UNIKASSEL
VERSITÄT

Fachbereich 16 Elektrotechnik/Informatik
Studiengang Master Informatik

Dynamisches Hinzufügen und Entfernen von Places innerhalb der Global Load-Balancing Runtime von APGAS

Masterarbeit

Jonas Scherbaum
Matrikelnummer: 30204135

Betreuer: Frau Prof. Dr. Fohry
Erstprüfer: Frau Prof. Dr. Fohry
Zweitprüfer: Herr Prof. Dr. Zündorf

Kassel, den 01.06.2020

Selbständigkeitserklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kassel, den 01.06.2020

Jonas Scherbaum

Inhaltsverzeichnis

1 Einleitung

Viele der heutigen Problemstellungen der Softwareentwicklung sind mittlerweile so umfangreich, dass sie sich nur noch lösen lassen, indem sie von einer Vielzahl von Computern und Prozessoren gleichzeitig verarbeitet werden. Der Bereich des **HPC (High Performance Computing)**, zu deutsch Hochleistungsrechnen, beschäftigt sich mit der Berechnung dieser umfangreichen Problemstellungen. Die dafür benötigten **Hochleistungsrechner** können hochparallele Supercomputer sein, oder Computercluster. Supercomputer sind spezielle für das Hochleistungsrechnen entwickelte Computer, die oft aus eine sehr hohen Anzahl von Prozessoren bestehen, die auf einen teilweise gemeinsamen Speicher und gemeinsame Peripherie zugreifen. Ein Computercluster oder einfach **Cluster**, bezeichnet eine beliebige Anzahl von vernetzten Computern. Die einzelnen in einem Cluster vernetzten Computer werden auch als **Knoten** bezeichnet. Zur Verwaltung der im HPC Bereich auszuführenden Programme werden häufig **Workload-Mananger** eingesetzt. Dem Workload-Manager werden die Porgramme mitgeteilt, die auf dem Cluster ausgeführt werden sollen. Die auszuführenden Programme und die mit ihnen verbundenen Hardwareanforderungen werden als **Jobs** bezeichnet und vom Workload-Manager mit sogenannten **Scheduling-Verfahren** auf die Knoten des Clusters verteilt. Der Workload-Manager reiht alle eingehenden Jobs in eine Warteschlange ein. Jobs können nicht nur eine unterschiedliche Anzahl an Knoten benötigen, sondern darüber hinaus auch unterschiedlich viele Prozessoren pro Knoten oder unterschiedlich viel RAM-Speicher. Die Knoten eines Clusters werden als **Ressourcen** abstrahiert. Ein Scheduling-Verfahren verteilt die Jobs in der Warteschlange auf verfügbare Ressourcen des Clusters. In welcher Reihenfolge das Scheduling-Verfahren die Jobs auf die verfügbaren Ressourcen verteilt, kann unterschiedlich sein. Die Jobs können zum Beispiel anhand von Prioritäten verteilt werden, die sich in Abhängigkeit der Zeit, die sich die Jobs bereits in der Warteschlange befinden, erhöhen. Sobald ein Job beendet wurde, werden die von

ihm verwendeten Ressourcen wieder freigegeben, sodass sie dem Workload-Manager wieder zur Verfügung stehen.

Feitelson und Rudolph [1] klassifizieren Jobs in vier Kategorien: **rigid**, **evolving**, **moldable** und **malleable**. Ein rigid Job benötigt eine ganz bestimmte Anzahl an Ressourcen, um ausgeführt zu werden. Diese Ressourcen müssen beim Einreihen des Jobs mit angegeben werden. Ein evolving Job kann die benötigten Resourcen während seiner Ausführung verändern, um sich selbst anzupassen. Der Workload-Manager des Clusters muss dem Job diese Ressourcen dann zur Verfügung stellen, damit der Job erfolgreich beendet werden kann. Ein moldable Job ist in der Lage sich selbst an vorgegebene Ressourcen anzupassen. Zum Beispiel an die zur Verfügung stehende Menge an Prozessoren oder Knoten. Dies ist allerdings nur zum Zeitpunkt des Starts möglich. Im Gegensatz zum moldable Job ist ein malleable Job in der Lage, sich während seiner Ausführung an sich ändernde Ressourcen anzupassen. Was ihn zum flexibelsten der Job-Typen macht.

