varianten. Die kooperative Variante weist eine hohe Standardabweichung auf und hat einen deutlich höheren Overhead im Vergleich zu der koordinierten Variante. Der Overhead steigt mit jedem zusätzlichen place außer bei 6 places. Die Standardabweichung ist relativ hoch und bei nur 10 Ausführungen kann der Wert bei 6 places als Ausreißer gewertet werden. Koordiniertes Work Stealing hat eine sehr geringe Standardabweichung und der Overhead steigt langsam mit der Anzahl an places.

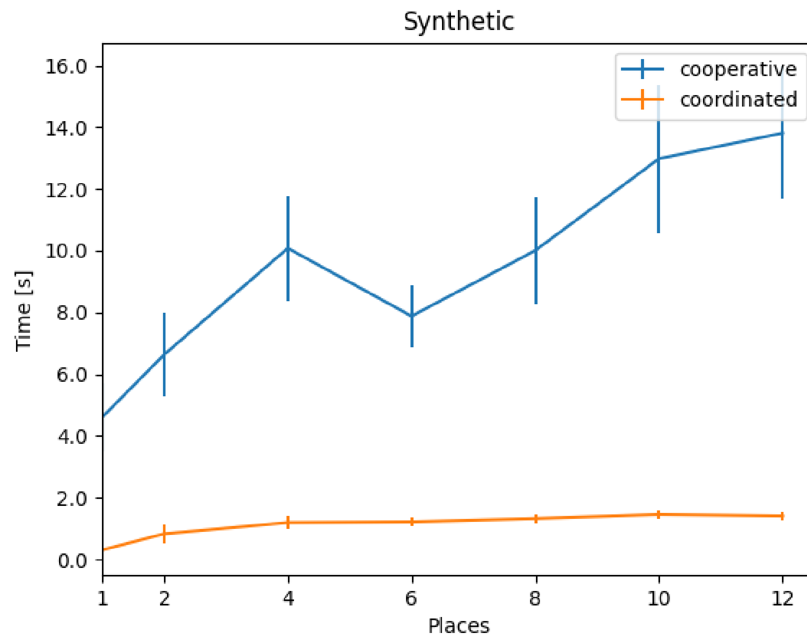

Abbildung 5.4: Synthetic Benchmark, Overhead des Laufzeitsystems.

5.4 Diskussion der Ergebnisse

Im Vergleich zwischen den Work Stealing Varianten ist koordiniertes Work Stealing leicht besser beim BC und UTS Benchmark. Der vergleichsweise größere Unterschied bei 10 und 12 places stellt einen Trend für koordiniertes Work Stealing da und sollte bei höheren place Zahlen weiter untersucht werden. Mit höherer Anzahl an places steigt der Einfluss von Work Stealing, besonders wenn wenig Arbeit im System ist. Vermutlich sinkt die Zeit ab einer place Anzahl nicht weiter, da die meisten places aktiv nach Arbeit suchen und so das Netzwerk belasten.

Interessanter für den Vergleich der Varianten ist der Syntheic Benchmark, wo die koordinierte Work Stealing Variante mit der SplitQueue klar besser ist. Durch das parallele Stehlen von Tasks während der Arbeit ist der Overhead vom Stehlen minimal und ein Worker mit Tasks in beiden Bereichen der SplitQueue muss nur die lifelines verwalten. Die teuerste Operation acquire wird zudem

selten aufgerufen, da viele neue Tasks erzeugt werden und immer die Hälfte der freigegebenen Tasks gestohlen wird. Koordiniertes Work Stealing funktioniert besser bei dem Synthetic Benchmark, weil weniger Verwaltungsaufwand über Synchronisation erforderlich ist und paralleles Stehlen möglich ist. Hingegen ist die Standardabweichung vom kooperativen Work Stealing sehr hoch, da das Stehlen von Arbeit den Worker blockiert und die Berechnung aufhält.

