Eine lokalitätsoptimierte Lastenbalancierung für Task-basierte parallele Programmiersysteme in Rechenclustern

von Kai Hardenbicker
33380467

13.04.2022

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

Inhaltsverzeichnis

Abkürzungen

PGAS	Partitioned Global Address Space Modell
APGAS	Asynchronous PGAS Bibliothek
GLB	Lifeline-based Global Load Balancer Bibliothek[6]
J_GLB	APGAS Variante der GLB Bibliothek[4]
KobeGLB	Weiterentwicklung des J_GLB der Universität Kobe[1]
MOSGLB	Multiworker and Only Stealing GLB (neue Variante dieser Arbeit)
UTS	Unbalanced Tree Search Benchmark[2]

1 Einleitung

In der heutigen Zeit existieren viele rechenintensive Aufgaben, die von Parallelisierung stark profitieren. Dazu zählen beispielsweise verschiedene Arten von Simulationen, wie die Wettervorhersage. Allerdings ist es in der Regel schwierig, die höhere Rechenleistung, die durch parallele Nutzung von mehreren Prozessoren entsteht, voll auszunutzen.

Beim Erstellen von Programmen, die auf Multiprozessorumgebungen arbeiten, treten verschiedene Hürden auf. Wir werden uns in dieser Arbeit vor allem auf die Lastenbalancierung, also die Verteilung von der zu bewältigenden Aufgabe auf viele Prozessoren, fokussieren. Dabei wird eine Aufgabe, wie beispielsweise die Wettervorhersage, in einzelne Arbeitsschritte, die nicht weiter geteilt werden können, zerlegt. Alle Arbeitsschritte zusammen ergeben die Last, die das System bewältigen muss.

Einzelne Arbeitsschritte können verschieden hohen Rechenaufwand haben und sie können dynamisch sein. Das heißt, dass ein Arbeitsschritt während der Laufzeit neue Arbeitsschritte erzeugen kann. Daher sollte eine Lastenbalancierung nicht initial vorgenommen werden, da zur Laufzeit die Verteilung der Arbeitsschritte ungleichmäßig werden kann. Die Last muss also während der Laufzeit balanciert werden.

Um Programmierern die Lastenbalancierung zu erleichtern, wurde unter anderem von der Kobe Universität eine Global Load Balancer Bibliothek[1] (KobeGLB) entwickelt. Diese nimmt vordefinierte Arbeitsschritte entgegen und verwaltet anschließend die Parallelisierung und vor allem die Lastenbalancierung selbst.

Das Ziel dieser Arbeit ist es, eine Variante des KobeGLB zu entwickeln und sie bezüglich der Effizienz mit KobeGLB zu vergleichen. Diese Variante nennen wir Multiworker and Only Stealing GLB (MOSGLB). Sie ist zum einen durch andersartige Ausnutzung von Lokalität effizienter und zum anderen eliminiert sie work-sharing, auf welches wir später eingehen. Unter Lokalität ist in diesem Fall das Nutzen gleicher Hardware im Sinne eines Rechners in der Multiprozessorumgebung zu verstehen. Dazu zählen Speicherzugriffe und prozessübergreifende

Kommunikation.

MOSGLB und KobeGLB, nutzen die APGAS Bibliothek. Diese arbeitet mit dem ”Partitioned Global Address Space” (PGAS) Modell, welches in einem Multiprozessorsystem den Arbeitsspeicher aller Prozessoren durch einen globalen Adressbereich darstellt, sodass jeder Prozessor auf den Speicher anderer Prozessoren zugreifen kann. Lokale Zugriffe sind jedoch deutlich schneller. Mit Place bezeichnen wir typischerweise einen Prozessor mit seinem Speicher.

APGAS steht für Asynchronous PGAS und fügt dem Modell Asynchronität hinzu. Diese erlaubt einem Place Aktivitäten zur Laufzeit auf anderen Places oder lokal zu starten. Ein Algorithmus wird auf Place 0 gestartet und sendet seine Aufgaben durch asynchrone Aufrufe über das Netzwerk an andere Places, die initial im Leerlauf gestartet wurden.

Auf einem Place können mehrere Prozesse parallel ausgeführt werden. MOSGLB und KobeGLB starten mehrere Prozesse, die wir Worker nennen. Die Worker erhalten jeweils einen Teil der Arbeitsschritte und arbeiten diese ab. Wie die Worker jeweils bei MOSGLB und KobeGLB arbeiten und worin sie sich unterscheiden, werden wir im späteren Verlauf der Arbeit sehen.

KobeGLB nutzt als Mittel zur Lastenbalancierung work-sharing und work-stealing. Für work-sharing gibt es auf jedem Place eine Warteschlange, in die ein Worker des Places Arbeitsschritte ablegen kann. Wenn ein Worker keine Arbeit mehr hat, versucht er zuerst aus dieser Warteschlange Arbeitsschritte zu nehmen.
Ist diese leer, fährt der Worker mit work-stealing fort. Dazu versucht er zunächst von zufälligen Places zu stehlen. Falls das fehlschlägt, versucht er vordefinierte Places zu bestehlen, wobei er sich schlafen legt, falls er eine negative Antwort erhält. Sobald einer der vordefinierten Places dem Worker Arbeitsschritte liefern kann, wird er wieder geweckt.

MOSGLB hingegen nutzt kein work-sharing. Wenn ein Worker keine Arbeit mehr hat, versucht er stattdessen als Erstes von dem Worker auf dem gleichen Place zu stehlen, der die meisten Arbeitsschritte hat. Anschließend verfährt er beim restlichen work-stealing analog zu KobeGLB.

Im Folgenden werden wir untersuchen, ob KobeGLB durch work-sharing und work-stealing weniger effizient ist als MOSGLB, welches nur work-stealing benutzt. Bei der Effizienzuntersuchung spielt vor allem die Ersetzung von work-sharing durch das zuvor beschriebene lokale work-stealing in MOSGLB eine Rolle.

Die nachfolgende Arbeit ist in sieben Abschnitte gegliedert. In Abschnitt 2 werden die APGAS-Bibliothek und KobeGLB vorgestellt. Danach stellen wir in Abschnitt 3 den im Rahmen dieser Arbeit entstandenen MOSGLB vor und erläutern die Unterschiede zu KobeGLB. Anschließend werden in Abschnitt 4 wichtige Aspekte der Implementierung des MOSGLB gezeigt und in Abschnitt 5 vergleichen wir die Laufzeit von KobeGLB und MOSGLB. Dazu nutzen wir den Unbalanced Tree Search Benchmark. In Abschnitt 6 werden wir verwandte Arbeiten betrachten. Abschließend fassen wir die Ergebnisse der Arbeit in Abschnitt 7 zusammen und bewerten sie.

2 Hintergrund

2.1 APGAS Bibliothek

Um Berechnungen auf dem Cluster verteilt auszuführen, nutzen wir die Java-Bibliothek APGAS. Diese wird sowohl von KobeGLB, als auch von MOSGLB genutzt, um die Ausführung auf einem Cluster zu parallelisieren. APGAS hat eine doppelte Bedeutung. Zum einen steht es für das APGAS Modell, welches in der Einleitung beschrieben wurde und zum anderen steht es für die Bibliothek, die das Modell implementiert.

Mit Place wird in APGAS ein partitionierter Speicher bezeichnet, sodass asynchrone Tasks innerhalb eines Places sich den Speicher teilen. In unserem Fall kann dies analog zu einem Prozessor mit lokalem Arbeitsspeicher angesehen werden. Ein Place wird in APGAS durch eine JVM realisiert.