Innerhalb eines Clusters werden üblicherweise viele Jobs ausgeführt. Häufig werden einige Jobs auch parallel zueinander ausgeführt, wenn es die Ressourcen des Clusters zulassen. Die Auslastung eines Clusters zu erhöhen, also zu jedem Zeitpunkt so viele Ressourcen zur Berechnung von Jobs zu nutzen wie möglich, ist eine der aktuellen Herausforderungen im Bereich des HPC, siehe [1] [2] [3]. Malleable Jobs sind dabei eine Möglichkeit die Ressourcen eines Clusters möglichst vollständig zu nutzen. Entweder indem einigen der Jobs während ihrer Ausführung Ressourcen zugunsten anderer Jobs wieder entzogen werden, oder indem freie Ressourcen einem bereits in der Ausführung befindlichen Job zur Verfügung gestellt werden.

In Abbildung 1.1 ist illustriert, wie das Hinzufügen und Entfernen von Ressourcen während der Ausführung eines malleable Jobs die Nutzung der Ressourcen im Cluster optimieren könnte. Es existieren drei Jobs in der Warteschlange mit folgender Reihenfolge: A, B, C. Job A ist ein malleable Job und benötigt aus diesem Grund keine feste Anzahl an Ressourcen. Job B benötigt hingegen genau 2 Ressourcen und Job C benötigt exakt 3 Ressourcen. Es stehen 6 Ressourcen zur Verfügung, um die Jobs auszuführen. Dem Job A werden 4 Ressourcen zur Verfügung gestellt, damit Job B 2 der 6 Ressourcen bekommen kann und beide parallel ausgeführt werden können. Sobald Job B beendet wurde, wird dem Job A eine Ressource entzogen, um Job C auszuführen. Nachdem

auch Job C beendet wurde, können alle freien Ressourcen für Job A verwendet werden. Alternativ wäre auch das gleichzeitige Ausführen der Jobs A mit 1 Knoten, Job B mit 2 Knoten und Job C mit 3 Knoten eine mögliche Ausführungsreihenfolge gewesen, um die Nutzung der Ressourcen im Cluster zu optimieren.