6 Verwandte Arbeiten

Der Vergleich zwischen den beiden Work Stealing Varianten mit der SplitQueue in GLB wurde schon im Fachgebiet von Posner in seiner Masterarbeit durchgeführt[4]. In der Arbeit wird GLB von der X10 Implementierung zu APGAS übertragen und die SplitQueue integriert. Die SplitQueue ist eine ältere Variante ohne paralleles Stehlen; der public Bereich wird mit einem Lock gesichert. Auf jedem place wird nur ein Worker ausgeführt. Basierend auf dieser Implementierung wird im Fachgebiet weiter geforscht. Ein weiterer Vergleich wurde im Fachgebiet von Posner und Fohry basierend auf ähnlicher Implementierung veröffentlicht[5].

7 Fazit

In dieser Arbeit wurde die neue Variante der SplitQueue[1] vorgestellt und in das Framework für Lastenbalancierung GLB integriert. Die Implementierung baut auf der Arbeit von Hardenbicker[3] auf und modifiziert diese von dem kooperativen Work Stealing zu dem koordinierten Work Stealing. Beide Implementierungen wurden auf dem Cluster der Universität Kassel mittels der Benchmarks BC, UTS und Synthetic ausgemessen. Die Ergebnisse zeigen eine ähnliche Performance von beiden Varianten bei BC und UTS. Im Synthetic Benchmark gewinnt der koordinierte Ansatz mit einem Vorsprung von mindestens 4 Sekunden. Der Overhead von diesem Ansatz scheint geringer zu sein und gibt Ansatz den genauen Overhead weiter zu untersuchen. Nach den Ergebnissen scheint die koordinierte Implementierung immer schneller oder gleich schnell zu sein. Fraglich bleibt wie gut dieser Vorteil auf mehr als 12 places und einer größeren Arbeitslast skaliert. Wobei die Implementierung der SplitQueue zukünftig verbessert werden kann. Zurzeit erhält jeder Tasks für acquire einen Status. Aber Tasks werden immer in einer Gruppe gestohlen und der Status der Tasks in der Gruppe ändert sich gemeinsam. Würden Gruppen den Status tragen wäre der Markierungsaufwand logarithmisch in der Anzahl der Tasks und nicht linear wie in der Implementierung. Weiterhin gibt es eine Optimierung in SplitQueue Variante[1], die versucht die Performance von acquire zu verbessern. Die Implementierung von GLB könnte das synchrone Stehlen verbessert werden. Zum Ende der Berechnung sind die meisten Stehlversuche erfolglos und anstatt auf erfolglose Versuche zu warten könnten mehrere Versuche gestartet werden. Das würde eine erhöhte Netzwerkbelastung bedeuten.

Literatur

[1] H. Cartier, J. Dinan und D. B. Larkins. „Optimizing Work Stealing Communication with Structured Atomic Operations“. In: *50th International Conference on Parallel Processing*. ICPP 2021. Lemont, IL, USA: Association for Computing Machinery, 2021. doi: 10.1145/3472456.3472522.

[2] P. Finnerty, T. Kamada und C. Ohta. „Self-Adjusting Task Granularity for Global Load Balancer Library on Clusters of Many-Core Processors“. In: PMAM ’20. San Diego, California: Association for Computing Machinery, 2020. doi: 10.1145/3380536.3380539. url: https://doi.org/10.1145/3380536.3380539.

[3] K. Hardenbicker. „Eine lokalitätsoptimierte Lastenbalancierung für Task-basierte parallele Programmiersysteme in Rechenclustern“. Bachelorarbeit. Universität Kassel, 2022.

[4] J. Posner. „Global Load Balancing and Intra-Node Synchronization with the Java Framework APGAS“. Masterarbeit. Universität Kassel, 2016.

[5] J. Posner und C. Fohry. „Cooperation vs. coordination for lifeline-based global load balancing in APGAS“. In: *Proc. ACM SIGPLAN Workshop on X10*. 2016. doi: 10.1145/2931028.2931029.

[6] V. A. Saraswat, P. Kambadur, S. B. Kodali, D. Grove und S. Krishnamoorthy. „Lifeline-based global load balancing“. In: *PPoPP ’11*. 2011.

[7] P. Suter, J. Milthorpe und O. Tardieu. „Distributed programming in Scala with APGAS“. In: Juni 2015. doi: 10.1145/2774975.2774977.