In der APGAS Bibliothek werden Places als Objekte vom Typ Place dargestellt. Die wichtigsten Methoden der APGAS Bibliothek für einen Place p und eine Funktion (SerializableJob) f sind:

- async(f): Erzeugt einen parallelen Task, der f ausführt, auf dem aktuellen Place.
- at(p, f): Führt auf synchrone Art auf dem entfernten Place p die Funktion f aus.
- asyncAt(p, f): Kombiniert async und at.
- finish(f): Wartet auf die Beendigung aller Tasks, die innerhalb des Codeblocks erzeugt werden. Darunter fallen auch geschachtelte asynchrone Aufrufe.
- uncountedAsyncAt(p, f): Wie asyncAt, wird aber nicht von finish verfolgt.
- here(): Gibt den aktuellen Place zurück
- place(ID): Gibt den Place mit der entsprechenden ID zurück.
- places(): Gibt die Liste aller Places zurück

2.2 J_GLB Framework

In der parallelen Programmiersprache X10 wurde das Lifeline-based Global Load Balance Framework (GLB) entwickelt, welches das Ziel hat eine Lastenbalancierung zu implementierten. [6]

J_GLB ist eine Java-Implementierung von GLB. Beide implementieren das Taskpool-Pattern, das heißt eine zu bewältigende Aufgabe wird in möglichst viele Tasks aufgeteilt, welche von einer festen Anzahl Workern abgearbeitet werden sollen. Jeder Worker enthält einen lokalen Taskpool, der einen Teil der gesamten Tasks enthält. Die Worker entnehmen einzeln die Tasks aus ihren Taskpools und arbeiten sie sukzessive ab. Hat ein Worker in seinem Taskpool keine Tasks mehr, versucht er Tasks von anderen Workern zu stehlen.

In beiden Implementierungen arbeiten die Worker die Tasks in drei Schritten ab:

1) Solange Tasks im Taskpool vorhanden sind, arbeitet der Worker n Tasks als Bündel ab und beantwortet zwischen den Bündeln, mögliche Stehlanfragen.

2) Sobald der Taskpool des Workers leer ist, versucht er auf 2 Arten Tasks von anderen Workern zu stehlen. Zunächst versucht er bis zu z zufällige Worker zu bestehlen. Sollten diese Worker keine Tasks bereitstellen können, so versucht der stehlende Worker von seinen Partnern, die in der Lifeline definiert sind, zu stehlen. Die Lifeline ist ein Graph, der jedem Worker w feste Partnern zuweist. Die Lifeline funktioniert gleich wie die von KobeGLB und wird in Abschnitt 2.3.2 genauer erklärt.

3) Sind alle Worker mit ihren Tasks fertig, so terminiert das Programm und das Ergebnis wird mittels Reduktion zu dem finalen Ergebnis zusammengeführt.

Auf die Stehlversuche wollen wir genauer eingehen. Einen Task, der versucht zu stehlen, nennen wir Dieb. Er versucht als Erstes von einem zufälligen anderen Worker Arbeit zu stehlen. Dabei greift er nicht direkt auf den Taskpool des Workers zu, sondern er trägt sich in eine Liste von Dieben ein. Dadurch kann der zu bestehlende Worker seine aktuellen Tasks abarbeiten und daraufhin die Stehlanfrage beantworten. Wenn der Worker Tasks abgeben kann, sendet er dem Dieb einen Teil seiner Tasks und der Dieb beendet das Stehlen. Andernfalls teilt er dem Dieb mit,

dass die Stehlanfrage abgelehnt wurde und der Dieb versucht es bei einem weiteren zufälligen Worker.

Sind die zufälligen Stehlanfragen z oft fehlgeschlagen, versucht der Dieb über die Lifeline zu stehlen. Der Dieb stellt dabei eine Anfrage an alle w Partner. Kann ein Partner die Anfrage nicht erfüllen, benachrichtigt er den Dieb darüber, merkt sich die Anfrage aber weiterhin. Er erfüllt die Anfrage später, wenn er selber wieder genug Arbeit hat. Das kann entweder durch generierende Tasks oder durch eigene Stehlanfragen passieren.

Bei J_GLB wird ein Worker pro Place gestartet. Dieser Worker kann durch mehrere Tasks den lokalen Taskpool abarbeiten.

2.3 KobeGLB

2.3.1 Überblick

Von der Kobe Universität wurde die Self-adjusting Task Granularity for Global Load Balancer Bibliothek (KobeGLB) entwickelt. [1] KobeGLB ist ebenfalls eine Implementierung des GLB in Java und nutzt dazu APGAS. Bei J_GLB wurde pro Place nur ein Worker gestartet. Das führte unter anderem zu einem hohen Kommunikationsaufwand. Dahingegen startet KobeGLB mehrere Worker pro Place.

2.3.2 Die Lifeline

Die Lifeline kann als gerichteter Graph zwischen den Places verstanden werden. Diesen Graph nennen wir von nun an Lifeline-Graph. Die Knoten des Graphen repräsentieren die Places und die Kanten beschreiben die sogenannten Lifeline-Partner.

Die Lifeline-Partner werden bei der Lastenbalancierung genutzt, um feste Partner zwischen den Places zu haben. Der Vorteil an festen Partnern ist, dass wir Eigenschaften des Graphen ausnutzen können.

Der Lifeline-Graph ist zusammenhänged, das heißt zwischen je zwei Knoten existiert ein Pfad, der die beiden Knoten verbindet. Wenn das nicht der Fall wäre, könnte

es passieren, dass einzelne Places keine Arbeit mehr erhalten, da ihre Stehlanfragen nicht mehr beantwortet werden können.

2.3.3 Lastenbalancierung

In der Einleitung haben wir die Lastenbalancierung von KobeGLB grob vorgestellt. Nun wollen wir sie genauer betrachten. Um die Arbeitsschritte auf den einzelnen Places zu verteilen, nutzt KobeGLB work-sharing und work-stealing.

Das Ziel des work-sharing ist es, innerhalb eines Places die Arbeitsschritte unter den Workern aufzuteilen. Da jeder Worker seinen eigenen Pool an Arbeitsschritten hat, müssen sich die Worker aktiv austauschen. Dazu gibt es auf jedem Place eine Warteschlange, die intra-queue genannt wird, mit der die Worker eines Places Arbeitsschritte austauschen können.

Ist die intra-queue leer und kann ein Worker des Places einen Teil seiner Arbeitsschritte abgeben, so legt er einen Teil in der Queue ab. Hat ein Worker keine Arbeitsschritte mehr, versucht er als erstes aus der intra-queue Arbeitsschritte zu entnehmen. Da die intra-queue auf dem selben Place liegt, erhält der Worker die Arbeitsschritte schneller, als wenn er sie von einem anderem Place erhält.

Da alle Worker des Places Arbeitsschritte ablegen können, ist die intra-queue nur leer, wenn ein Worker gerade Arbeitsschritte entnommen hat und noch kein anderer Worker welche abgelegt hat oder wenn kein Worker des Places einen Teil seiner Arbeitsschritte abgeben kann.

Kann ein Worker keine Arbeitsschritte aus der intra-queue entnehmen, fährt er mit work-stealing fort. Dabei versucht der Worker auf zwei Arten von einem anderen Place zu stehlen. Als Erstes versucht er es bei zufälligen Places.

Bei den zufälligen Stehlanfragen wird eine Anfrage geschickt und der zu bestehlende Place, den wir von nun an Opfer nennen, prüft, ob er einen Teil seiner Arbeitsschritte abgeben kann. Ist dies der Fall, sendet er dem Dieb diesen Teil. Kann er die Anfrage nicht positiv beantworten, benachrichtigt er den Dieb darüber.

Der Dieb versucht nacheinander bei bis zu z zufälligen Workern Arbeitsschritte zu stehlen. Hat er bei einem Worker Erfolg, arbeitet er die erhaltenen Arbeitsschritte

ab und beendet den Stehlvorgang. Werden jedoch alle Versuche abgelehnt fährt der Worker mit den Lifeline-steals fort. In jedem Fall wird die Anfrage bei den Opfern nach der Beantwortung gelöscht.

Versucht der Worker über die Lifeline von einem anderen Place zu stehlen, so läuft es grundsätzlich ähnlich zu den zufälligen Stehlanfragen ab. Allerdings legt sich der Worker bei einer negativen Antwort schlafen. In diesem Fall merkt sich der Lifeline-Partner die Anfrage und beantwortet sie zu einem späteren Zeitpunkt, wenn er genug Arbeitsschritte hat.

2.3.4 Ablauf auf einem Place

In diesem Abschnitt wollen wir den Ablauf auf einem Place zusammenfassen. Begriffe, die in diesem Abschnitt verwendet werden, sind von der Implementierung losgelöst zu verstehen. Dazu gehört beispielsweise der Begriff 'Queue', welcher hier eine allgemeine Warteschlange darstellt.

Die Hauptschleife eines Places ist wie folgt aufgebaut:

Listing 1: Hauptschleife KobeGLB

```
1 do {
2     do {
3         1. start workerActivity
4         2. wait for workerActivity to finish
5         3. random stealing
6     } while(steal successful)
7     4. lifeline stealing
8 } while(steal successful)
9 5. sleep
```

Bei Schritt 1 wird ein Worker mit allen Arbeitsschritten gestartet. Dieser startet möglichst viele weitere Worker auf dem Place und arbeitet dann die Arbeitsschritte ab. Den Ablauf werden wie im Anschluss genauer betrachten.