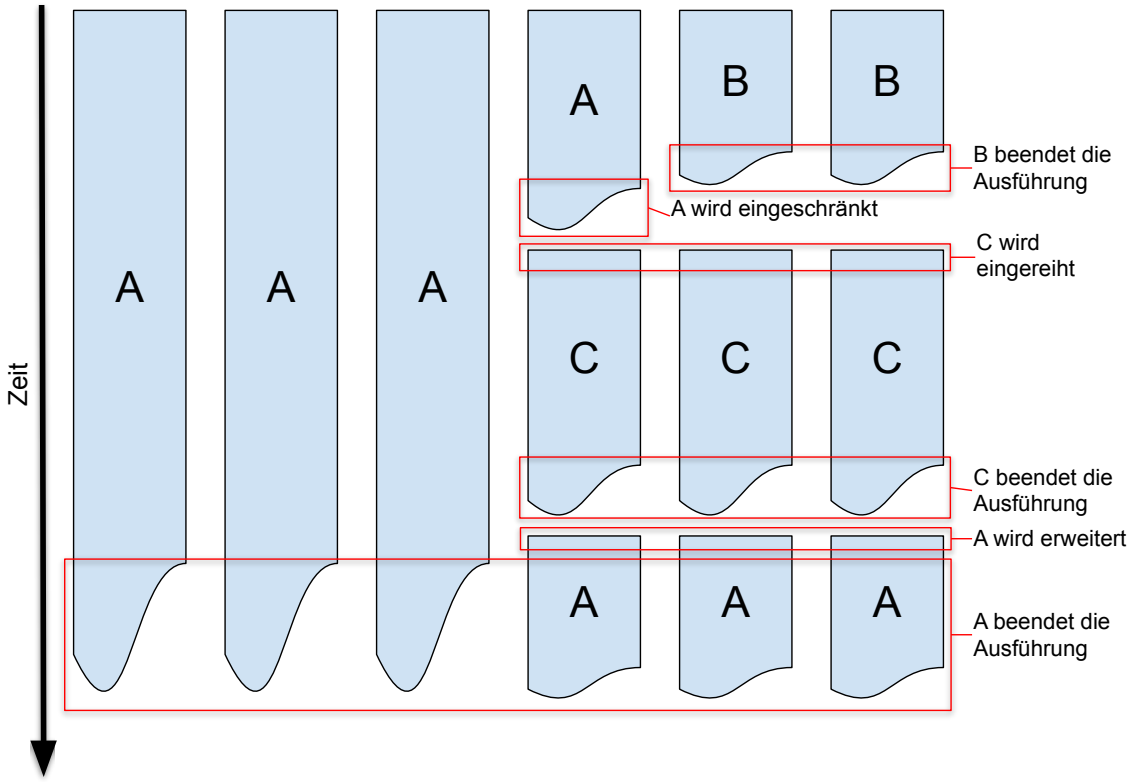

Abbildung 1.1: Optimierung der Nutzung von Ressourcen im Cluster

Einige Ansätze Programme kompatibel mit malleable Jobs zu machen, basieren darauf eine zusätzliche Abstraktionsschicht zwischen dem Programm und den verwendeten Ressourcen einzuführen, siehe [4] [5] [6] [7]. Diese Abstraktionsschicht bildet die physikalischen Ressourcen auf virtuelle Ressourcen ab. Das Programm wird dann auf den virtuellen Ressourcen ausgeführt. Während der Ausführung des Programms kann dann die Abbildung der physikalischen Ressourcen auf die virtuellen Ressourcen geändert werden. Auf diese Weise wird es ermöglicht einem Job während seiner Ausführung Ressourcen hinzuzufügen oder zu entfernen. Eine andere Variante malleable Jobs zur

Verfügung zu stellen ist es, dem Programm selbst zu ermöglichen sich an Ressourcenveränderungen während der Ausführung anzupassen, siehe [8] [9]. Dabei wird dem Programm während seiner Ausführung mitgeteilt, dass sich seine Ressourcen zukünftig ändern werden. Das Programm muss dann darauf reagieren, indem es seine Berechnungen und Daten entsprechend der geänderten Ressourcen umverteilt. Diese Eigenschaft des Programms, sich an ändernde Ressourcen anzupassen, wird im weiteren Verlauf dieser Arbeit als **Ressourcen-Elastizität** bezeichnet. Die genauere Bezeichnung der Ressourcen-Elastizität dient der Abgrenzung zum Begriff malleable Job, da die Ressourcen-Elastizität sich nur auf die eigenständige Anpassung des Programms bezieht.

Diese Arbeit präsentiert ein Konzept für eine Programmierumgebung, die Ressourcen-Elastizität ermöglicht. Die größten Herausforderungen waren das effiziente Verteilen der Berechnungen auf die sich ändernden Ressourcen, als auch sicherzustellen, dass alle für die Berechnungen notwendigen Daten bei Anpassungen an sich ändernde Ressourcen erhalten bleiben. Im Hinblick auf diese beiden Herausforderungen wurde als Ausgangsbasis die Java-Bibliothek **APGAS** gewählt. Diese verwendet ein **Task**-basiertes Berechnungsmodell, dass die Berechnung des Programms in viele kleinere Teilberechnungen aufteilt, die Tasks. Für die Ressourcen-Elastizität bieten sich die Tasks sehr gut an, um Teile der Berechnung des Programms bei sich ändernden Ressourcen umzuverteilen. Werden neue Ressourcen während der Programmausführung hinzugefügt, können Tasks von bereits genutzten Ressourcen zu den neuen Ressourcen transferiert werden. Sollten Ressourcen während der Programmausführung entfernt werden, dann können deren Tasks auf die verbleibenden Ressourcen umverteilt werden.