Mit der APGAS Methode 'finish()' wird in Schritt 2 sichergestellt, dass alle Arbeitsschritte auf dem Place bearbeitet wurden. Schritt 3 und 4 beschreiben die oben

genannte Lastenbalancierung. Sollten diese nicht erfüllt werden, so wird der Place in Schritt 5 schlafen gelegt.

Die Hauptmethode des Workers, die in Schritt 1 aufgerufen wurde ist wie folgt aufgebaut:

```
1  void workerActivity(Bag workerbag){
2      do {
3          do {
4              1. start new worker, if possible
5              2. fill intra-queue, if needed/possible
6              3. fill inter-queue, if needed/possible
7              4. answer lifeline thieves
8              5. process n tasks
9          } while(has tasks)
10         6. try to take work from intra-queue
11         if failed
12             7. steal from inter-queue
13     } while(has tasks)
14 }
```

Der Worker arbeitet solange er noch Arbeitsschritte, die in dem workerBag gespeichert sind, hat. In Schritt 1 versucht er einen Teil seiner Arbeitsschritte einem neuen Worker zu geben und diesen zu starten. Das kann er nur machen, wenn die Menge seiner Arbeitsschritte teilbar ist und auf dem Place nicht die maximale Anzahl an paralleler Worker gestartet wurde.

Die intra-queue wird für das work-sharing genutzt und speichert die Arbeitsschritte eines Places ab. Ist die intra-queue leer und die Menge der Arbeitsschritte des Workers ist nach Schritt 1 weiterhin teilbar, legt der Worker Arbeitsschritte in die intra-queue ab.

Die inter-queue wird für die Lastenbalancierung zwischen den Places verwendet und bei Bedarf gefüllt. Diesen Schritt werden wir hier nicht weiter ausführen.

In Schritt 4 werden die Anfragen beantwortet, die über die Lifeline gestellt wurden.

Dazu wurden sie zwischengespeichert.

Schließlich werden in Schritt 5 n Arbeitsschritte abgearbeitet. Sobald der Worker keine Arbeitsschritte hat, versucht er welche aus der intra-queue zu entnehmen und nutzt so das work-sharing. Kann er keine Arbeitsschritte erhalten, beginnt er mit dem work-stealing.

Über das finish, welches am Anfang des Places aufgerufen wird, werden die asynchronen Aktivitäten der Worker verfolgt. Daher legt sich ein Place erst schlafen, wenn alle Worker ihre Arbeitsschritte abgearbeitet haben und die intra-queue und die inter-queue leer sind.

3 Neue Variante von GLB

Die neue Variante MOSGLB ist ähnlich zu KobeGLB. Im folgenden Abschnitt untersuchen wir deshalb die Unterschiede zwischen KobeGLB und MOSGLB. Wir werden feststellen, dass die wesentlichen Unterschiede in der Lastenbalancierung und in der Nutzung der Lokalität liegen.

3.1 Mehrere Worker

Sowohl KobeGLB als auch MOSGLB nutzen auf einem Place mehrere Worker, die die Arbeitsschritte abarbeiten. Allerdings werden in MOSGLB die einzelnen Worker stärker getrennt. In MOSGLB existiert auf jedem Place genau ein Handler. Der Handler übernimmt die Aufgabe die Worker untereinander kommunizieren zu lassen und die Worker zu organisieren. Das heißt er startet den ersten Worker und sorgt dafür, dass das Ergebnis am Ende der Ausführung gesammelt wird.

Bei der Trennung in Worker und Handler übernehmen die Worker zwei wesentliche Aufgaben. Die Hauptaufgabe der Worker ist das Erledigen der Arbeitsschritte, wobei sie zusätzlich auf die Synchronisierung der Arbeitsschritte achten müssen. Das heißt, wenn sie über den Handler von einem anderen Worker Arbeit erhalten, müssen sie sicherstellen, dass ihr Taskpool die Arbeit erhält.

3.2 Lastenbalancierung

Die Lastenbalancierung bestand im KobeGLB aus work-sharing und work-stealing. Die Lastenbalancierung bei MOSGLB funktioniert ähnlich wie die bei KobeGLB. MOSGLB nutzt jedoch ausschließlich Verfahren, die stehlen, und erweitert dafür das work-stealing.

Das work-sharing, wie es in KobeGLB genutzt wird, ersetzen wir im MOSGLB durch einen weiteren Schritt im work-stealing. Dazu kann ein Handler die Worker nach der Anzahl der zu erledigenden Arbeitsschritte fragen. Der Handler kann so den Worker ermitteln, der die meisten zu erledigenden Arbeitsschritte hat.

Wenn ein Worker seine Arbeitsschritte erledigt hat, versucht er als erstes über den

Handler von dem Worker mit den meisten zu erledigenden Arbeitsschritten auf dem gleichen Place zu stehlen. Diesen Schritt nennen wir lokales work-stealing.

3.3 Lokalitätsoptimierung

Da zwei Places in der Regel auf physikalisch verschiedenen Maschinen laufen, ist die Kommunikation innerhalb eines Places weniger aufwendiger, als die zwischen zwei Places. Das wollen wir bei der Lokalitätsoptimierung ausnutzen.

Beim lokalen work-stealing fragt ein Worker zuerst den Handler, ob auf diesem Place ein Worker existiert, der noch genug Arbeitsschitte hat. Das führt dazu, dass der Worker in der Regel seine Arbeitsschritte von einem Worker auf dem gleichen Place bekommt. Allerdings kann dies auch dazu führen, dass alle Worker des Places nahezu gleichzeitig ihr Arbeit beenden und neue Arbeit anfordern.

Dann wird zum einen sehr oft nach dem besten Worker gesucht und zum anderen viele Anfragen an entfernte Places gestellt. Die Suche nach dem besten Worker ist durch Nutzung von flüchtigen Variablen mit wenig Aufwand verbunden.

Ein weiterer Punkte, der die Lokalität ausnutzt, ist das weiterleiten von zufälligen Stehlanfragen. Wenn ein zufälliger Stehlversuch fehlschlägt, wird der Dieb nicht sofort informiert, sondern der Handler sucht den besten Worker auf dem Place des Opfers und versucht von ihm Arbeit zu nehmen.

Durch die Weiterleitung der Anfrage wird die Kommunikation zwischen Places reduziert. Denn ein zufälliger Stehlversuch wird nur dann abgelehnt, wenn alle Worker auf dem Place nicht genug Arbeitsschritte haben.

Statt bei zufälligen Stehlversuchen zufällige Worker zu wählen, könnte man die Stehlanfrage direkt an den Handler stellen. Der Handler kann die Anfrage dann an den Worker mit den meisten Arbeitsschritten weiterleiten. Das würde dazu führen, dass einzelne Stehlanfragen schneller beantwortet werden, da pro Place nur ein Worker angefragt wird. Zusätzlich bekommt der Dieb im Durchschnitt mehr Arbeitsschritte.

Allerdings können Arbeitsschritte je nach gestellter Aufgabe stark variieren. Das heißt, ein Arbeitsschritt kann je nach Problem auch beliebig viele weitere Arbeits-

schritte erzeugen, sodass deren Anzahl weniger wichtig ist.

Daher ist die Entscheidung, ob wir die Anfrage an einen Worker oder an den Place stellen, abhängig vom gegeben Problem.

3.4 Kommunikation zwischen den Workern

In Abbildung 1 sind die Beziehungen zwischen Place, Handler und Worker schematisch dargestellt. In der Abbildung sind Worker mit W und Handler mit H gekennzeichnet. Ein Place wird durch ein Rechteck, welches einen Handler und seine Worker beinhaltet, dargestellt. Die Linien, die die Worker mit den Handlern und die Handler untereinander verbinden, stellen den Kommunikationsfluss dar. Dabei können die Worker und Handler auf einem Place direkt kommunizieren. Die Handler nutzen untereinander das Netzwerk mittels APGAS.

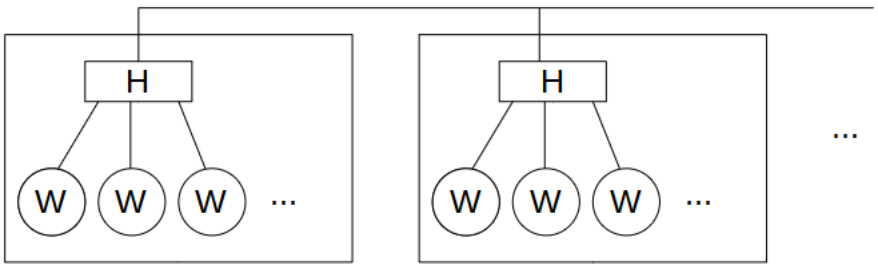


Abbildung 1: Beziehung: Place, Handler, Worker

Wenn zwei Worker miteinander kommunizieren wollen, müssen sie dazu immer den Handler nutzen. Der Handler kann jedoch nur mit seinen Workern direkt reden und muss, wenn er mit anderen kommunizieren will, sich mit den anderen Handlern austauschen.

4 Implementierung

4.1 Abstraktion für den Programmierer

Im Folgenden gehen wir genauer auf die Eingabe der Arbeitsschritte und die zu implementierenden Konstrukte ein. Die Arbeitsschritte werden in Bags (B) organisiert, das heißt ein Bag kann beliebig viele Arbeitsschritte enthalten und verschiedene Operationen ausführen. Das Ergebnis der Berechnung wird in Result (R) gespeichert. Die Operationen, die ein Bag ausführen kann sind:

- `void process(int, R)`: Hierbei wird eine bestimmte Anzahl von Arbeitsschritten abgearbeitet, die durch den ersten Parameter gegeben ist. Im zweiten Parameter (Result) können Lösungen ausgelesen oder gespeichert werden. Result wird dabei innerhalb eines Places geteilt.
- `B split(boolean)`: Es wird versucht einen Teil der Arbeitsschritte einem neu erstellten Bag abzugeben. Ist dies erfolgreich, so wird der neue Bag zurückgegeben. Können die Arbeitsschritte nicht geteilt werden, wird je nach Wert des Parameters, der gesamte Bag oder ein leerer Bag zurückgegeben.
- `void merge(B)`: Die Arbeitsschritte, die in dem Bag des Parameters liegen, werden von dem ausführenden Bag absorbiert.
- `boolean isEmpty()`: Gibt ”Wahr” genau dann zurück, wenn der Bag keine Arbeitsschritte enthält.
- `boolean isSplittable()`: Gibt ”Wahr” genau dann zurück, wenn der Bag seine Arbeitsschritte teilen kann.
- `void submit(R)`: Diese Methode wird am Ende der Berechnung, in der Phase, in der die Ergebnisse gesammelt werden, aufgerufen. Sie erlaubt es dem Bag zu dem Result im Parameter beizutragen.

4.2 Arbeitsabwicklung der Worker

Listing 2: Ablauf der Process-Methode

```
1  void process() {
2    do {
3      do {
4        1. process n tasks from its bag
5        // 2 Balance load
6        2.a Check if a lifeline requested some loot
7        2.b Answer random steals
8      } while (!bag.isEmpty()); // 3. Repeat previous
         steps until the bag becomes empty.
9      // try to steal some work
10     1. try steal from this place
11     Wait for answer
12     2. try random steals
13     Wait for answer
14     3. try lifelinesteal
15     // do not wait for answer, sleep instead
16   } while (true);
17 }
```

Der Lifelinethread hat im KobeGLB die Kommunikation über die Lifelines verwaltet. Er wird durch direkte Anfragen über die Handler ersetzt. Das Opfer merkt sich dann die Anfrage und beantwortet sie, sobald es das aktuelle Arbeitspaket abgearbeitet hat.

Durch die Umstellung von prozessorientierter zu objektorientierter Verwaltung der asynchronen Abschnitte können die intra-queue und die inter-queue ebenfalls entfernt werden. Diese werden durch eine Anfragen-queue, die beide Anfragetypen speichert, ersetzt. Diese Queue erlaubt es dem Worker sein aktuelles Arbeitspaket abzuarbeiten, bevor er die Anfrage beantwortet.

4.3 Kommunikation zwischen den Workern

In Abschnitt 3.4 haben wir mit Abbildung 1 die Beziehungen zwischen Place, Handler und Worker betrachtet. Dabei haben wir gesehen, dass zwei Worker, wenn sie miteinander kommunizieren wollen, immer den Handler nutzen müssen.

Wir wollen nun den Ablauf der Stehlanfragen am Beispiel des zufälligen Stehlens genauer betrachten. Dabei werden Methoden etwas verkürzt dargestellt, da wir nicht das Logging und try-catch-Methoden berücksichtigen. Wir befinden uns in der Situation, dass der Worker seine Arbeitsschritte abgearbeitet hat und er von seinem Place keine erhalten konnte.

Er wählt einen zufälligen Worker aus und ruft bei seinem Handler die steal-Methode auf. Dazu erhöht er ein CountDownLatch, welches jeder Worker individuell als globale Variable hat, um asynchrone Anfragen mit einem Timeout versehen zu können. Die steal Methode leitet die Anfrage weiter.

```
1 // On thief side, in Handler
2 public void steal(
3   int victimPlace, int victimID, int thiefID) {
4   asyncAt(place(victimPlace), () -> {
5     dealWithThief(thiefID, victimID)
6   });
7 }
```

Auf dem zu bestehlenden Place leitet der Handler die Anfrage direkt an den Worker weiter.

```
1 // On victim side, in Handler
2 public void dealWithThief(
3     int thiefID, int victimID) {
4   GLBWorker worker = workers[victimID];
5   worker.dealWithThief(thiefID, true);
6 }
```

Der Worker prüft zunächst, ob er gerade Arbeitsschritte erledigt, indem er eine Variable isProcessing, die vom Typ AtomicBoolean ist, abfragt. AtomicBoolean

stellt dabei sicher, dass der Wert über verschiedene asynchrone Abschnitte gleich ist. Ist der Wert von isProcessing 'Falsch', das heißt der Worker hat keine Arbeitsschritte mehr, leitet er die Anfrage direkt an den Handler zurück.

```
1 // On victim side, in Worker
2 public void dealWithThief(
3       int thiefId, boolean isRandomSteal) {
4   if(!isProcessing.get()) {
5     myWorkerHandler.moveInquiryOrAnswer(thiefId,
        myWorkerID);
6     return;
7   }
8   inquiry-queue.add(-thiefId-1);
9 }
```

Der Handler versucht, wenn der Worker keine Arbeitsschritte liefern konnte, einen besseren Worker auf dem Place zu finden. Wenn das nicht möglich ist, ist der Stehlversuch fehlgeschlagen und der Dieb wird darüber informiert. Dabei nutzt er die Methode findBestWorkerHere, welche über flüchtige Variablen der Worker iteriert. Die flüchtigen Variablen stellen nicht sicher, dass sie threadübergreifend gleich sind. Daher dienen sie dem Handler nur als Richtwert, haben aber dafür fast keinen zusätzlichen Aufwand. Mit notifyWaitingThief wird das CountDownLatch auf dem Place des Diebes reduziert und damit dem Dieb mitgeteilt, dass der Stehlversuch erfolglos war.

Listing 3: moveInquiryOrAnswer

```
1 // On victim side, in Handler
2 public void moveInquiryOrAnswer(
3     int thiefId, int oldVictimId) {
4   int bestWid = findBestWorkerHere();
5   if(bestWid != -1 && bestWid != oldVictimId) {
6     workers[bestWid].dealWithThief(thiefId, true);
7   } else {
8     notifyWaitingThief(thiefId);
```

```
9     }
10 }
```

Ansonsten wird die Anfrage in der Warteschlange inquiry-queue abgelegt, um sie zwischen den Arbeitsschritten zu beantworten. In der inquiry-queue werden die zufälligen Stehlanfragen neagtiv und um 1 verschoben abgespeichert, um sie von den Lifelineanfragen unterscheiden zu können.

Der 2. Schritt in Listing 2 beschreibt die Lastenbalancierung innerhalb der process-Methode. Dabei ist zu berücksichtigen, dass die inquiry-queue initial mit allen LifelinePartnern gefüllt wird, um die Arbeitsschritte am Anfang durch das Netzwerk zu verteilen. Die inquiry-queue ist vom Typ ConcurrentLinkedQueue<Integer>. Das heißt, sie wird wie AtomicBoolean, über Threads hinweg gesichert und die Elemente sind verknüpft. Die Ordnung auf der Queue ist FIFO (first-in-first-out), das heißt das Element, welches am längsten in der Queue ist, steht am Anfang der Queue. Dadurch werden die Anfragen der Reihe nach beantwortet.