Die APGAS-Bibliothek bietet einen weiteren Vorteil im Hinblick auf die Ressourcen-Elastizität durch dessen **globaler Lastenbalancierung** auf Basis eines **Lifelinegraphen**. Die globale Lastenbalancierung sorgt dafür, dass die Tasks während der Programmausführung so auf die verfügbaren Ressourcen verteilt werden, dass eine möglichst hohe Auslastung aller Ressourcen erreicht wird. Für die Verteilung der Tasks auf die Ressourcen wird ein **Workstealing** Algorithmus eingesetzt, der es den Ressourcen ermöglicht sich Tasks von anderen Ressourcen zu stehlen, wenn diese selbst keine Tasks mehr besitzen. Dieser Algorithmus wird durch einen Lifelinegraphen ergänzt, der jeder der

Ressourcen sogenannte **Lifelines** zuordnet. Diese Lifelines geben der zugeordneten Ressource vor, von welchen anderen Ressourcen diese Tasks stehlen kann. Der Lifelinegraph bestimmt die Lifelines dabei so, dass die Ressourcen gleichmäßig Tasks voneinander stehlen.

Aufbauend auf den Vorteilen der APGAS-Bibliothek wurde im Rahmen dieser Arbeit ein Konzept erarbeitet, das bei sich ändernden Ressourcen dafür sorgt, dass alle betroffenen Daten und Tasks automatisch umverteilt werden. Bei neu hinzukommenden Ressourcen werden alle notwendigen Daten zu den neuen Ressourcen transferiert und anschließend wird die globale Lastenbalancierung genutzt um diesen Ressourcen Tasks zu schicken. Werden Ressourcen während der Ausführung des Programms entfernt, werden zuvor alle Daten der Ressource gesichert und alle Tasks über die globale Lastenbalancierung umverteilt. Dabei werden die Eigenschaften des Workstealing ausgenutzt, um die Daten und Tasks gleichmäßig zu verteilen. Der Lifelinegraph wird ebenfalls bei jeder Änderung an den Ressourcen aktualisiert. Die Aktualisierung des Lifelinegraphen ist ein wesentlicher Bestandteil der Ressourcen-Elastizität, da die Lifelines des Lifelinegraphen dazu genutzt werden, die Tasks der Ressourcen gleichmäßig umzuverteilen, sobald sich diese ändern.

Die für das weitere Verständnis dieser Arbeit notwendigen Grundlagen werden im nachfolgenden Kapitel 2 erläutert und umfassen die APGAS-Bibliothek, die globale Lastenbalancierung sowie den Lifelinegraphen. Im darauf folgenden Kapitel 3 wird das erarbeitete Konzept der Ressourcen-Elastizität beschrieben. Dies umfasst das Hinzufügen und das Entfernen von Ressourcen. Kapitel 4 stellt die Implementierung des Konzeptes aus Kapitel 3 vor. Die zur Evaluation verwendeten Benchmarks und die Ergebnisse werden im Kapitel 5 vorgestellt und evaluiert. Das Kapitel 7 fasst abschließend diese Arbeit und ihre Ergebnisse kurz zusammen und gibt einen Ausblick auf zukünftige Möglichkeiten der Ressourcen-Elastizität.

2 Grundlagen

Die APGAS-Bibliothek, auf der das Konzept der Ressourcen-Elastizität aufbaut ist in Kapitel 2.1 beschrieben. Zu Beginn wird ihr Programmiermodell beschrieben, mit den grundlegenden Konstrukten **async**, **asyncAt**, **finish** und **GlobalRef**. Das folgende Kapitel 2.2 beschreibt im Detail die globale Lastenbalancierung der APGAS Bibliothek einschließlich der Konstrukte **asyncAny** und **finishAsyncAny**. Abschließend beschreibt Kapitel 2.3 den Lifelinegraphen, der für die Bestimmung der Lifelines sorgt.