Der Worker ermittelt zunächst, ob es ein zufälliger Stehlversuch war und wer Place und Worker des Diebes sind. Anschließend versucht er seine Arbeitsschritte mit der Methode loot(), welche die Arbeitsschritte mit synchronized schützt, zu teilen. Synchronized schützt einen Codeblock, indem es ein Javaobjekt überwacht. Ein Block zu diesem Objekt wird immer exklusiv abgearbeitet.

Konnten sie erfolgreich geteilt werden, wird dem Handler der Loot, welcher die Arbeitsschritte beinhaltet, übergeben. War ein zufälliger Stehlversuch nicht erfolgreich, wird dem Handler mitgeteilt einen besseren Worker auf dem Place zu finden. Siehe dazu Listing 3.

```
1  // On victim side, in Worker
2  if (!inquiry-queue.isEmpty() && latestWasSplittable) {
3    int id = inquiry-queue.peek();
4    boolean isRandomSteal = id < 0;
5    int thiefId = (isRandomSteal ? -id-1:id);
6    int thiefPlace = myWorkerHandler.getPlaceById(
       thiefId);
7    B loot = loot();
```

```
8     if (loot != null && !loot.isEmpty()) {
9       myWorkerHandler.dealAtWorker(thiefPlace, thiefId,
            loot);
10      inquiry-queue.remove(id);
11    } else {
12      if(isRandomSteal) {
13        myWorkerHandler.moveInquiryOrAnswer(thiefId,
              myWorkerID);
14        inquiry-queue.remove(id);
15      }
16    }
17  }
```

Die Methode dealAtWorker wird auf dem Place des Diebes ausgeführt und teilt dem Worker mit, den Loot in seinen Taskpool aufzunehmen. Anschließend wird das CountDownLatch reduziert.

```
1  // On thief side, in Handler
2  public void dealAtWorker(
3      int thiefPlace, int workerId, B loot) {
4    asyncAt(place(thiefPlace), () -> {
5      GLBWorker worker = workers[workerId];
6      worker.addToBagAndWakeUp(loot);
7      worker.countDownWaitLatch();
8    });
9  }
```

Die Methode addToBagAndWakeUp muss richtig synchronisiert werden, um Verlust der Arbeitsschritte zu verhindern. Offensichtlich müssen die Arbeitsschritte sicher in den Taskpool abgelegt werden. Allerdings muss auch sichergestellt werden, dass wir den Worker aufwecken, wenn er schläft. Dabei kann es passieren, dass er sich schlafen legt direkt nachdem er Arbeitsschritte erhalten hat. Um das zu verhinden, nutzen wir wieder einen synchronized Block. Die Variable active ist vom Typ AtomicBoolean und repräsentiert, ob der Worker aktiv ist oder schläft.

```
1  // On victim side, in Worker
2  void addToBagAndWakeUp(B b) {
3    synchronized (bag) {
4      bag.merge(b);
5    }
6    boolean hasToProcess = false;
7    synchronized (shuttingDown) {
8      if(active.compareAndSet(false, true)) {
9        myWorkerHandler.workerCount++;
10       hasToProcess = true;
11     }
12   }
13   if(hasToProcess) process();
14 }
```

4.4 Übersichtlichkeit des Codes

Ein Nebenziel von MOSGLB ist es, die Implementierung gegenüber KobeGLB übersichtlicher zu gestalten. Das wird vor allem durch die Trennung der Handler und Worker in zwei separate Klassen erreicht.

Parallel zu der Trennung der Objekte werden einige weitere Aufgaben durch diese Trennung aufgeteilt. So ist der Worker verantwortlich dafür, die Synchronisierung der Variablen zu sichern und die ihm zugeteilten Arbeitsschritte zu erledigen. Der Handler hingegen übernimmt die Aufgabe der Kommunikation zwischen den Workern, sowohl innerhalb eines Places, als auch zwischen verschieden Places.

Die Worker müssen in MOSGLB innerhalb der process-Methode weniger Aufgaben behandeln. Das führt in Kombination mit der Entfernung des Lifelinethreads zu einer Reduzierung der Zeilen des Codes.

Mit den Änderungen ist es für einen Programmierer leichter den Code zu warten oder gegebenenfalls Anpassungen vorzunehmen.

5 Vergleich von KobeGLB und MOSGLB

5.1 Unbalanced Tree Search Benchmark

Der Benchmark, der zum Vergleich der Laufzeiten genutzt wurde, ist der Unbalanced Tree Search Benchmark (UTS). Bei UTS wird eine Tiefensuche auf einem pseudo-zufällig generierten Baum durchgeführt.

Die Parameter, die uns dabei zur Verfügung stehen, sind die maximale Höhe des Baums, ein initialer Seed und zusätzliche Parameter, die die Struktur des Baums definieren. Innerhalb einer Testreihe, beziehungsweise innerhalb eines Vergleichs, werden die Parameter gleich bleiben und es wird nur die Baumhöhe angeben.

Da die Bäume, die generiert werden, unbalanciert sind, können wir mit ihnen gut die Lastenbalancierung testen. Bei der Verteilung der Arbeit bekommt jeder Knoten einen Teil der Aufgabe. Die Knoten brauchen zudem jeweils unterschiedlich lange, bis sie ihren Anteil abgearbeitet haben.

5.2 Testumgebungen

Im Folgenden wurden Auswertungen mit FB16 gekennzeichnet sind, wenn sie auf der Partition ”FB16” des Clusters der Universität Kassel ausgeführt wurden. Dieser bestand zum Zeitpunkt der Messungen aus 12 Doppelprozessorsystemen mit je 2 Intel Xeon 6-Kern Prozessoren und Infiniband-Vernetzung. Die Inifiniband-Vernetzung ist eine Punkt-zu-Punkt-Verbindung zwischen den einzelnen Knoten, die deutlich schneller ist als beispielsweise Ethernet.

Die Auswertungen, die mit Goethe gekennzeichnet sind, wurden auf der Partition ”general1” des Goethe Clusters der Universität Frankfurt ausgeführt. Dieser bestand zum Zeitpunkt der Messungen aus mehr als 100 Doppelprozessorsystemen mit je 2 Intel Xeon Gold 6148 (Skylake), wodurch pro Knoten des Clusters 40 Cores zur Verfügung stehen. Auf dem Cluster wurde ebenfalls die Infiniband-Vernetzung genutzt.

5.3 Testläufe

Eine Messung besteht aus k einzelnen Messpunkten $X : x_0, \ldots, x_{k-1}$, da einzelne Tests Abweichungen aufweisen können. Ein Messpunkt beschreibt dabei die Laufzeit des Programmabschnitts, der ein gegebenes Problem berechnet. Das heißt die Inititalisierung von APGAS wird nicht berücksichtigt. Auf dem Cluster FB16 wurden je Messungen $k = 10$ Messpunkte erhoben. Für den Goethe-Cluster sind es $k = 5$ Messpunkte je Messung.

Von jeder Messung wurde der Mittelwert $a = \mathbb{E}(X) = \frac{1}{k} \sum_{i=0}^{k-1} x_i$ der Laufzeiten bestimmt und anschließend die Konsistenz der Ergebnisse durch den Variationskoeffizienten

$$d = \frac{\sqrt{\mathbb{V}(X)}}{\mathbb{E}(X)} = \frac{\sqrt{\frac{1}{k}\left(\sum_{i=0}^{k-1} x_i^2 - \frac{1}{k}\left(\sum_{i=0}^{k-1} x_i\right)^2\right)}}{a}$$

berechnet. Für eine Messung ist ein geringer Mittelwert und ein geringer Variationskoeffizient besser als ein hoher.

Der Vorteil des Variationskoeffizienten ist, dass wir gut die verschieden parametrisierten Messungen vergleichen können. Denn er vergleicht die relative Abweichung vom Mittelwert und ist somit nicht von der Größenordnung der Messpunkte abhängig. Außerdem ist er gegenüber Ausreißern weniger empfindlich.

Die Effizienz von MOSGLB und KobeGLB können wir vergleichen, indem wir den Mittelwert der Laufzeit für n Places mit dem Mittelwert für 1 Place vergleichen. Ist a_n der Mittelwert der Messungen für n Places, dann ist $\frac{a_1}{na_n}$ die Effizienz für n Places. Je größer die Effizienz, desto besser. Ist die Effizienz 1, dann ist die Laufzeit für 2 Places genau doppelt so schnell wie die für 1 Place.

Generell werden in Graphen auf der X-Achse die Anzahl der Places und auf der Y-Achse die Laufzeit in Sekunden dargestellt.

5.3.1 Vergleich mit einem Worker pro Place

Für den ersten Test betrachten wir den UTS Benchmark mit maximaler Baumhöhe 16. Pro Place soll maximal ein Worker gestartet werden dürfen. Dadurch erhalten

wir Ergebnisse ohne work-sharing, beziehungsweise ohne die Vorteile der Lokalität.

Zunächst vergleichen wir die Laufzeiten. In Abbildung 2 werden die Mittelwerte der Messungen aufgeführt. MOSGLB benötigt in unseren Testläufen mehr Zeit als KobeGLB. Allerdings brauchen beide mit 2 Places in etwa gleichlang und mit mehr Places wird MOSGLB schneller als KobeGLB.

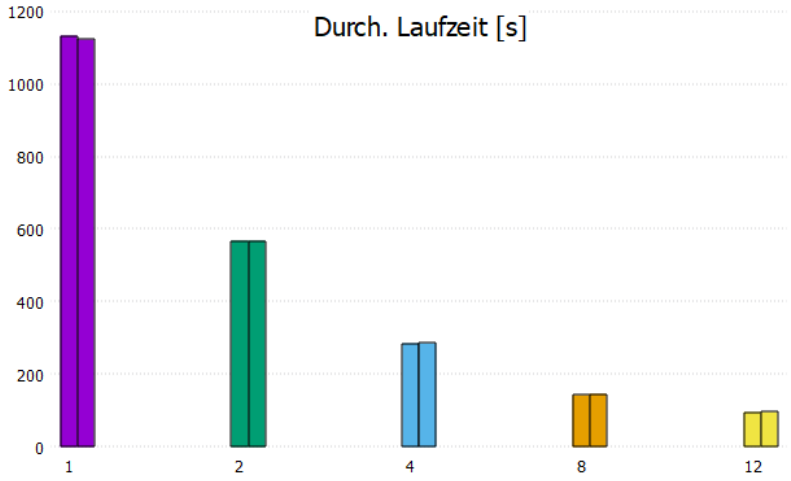


Abbildung 2: Durchschnittliche Laufzeit ohne Initialisierung in Sekunden. Reihenfolge: MOSGLB, KobeGLB.

Die expliziten Mittelwerte sind in Tabelle 1 gelistet.

Places	1	2	4	8	12
MOSGLB	1131,99	567,50	285,05	143,67	96,30
KobeGLB	1123,72	566,76	286,65	145,25	97,70

Tabelle 1: Mittelwerte in Sekunden - 1 Worker pro Place

Da in dieser Auswertung die Gesamtzahl an Worken im Netzwerk nicht über 12 hinausgeht, sind die Abweichung der einzelnen Messungen hier sehr gering. Die Konsistenz der Testläufe liegt zwischen 0.31% und 0.82%. Außerdem sind MOSGLB und KobeGLB sehr ähnlich. Daher wird dieser Fall hier nicht weiter ausgeführt.

5.3.2 Vergleich mit mehreren Workern pro Place

Nun wollen wir KobeGLB und MOSGLB mit vielen Workern vergleichen. Wir nutzen erneut den UTS Benchmark, um die Laufzeiten der beiden zu vergleichen.

Für diesen Vergleich haben wir zwei Auswertungen durchgeführt. Die erste wurde auf dem Cluster FB16 ausgeführt, wobei der UTS Benchmark mit Tiefe 18 berechnet wurde. Pro Place wurden 12 Worker gestartet und es wurde ein Testlauf für bis zu 12 Places erstellt. Die zweite Auswertung wurde auf dem Goethe Cluster erstellt. Dabei wurde der UTS Benchmark mit Tiefe 19 berechnet. Die Testläufe wurden mit bis zu 32 Places und 40 Workern pro Place gestartet.

Zunächst betrachten wir die Auswertung auf FB16 genauer. Für KobeGLB und MOSGLB wurde jeweils ein Testlauf für 1,2,4,8 und 12 Places gestartet. In Abbildung 3 sehen wir die Mittelwerte der Testläufe. Für jeden Place wurden zwei Balken erstellt. Jeweils der linke Balken zeigt den Mittelwert der Messungen für MOSGLB und der rechte für KobeGLB.

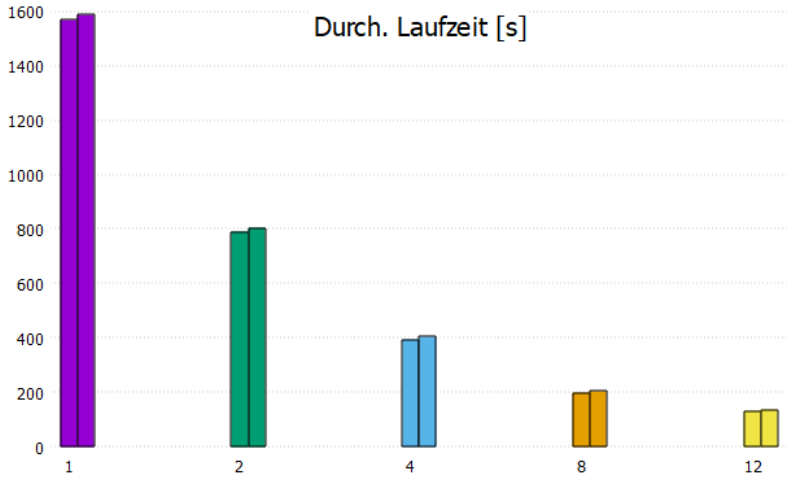


Abbildung 3: Mittelwerte in Sekunden FB16 - Tiefe 18. Reihenfolge: MOSGLB, KobeGLB.

Die Mittelwerte sind in Tabelle 2 gelistet.

Places	1	2	4	8	12
MOSGLB	1567,57	789,6	393,36	198,51	133,60
KobeGLB	1587,99	804,00	405,19	205,70	137,15

Tabelle 2: Mittelwerte in Sekunden - 12 Worker pro Place

In Tabelle 3 ist das Verhältnis von MOSGLB und KobeGLB dargestellt. Ist a_M der Mittelwert von MOSGLB und a_K der Mittelwert von KobeGLB für den Testlauf mit n Places, so zeigt die Zeile mit n Places in der ersten Spalte $\frac{a_M}{a_K}$ und in der zweiten Zeile $\frac{a_K}{a_M}$. Wir sehen, dass MOSGLB in unseren Testlaufen immer schneller ist als KobeGLB. Für mehr Places stellen wir zusätzlich fest, dass MOSGLB seinen Vorteil bestärkt.

Places	**MOSGLB/KobeGLB**	**KobeGLB/MOSGLB**
1	98,71%	101,3%
2	98,2%	101,82%
4	97,08%	103,01%
8	96,51%	103,62%
12	97,42%	102,65%

Tabelle 3: Verhältnis der Laufzeiten

Nun vergleichen wir die Effizenz $\frac{a_1}{na_n}$ für diese Auswertung in Tabelle 4. In unserer Auswertung ist MOSGLB effizienter als KobeGLB für jeden Testlauf.

Places	**MOSGLB**	**KobeGLB**
1	1	1
2	0,993	0,988
4	0,996	0,98
8	0,987	0,965
12	0,978	0,965

Tabelle 4: Effizienz - 12 Worker pro Place

Um diese Auswertung abzuschließen sind in Tabelle 5 die Abweichungen der Testläufe aufgeführt. Wir sehen in der Tabelle gemischte Ergebnisse. Allerdings lässt sich anhand der geringeren Werte feststellen, dass MOSGLB in der Regel effizienter ist.

Places	MOSGLB	KobeGLB
1	0.45%	0.97%
2	1.66%	1.56%
4	0.4%	1.74%
8	0.53%	2.82%
12	1.14%	1.31%

Tabelle 5: Abweichungen - 12 Worker Pro Place

Die zweite Auswertung wurde auf dem Goethe Cluster ausgeführt. Der Vorteil des Goethe Clusters ist, dass wir unsere Tests mit bis zu 40 Workern pro Place und vielen Places durchführen konnten. Allerdings schwanken die Ergebnisse in seltenen Fällen mit bis zu 150% höherer Laufzeit gegenüber der minimalen Laufzeit. Für einen Place wurden zusätzliche Messungen gestartet, um die Ausreißer zu entfernen.