2.1 Die APGAS-Bibliothek

Die Java-Bibliothek APGAS implementiert das **APGAS-Programmiermodell (Asynchronous Partitioned Global Address Space)**, das eine Erweiterung des **PGAS-Programmiermodells (Partitioned Global Address Space)** ist. Im PGAS-Programmiermodell wird der gesamte Speicher aller dem Programm zur Verfügung gestellten Ressourcen zu einem gemeinsamen Adressraum zusammengefasst. Dies ermöglicht es von jeder Ressource des Programms aus auf jede Speicherzelle einer jeden anderen Ressource direkt zuzugreifen. Der gemeinsame Speicher wird allerdings wieder partitioniert in sogenannte Places. Ein Place stellt eine abstrakte Einheit dar, bestehend aus dem Speicher und der Recheneinheit (üblicherweise einem Mehrkernprozessor) einer Ressource. Speicherzugriffe von einem Place auf einen anderen Place sind deutlich langsamer, als Speicherzugriffe innerhalb desselben Place, da dafür in der Regel ein Netzwerkzugriff notwendig ist.

Ein Place kann mehrere Threads enthalten, um Teilaufgaben separat zu verarbeiten. Dies macht sich die Erweiterung des PGAS-Modells, das APGAS-Modell zunutze. Das

APGAS-Modell erweitert das PGAS-Modell um asynchrone Tasks, die parallel zueinander auf einem Place oder verteilt auf mehreren Places ausgeführt werden können. Diese Threads werden als Worker oder Worker-Threads bezeichnet. Während der Ausführung eines Tasks durch einen Worker können wieder neue Tasks erstellt werden. Diese werden als transitive Tasks bezeichnet. Tasks können direkt nach deren Erstellung im APGAS-Modell von einem Place zu einem anderen transferiert werden, um diese auf einem anderen Place auszuführen. Das APGAS-Modell definiert für das Erstellen von Tasks das Konstrukt `async`, vergleiche [10] [11] [12] [13], und für die Synchronisierung von Tasks das `finish`-Konstrukt. Ein `finish`-Konstrukt definiert einen Block, in dem Tasks erstellt werden können. Der Block des `finish`-Konstrukts kann vom aufrufenden Thread erst dann wieder verlassen werden, wenn alle in ihm definierten Tasks abgearbeitet wurden, einschließlich aller transitiven Tasks.

Die APGAS-Bibliothek ist ein Ableger des X10 Projektes [10] [11] und portiert einige der Grundfunktionen von X10 nach Java. Places repräsentiert die APGAS-Bibliothek durch eine eigene Instanz einer Java Virtual Machine (JVM). Für die Inter-Place Kommunikation wird die Bibliothek Hazelcast eingesetzt. Die APGAS-Bibliothek verwendet für Worker-Threads **ForkJoinThreads** und den **ForkJoinPool** von Java, vgl. [14]. Der ForkJoinPool stattet die Bibliothek mit einem Intra-Place **Workstealing** Algorithmus aus, der Tasks gleichmäßig auf alle zur Verfügung stehenden Threads aufteilt. Der Workstealing Algorithmus des ForkJoinPools ermöglicht es jedem ForkJoinThread, Tasks von anderen ForkJoinThreads zu stehlen, wenn dieser selbst keine weiteren Tasks mehr besitzt.

Das `async`-Konstrukt des APGAS Programmiermodells wird in der APGAS-Bibliothek in den `async` und `asyncAt` Varianten implementiert. Beide Varianten haben gemeinsam, dass der auszuführende Task durch eine Lambdafunktion definiert wird. Diese Lambdafunktion wird den Konstrukten beim Aufruf als Parameter übergeben. `async` erstellt einen Task, der lokal dem ForkJoinPool hinzugefügt wird. Der Workstealing Algorithmus des ForkJoinPools sorgt anschließend dafür, dass der Task ausgeführt wird. Wurde der Task durch einen ForkJoinThread erstellt, reiht dieser den Task in seine eigene Warteschlange ein. Hat jedoch ein anderer Thread den Task erstellt, wird dieser einer gemeinsamen Warteschlange des ForkJoinPools hinzugefügt. Sobald ein ForkJoinThread keine Tasks mehr in seiner eigenen Warteschlange besitzt, stiehlt dieser Tasks aus der

gemeinsamen Warteschlange, oder falls diese ebenfalls leer ist, aus Warteschlangen anderer ForkJoinThreads. `asyncAt` erstellt einen Task, der einem ForkJoinPool eines anderen Places hinzugefügt wird. Dieser Place wird beim Aufruf des `asyncAt`-Konstruktes mit übergeben. Die APGAS-Bibliothek besitzt ein weiteres Konstrukt zum Erstellen von Tasks für andere Places, das **uncountedAsyncAt**-Konstrukt. Tasks welche mit einem `uncountedAsyncAt`-Konstrukt erstellt wurden, werden nicht von einem umgebenden Finish überwacht.

In Quellcode 2.1 ist die Verwendung von `async`, `asyncAt` und `finish` anhand eines Beispiels illustriert. Die Methode *printPlaces()* erstellt für jeden verfügbaren Place einen `asyncAt`-Task (Zeilen 4-6), in dem der jeweilige Place „Hello“ sagt(Zeile 5). Der Master Place gibt zusätzlich an, dass er der Master ist (Zeile 7). Alle Tasks werden über ein `finish` synchronisiert (Zeilen 3-8), bevor eine weitere Ausgabe erzeugt wird mit „All Places have said Hello!“.

```
1  void printPlaces() {
2
3      finish(() -> {
4          places().forEach(place -> {
5              asyncAt(place, () -> LOG.info("Hello from " + here()));
6          });
7          async(() -> LOG.info(here() + " is the Master."));
8      });
9      LOG.info("All Places have said Hello!");
10 }
```

Quellcode 2.1: Hello World mit APGAS

Der Zugriff auf den gemeinsam genutzten partitionierten Speicher aller Places erfolgt mithilfe der Klasse **GlobalRef**. Dieser Zugriff kann dabei in zwei verschiedenen Formen erfolgen:

1. Der Zugriff auf eine bestimmte Variable wird für alle anderen Places freigegeben. Die Referenz zeigt nur auf diese Variable.

2. Es wird eine identische Variable auf jedem Place erstellt, auf die nur der Place selbst zugreifen kann. Die Referenz zeigt immer auf die lokale Variable.

Die Verwendung von GlobalRefs wird im folgenden Quellcode 2.2 näher erläutert. Der Programmcode aus Quellcode 2.2 lädt eine große Menge Messdaten von der Festplatte und durchsucht diese anschließend nach einem bestimmten Wert und zählt dessen Vorkommen. In Zeile 1 wird der zu suchende Wert spezifiziert. Anschließend werden in den Zeilen 2–5 zwei GlobalRef Instanzen erstellt, für jede Form der Verwendung eine. Die erste GlobalRef Instanz referenziert einen `AtomicInteger`, der auf Place 0 abgelegt wird und in dem das Endergebnis der Berechnung abgelegt werden soll. Die zweite GlobalRef Instanz erstellt auf jedem Place ein Integer-Array mit den Messdaten von der Festplatte. Dazu wird die Methode `loadDataSetForPlace()`, Zeile 5, verwendet, die die Datei für den aktuellen Place einliest. Die Implementierung der Methode `loadDataSetForPlace()` wurde aus Platzgründen nicht mit angegeben. Nun wird in Zeile 6 ein `finish`-Konstrukt aufgerufen, in dem für jeden Place (Zeile 7) ein `asyncAt`-Task erstellt wird. Innerhalb des `asyncAt`-Tasks werden die Messdaten nach dem zu suchenden Wert aus Zeile 1 durchsucht (Zeilen 10–14). Jeder Place verarbeitet dabei seinen eigenen Satz an Messdaten, der in der GlobalRef `distributedArray` gespeichert ist. Das lokale Ergebnis wird in der Variablen aus Zeile 9 gespeichert. Das Endergebnis wird in den Zeilen 16–19 bestimmt. Dazu schickt jeder Place innerhalb des `asyncAt`-Tasks sein lokales Ergebnis zum Place 0, dem Besitzer der Variablen `result`, mit einem weiteren `asyncAt`-Task. Der Besitzer der Variablen *result* wird dabei über die Methode `home()` ermittelt. Diese Methode gibt immer denjenigen Place zurück, der die GlobalRef erstellt hat. Der neu erstellt `asyncAt`-Task aktualisiert den Wert des `AtomicInteger`, den die GlobalRef `result` referenziert. In den Zeilen 23 und 24 wird das Endergebnis auf der Konsole ausgegeben.

```
1  final int valueToCount = 15;
2  final GlobalRef<AtomicInteger> result =
3      new GlobalRef<>(new AtomicInteger(0));
4  final GlobalRef<Integer[]> distributedArray =
5      new GlobalRef<>(places(), () -> loadDataSetForPlace(here()));
6  finish(() -> {
7      places().forEach(place -> {
8          asyncAt(place, () -> {
9              int localCount = 0;
10             for (Integer val: distributedArray.get()) {
11                 if (val.equals(valueToCount)) {
12                     localCount++;
13                 }
14             }
15             final Integer finalLocalCount = localCount;
16             asyncAt(result.home(), () -> {
17                 AtomicInteger atomicCount = result.get();
18                 atomicCount.updateAndGet(count -> count+finalLocalCount);
19             });
20         });
21     });
22 });
23 System.out.println("The value " + valueToCount + " was count " +
24     result.get().get() + "times.");
```