Die Abweichungen sehen wir in Tabelle 6. Sie sind sowohl bei MOSGLB, als auch bei KobeGLB zu sehen. Allerdings nehmen die Abweichungen in der Regel für mehr Places ab.

Places	MOSGLB	KobeGLB
1	0.79%	2.07%
2	1.73%	13.65%
4	4.87%	8.06%
8	3.31%	8.04%
16	1.23%	4.14%
32	1.39%	3.60%

Tabelle 6: Abweichungen - 40 Worker Pro Place

Da die Ausreißer nur die Laufzeit verlängern, betrachten wir von nun an die Mittelwerte und das Minimum. Für die Laufzeiten ergeben sich dann die Werte, die in Tabelle 7 abgelesen werden können.

In Abbildung 4 sehen wir die durchschnittlichen und die minimalen Laufzeiten als Graph dargestellt. Dabei ist für jeden Place ein Balkenpaar gezeichnet. Der linke Balken repräsentiert die Laufzeit für MOSGLB und der rechte für KobeGLB.

Die Verhältnisse der Laufzeiten sind in Tabelle 8 dargestellt. Dabei steht M für MOSGLB und K für KobeGLB.

	Places	1	2	4	8	16	32
MW	**MOSGLB**	2263,27	1138,17	593,33	292,31	145,75	77,18
MW	**KobeGLB**	2343,35	1336,08	669,08	343,88	179,74	105,61
Min	**MOSGLB**	2263,27	1118,31	565,15	284,26	143,1	75,56
Min	**KobeGLB**	2313,35	1133,54	594,28	318,43	173,24	101,54

Tabelle 7: Laufzeiten in Sekunden - 40 Worker pro Place

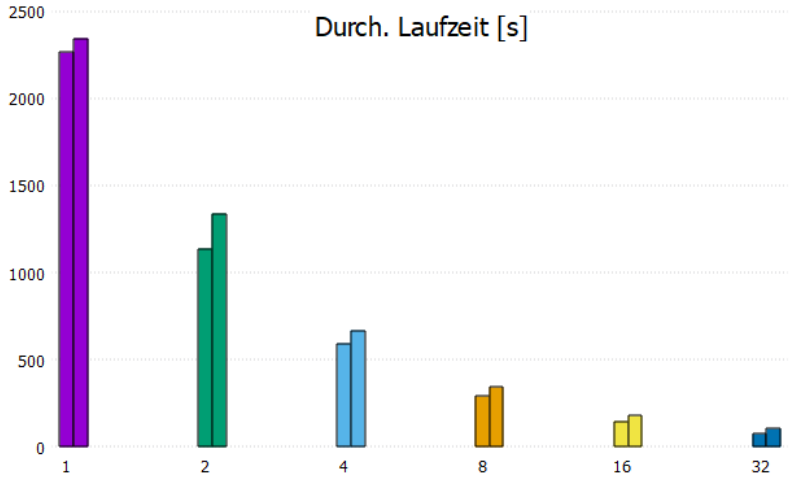
	Mittelwert		Minimum	
Places	**M/K**	**K/M**	**M/K**	**K/M**
1	96,58%	103,54%	96,67%	103,44%
2	85,19%	117,39%	98,66%	101,36%
4	88,68%	112,77%	95,10%	105,15%
8	85,01%	117,64%	89,27%	112,02%
16	81,09%	123,33%	82,60%	121,06%
32	73,08%	136,83%	74,41%	134,38%

Tabelle 8: Verhältnis der Laufzeiten - 40 Worker pro Place

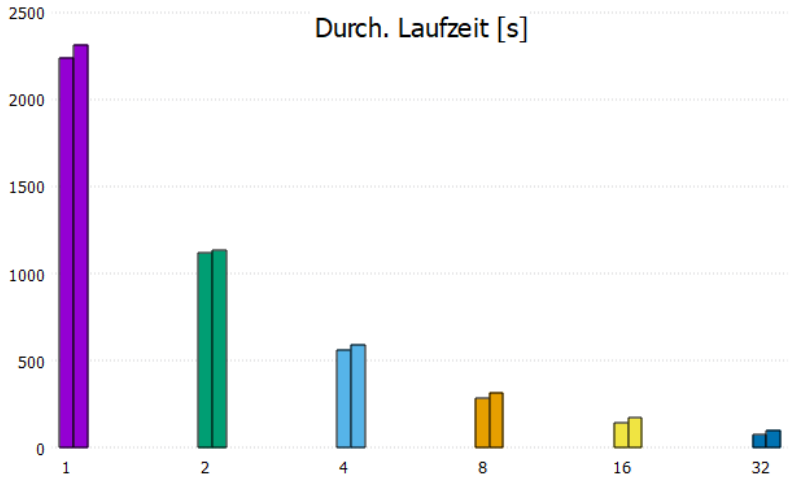
Abschließend zu dieser Auswertung ist die Effizienz in Tabelle 9 eingetragen, wobei wir auch hier die Effizenz bezüglich des Mittelwerts bestimmt haben. Jedoch haben wir zusätzlich die Effizienz für das Minimum bestimmt. Da keine Ausreißer nach unten existieren, erhalten wir so eine besser Übersicht über die Laufzeiten ohne die Ausreißer nach oben.

	Mittelwert		Minimum	
Places	**MOSGLB**	**KobeGLB**	**MOSGLB**	**KobeGLB**
1	1	1	1	1
2	0,994	0,877	0,996	1,03
4	0,954	0,876	0,989	0,973
8	0,968	0,852	0,983	0,908
16	0,971	0,815	0,977	0,835
32	0,916	0,693	0,925	0,712

Tabelle 9: Effizienz - 40 Worker pro Place



(a) Durchschnittliche Laufzeit



(b) Minimale Laufzeit

Abbildung 4: Laufzeiten - 40 Worker pro Place. Reihenfolge: MOSGLB, KobeGLB.

5.3.3 Effekt der lokalen Optimierung

Abschließend wollen wir untersuchen, welche Auswirkungen das Ausnutzen der Lokalität hat. Dazu vergleichen wir den Testlauf aus Abschnitt 5.3.2 von MOSGLB

mit einem neuen Testlauf. Bei diesem wurden bei MOSGLB die Stellen aus dem Code entfernt, die eine Anfrage innerhalb eines Places weiterleiten. Das heißt bei zufälligen Stehlversuchen wird in jedem Fall sofort eine Antwort an den Dieb gesendet und vor den zufälligen Stehlanfragen wird nicht auf dem lokalen Place geschaut, ob ein Worker Arbeitsschritte abgeben kann.

Da die Messungen auf dem Goethe Cluster durchgeführt wurden, gab es hier ebenfalls gelegentliche Ausreißer. Wir kennzeichnen mit M-LO die Variante von MOSGLB mit den Optimierungen und mit M-NO die Variante ohne die zuvor genannten Optimierungen.

Zunächst schauen wir auf die Laufzeiten in Tabelle 10 und die Abweichungen in Tabelle 11.

	Places	1	2	4	8	16	32
MW	**M-NO**	2270,72	1136	589,01	288,44	147,57	82,04
MW	**M-LO**	2263,27	1138,17	593,33	292,31	145,75	77,18
Min	**M-NO**	2218,01	1122,93	565,05	283,26	145,28	80,12
Min	**M-LO**	2236,33	1118,31	565,15	284,26	143,1	75,56

Tabelle 10: Laufzeiten in Sekunden - 40 Worker pro Place

Places	**M-NO**	**M-LO**
1	1,76%	0.79%
2	0,66%	1.73%
4	7,20%	4.87%
8	3,20%	3.31%
16	0,97%	1.23%
32	3,01%	1.39%

Tabelle 11: Abweichungen - 40 Worker Pro Place

Dabei stellen wir fest, dass wir in der Regel nur wenige Ausreißer haben und die Laufzeiten für 1-8 Places sehr ähnlich sind. Erst ab 32 Places stellen wir einen großen Unterschied bei der Laufzeit fest. Das spiegelt sich vor allem bei der Effizienz in Tabelle 12 und in den Verhältnissen der Laufzeiten in Tabelle 13 wieder.