Quellcode 2.2: Beispiel der GlobalRef Verwendung

2.2 Globale Lastenbalancierung

Die globale Lastenbalancierung in der APGAS-Bibliothek basiert auf einem Workstealing Algorithmus anhand des Lifelineschemas von X10 [15], welches im Folgenden beschrieben wird. In einem ersten Schritt werden jeweils W Stehlanfragen an zufällige Places geschickt. Sobald einer davon erfolgreich ist, kann der Algorithmus bereits beendet werden. Der Place, der eine Stehlanfrage schickt wird als Dieb bezeichnet und derjenige, der eine Stehlanfrage bekommt wird als Opfer bezeichnet. Sollten die W zufälligen Stehlanfragen alle fehlschlagen, also keine Tasks gestohlen werden können. Dann werden in einem zweiten Schritt bis zu Z viele Stehlanfragen an spezifische Places

geschickt. Sobald auch hier eine Stehlanfrage erfolgreich ist, kann der Algorithmus bereits beendet werden. Diese spezifischen Places werden als Lifeline-Buddys eines Places bezeichnet. Für jeden Lifeline-Boddy eines Places besitzt der Place eine Lifeline. Eine Lifeline hat dabei folgende Eigenschaften:

- Sie wird initial während des Programmstarts mithilfe des Lifelinegraphen generiert.
- Wird eine Stehlanfrage an einen Lifeline-Buddy gestellt, gilt dessen Lifeline als aktiv.
- Wird eine Stehlanfrage mit Tasks beantwortet (das Opfer schickt dem Dieb eine Menge von Tasks), gilt diese nun wieder als inaktiv.
- Kann eine Stehlanfrage von einem Place nicht mit Tasks bedient werden, dann speichert sich das Opfer den Dieb ab und schickt ihm Tasks, sobald er selbst wieder welche bekommen hat.

Der für die Generierung der Lifelines notwendige Graph wird im folgenden Abschnitt 2.3 im Detail erläutert. Die Tasks, die von der globalen Lastenbalancierung gestohlen werden können, müssen über ein `asyncAny`-Konstrukt erstellt werden. Diese Tasks werden im weiteren als **lokalitätsflexible Tasks** bezeichnet, da diese auf jedem Place ausführbar sind und keine Abhängigkeiten zu einem spezifischen Place haben. Ein mit `asyncAny` erstellter Task wird initial dem Place zugewiesen, der das `asyncAny`-Konstrukt ausgeführt hat. Zu einem späteren Zeitpunkt kann dieser Task von anderen Places gestohlen werden.