	Mittelwert		Minimum	
Places	**M-NO**	**M-LO**	**M-NO**	**M-LO**
1	1	1	1	1
2	0,999	0,994	0,988	0,996
4	0,964	0,954	0,981	0,989
8	0,984	0,968	0,979	0,983
16	0,962	0,971	0,945	0,977
32	0,865	0,916	0,865	0,925

Tabelle 12: Effizienz - 40 Worker pro Place

	Mittelwert		Minimum	
Places	**M-NO/M-LO**	**M-LO/M-NO**	**M-NO/M-LO**	**M-LO/M-NO**
1	100,33%	99,67%	99,18%	100,83%
2	99,81%	100,19%	100,41%	99,59%
4	99,27%	100,73%	99,98%	100,02%
8	98,68%	101,34%	99,65%	100,35%
16	101,25%	98,76%	101,52%	98,50%
32	106,30%	94,07%	106,03%	94,31%

Tabelle 13: Verhältnis der Laufzeiten - 40 Worker pro Place

5.4 Auswertung der Ergebnisse

Wir haben in Abschnitt 5.3.1 gesehen, dass unsere Tests ähnlich lange Laufzeiten haben. Da wir in den Tests nur einen Worker pro Place erlaubt haben, konnte von KobeGLB und MOSGLB keine Lokalität ausgenutzt werden.

In Tabelle 1 hatte KobeGLB nahezu gleiche Werte wie MOSGLB und in Tabelle 2 war MOSGLB immer etwas schneller. Daraus können wir schließen, dass MOSGLB im Vergleich zu KobeGLB stärker von mehr Workern profitiert.

Zusätzlich konnten wir feststellen, dass MOSGLB besser mit mehr Workern im gesamten System skaliert. Das konnten wir vor allem in Tabelle 4 sehen. Ein möglicher Grund dafür ist, dass die Worker durch das lokale work-stealing von anderen Places besser profitieren können als es bei KobeGLB der Fall ist.

Da mehr Places im System sind, müssen öfter zufällige Stehlanfragen gestellt werden. Diese Anfragen können in MOSGLB häufiger erfüllt werden.

Für den Cluster FB16 konnten wir in Tabelle 5 feststellen, dass MOSGLB und KobeGLB ihre Laufzeiten konsistent halten konnten. Da beide Bibliotheken sehr ähnlich vorgehen, ist es nicht verwunderlich, dass sie ähnlich geringe Abweichungen aufweisen.

Die Auswertung auf dem Goethe Cluster lieferte gemischte Ergebnisse, da Ausreißer in den Messwerten existieren. Gründe für die Ausreißer können beispielsweise Hyperthreading, Probleme mit der Kühlung der Hardware oder auch verschiedene Hardware je Knoten sein.

Dennoch sehen wir als Trend, dass MOSGLB vor allem in der Effizienz für große Cluster eine deutliche Steigerung gegenüber KobeGLB darstellt.

Wir konnten in Abschnitt 5.3.3 sehen, dass es nicht ganz offensichtlich ist, ob die Lokalitätsoptimierung eine Verbesserung darstellt. Für wenige Places, konnten wir feststellen, dass beide Varianten sehr ähnliche Laufzeiten haben. Die Schwankungen sind dabei durch die Ausreißer in den Messungen entstanden. Allerdings konnten wir bei 32 Places feststellen, dass die Lokalitätsoptimierung die Laufzeit um ca. 6% verkürzt hat. Insbesondere ist die maximale Laufzeit (78,5s) mit Optimierung kürzer als die minimale Laufzeit (80,12s) ohne.

Ein Grund dafür ist, dass bei wenigen Places die Wahrscheinlichkeit hoch ist, vom eigenem Place zu stehlen. Bei aktiver Lokalitätsoptimierung wird vor den zufälligen Stehlanfragen jedoch schon geprüft, ob Arbeitsschritte vorhanden sind. Ist dies nicht der Fall kann trotzdem ein zufälliger Stehlversuch an den Place gesendet werden. Bei diesem wird dann erneut der lokale Place abgesucht, wenn er fehlschlägt. Das führt vor allem dazu, dass zwei mal der beste Worker ermittelt wird.

6 Verwandte Arbeiten

Neben J_GLB und KobeGLB gibt es noch weitere Global Load Balancer Modelle. Auch im Fachgebiet wird derzeit an verschieden Modellen, die bisher nicht genannt wurden, geforscht.

Dazu gehört beispielsweise der Fault Tolerant GLB [5], welcher auch einen Global Load Balancer mit APGAS implementiert. Das Ziel ist es Fehlertoleranz auf Anwendungsebene zu realisieren, um beispielsweise mit ausfallenden Knoten umgehen zu können. Dabei können mit einem kleinem Overhead Fehler erkannt und teilweise sogar toleriert werden. Das heißt wenn ein Fehler auftritt, kann das Programm trotzdem das richtige Ergebnis berechnen.

Die wesentliche Idee von Fault Tolerant GLB ist es im Algorithmus regelmäßig Backups der Task Pools eines Places zu erstellen. Diese Backups werden zusätzlich beim work-stealing erstellt.

7 Fazit

In dieser Arbeit wurde MOSGLB, welcher ein neuer Global Load Balancer in Java ist, vorgestellt. Zuvor haben wir uns in Abschnitt 2 die APGAS-Bibliothek, welche wir genutzt haben, um unsere Programme auf einem Cluster verteilt parallel ausführen zu können. Außerdem haben wir bereits entwickelte GLB-Frameworks genauer untersucht. Dazu gehörten zum einen J_GLB, welches in X10 implementiert ist und zum anderen KobeGLB, welches pro Place mehrere Worker implementiert.

Wir haben anschließend in Abschnitt 3 die Konzepte von MOSGLB gezeigt und sie mit denen von KobeGLB verglichen. Dabei haben wir festgestellt, dass sie sich vor allem in der Lastenbalancierung unterscheiden. Denn MOSGLB ersetzt work-sharing durch einen zusätzlichen Schritt im work-stealing. Außerdem nutzt MOSGLB zusätzlich die Lokalität beim work-stealing aus.

In Abschnitt 4 haben wir einige Implementierungsdetails genauer untersucht und haben am Beispiel des zufälligen work-stealing den Ablauf der Kommunikation genauer betrachtet.

Beide Varianten haben wir in Abschnitt 5 verglichen. Wir haben mittels des UTS Benchmarks einige Szenarien getestet. Dabei konnten wir feststellen, dass MOSGLB gegenüber KobeGLB häufig schneller die Aufgabe abarbeiten konnte. Außerdem konnten wir sehen, dass MOSGLB mit vielen Worken und vielen Places in der Regel besser skaliert als KobeGLB. Vor allem für viele Places und Worker konnten wir in Abschnitt 5.3.2 sehen, dass MOSGLB signifikante Verbesserungen bringt.

Als wir untersucht haben, ob das Ausnutzen der Lokalität eine Verbesserung darstellt, haben wir festgestellt, dass dies in unserem Fall erst für 32 Places der Fall war.

Abschließend können wir festhalten, dass MOSGLB die gestellten Aufgaben effizienter auf dem Cluster berechnen konnte, als es bei KobeGLB der Fall war.

Literatur

[1] Finnerty, P., Kamada, T., and Ohta, C. (2020). Self-adjusting task granularity for global load balancer library on clusters of many-core processors. In *Proceedings of the Eleventh International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '20, New York, NY, USA. Association for Computing Machinery.

[2] Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., and Tseng, C.-W. (2006). Uts: An unbalanced tree search benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, page 235–250, Berlin, Heidelberg. Springer-Verlag.

[Oracle] Oracle. Java Platform, Standard Edition & Java Development Kit Version 17 API Specification. `https://docs.oracle.com/en/java/javase/17/docs/api/index.html`. Accessed: 2022-02 & 2022-03.

[4] Posner, J. and Fohry, C. (2016). Cooperation vs. coordination for lifeline-based global load balancing in apgas. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, X10 2016, page 13–17, New York, NY, USA. Association for Computing Machinery.

[5] Posner, J. and Fohry, C. (2018). A java task pool framework providing fault-tolerant global load balancing. *International Journal of Networking and Computing*, 8:2–31.

[6] Zhang, W., Tardieu, O., Grove, D., Herta, B., Kamada, T., Saraswat, V. A., and Takeuchi, M. (2013). GLB: lifeline-based global load balancing library in X10. *CoRR*, abs/1312.5691.

Eigenständigkeitserklärung

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

_____

Kai Hardenbicker