`asyncAny`-Tasks können aus technischen Gründen nicht zusammen mit `async` und `asyncAt` verwendet werden. Deshalb existiert für die `asyncAny`-Tasks ein eigenes `finishAsyncAny`-Konstrukt. Das `finishAsyncAny`-Konstrukt unterscheidet sich vom `finish`-Konstrukt dadurch, dass es nur für `asyncAny`-Tasks verwendet werden kann und nicht ineinander geschachtelt werden darf, was wiederum technisch begründet ist. In Quellcode 2.3 wird die Verwendung von `finishAsyncAny` und `asyncAny` an einem einfachen Beispiel illustriert. In Zeile 1 wird ein `finishAsyncAny`-Konstrukt aufgerufen, in dessen Lambdafunktion anschließend M `asyncAny`-Tasks erstellt werden (Zeilen 2–6). Die erstellten `asyncAny`Tasks geben die Textnachricht „Hello World from: Place(x)“ auf der Konsole

aus, wobei x derjenige Place ist, der den Task ausführt. Das Programm des Beispiels wird (in den meisten Fällen) unterschiedliche Ausgaben auf der Konsole ausgeben, je nachdem, welcher Place wie viele Tasks stiehlt.

```
1 finishAsyncAny(() -> {
2     for (int i = 0; i < M; i++) {
3         asyncAny(() -> {
4             LOG.info("Hello World from: " + here());
5         });
6     }
7 });
```

Quellcode 2.3: finishAsyncAny und asyncAny Beispiel

Für die Berechnung von Ergebnissen bei der Verwendung von lokalitätsflexiblen Tasks mit `finishAsyncAny` und `asyncAny` existieren zwei weitere Konstrukte. Diese Konstrukte erlauben eine Kombination der Teilergebnisse aller Tasks zu einem Gesamtergebnis. Der für die Kombination verantwortliche Algorithmus ist ein **Reduktionsalgorithmus**. Ein Reduktionsalgorithmus kombiniert eine Menge von Elementen zu einem einzelnen Element. Das kann zum Beispiel das Bilden einer Summe oder auch das Filtern von Elementen sein. Das Teilergebnis einer Berechnung von lokalitätsflexiblen Tasks eines Places wird innerhalb einer Instanz der Klasse **ResultAsyncAny<T>** abgespeichert. Diese Klasse ist eine abstrakte generische Klasse und enthält eine `result` Variable vom Typ T, die das jeweilige Ergebnis repräsentiert. Des Weiteren muss die Methode `mergeResult(...)` vom Nutzer der APGAS-Bibliothek implementiert werden, die den Reduktions-Schritt durchführt. Sobald diese Klasse implementiert ist, kann das Konstrukt **mergeAsyncAny** innerhalb von `asyncAny`-Tasks dazu verwendet werden, das Ergebnis des aktuellen `asyncAny`-Tasks dem Teilergebnis des aktuellen Places hinzuzufügen. Im Anschluss an die Ausführung aller `asyncAny`-Tasks, kann das Gesamtergebnis mithilfe des **reduceAsyncAny**-Konstruktes bestimmt werden. Dazu werden die Teilergebnisse jedes Places über die `mergeResult()` Methode zusammengefasst und anschließend an Place 0 gesandt. Place 0 kombiniert nun diese Teilergebnisse ebenfalls mit der `mergeResult()` Methode zum Gesamtergebnis.

In den nachfolgenden Quellcodes 2.4 und 2.5 ist die Verwendung der neuen Konstrukte `mergeAsyncAny` und `reduceAsyncAny` sowie der Klasse `ResultAsyncAny` anhand der

Berechnung von Pi illustriert. Die Implementierung der `ResultAsyncAny` Klasse verwendet als Ergebnis-Typ eine Double-Variable (Zeile 8) und kombiniert zwei Ergebnisse durch eine Addition (Zeile 12). Für die Berechnung von PI wird eine festgelegte Menge Iterationen verwendet. Die Menge an Iterationen ergibt sich aus der Multiplikation der Parameter N und IN in Zeile 3. N beschreibt die Menge an `asyncAny`-Tasks die verwendet werden sollen (Zeile 4) und der Parameter IN beschreibt die Anzahl an Iterationen pro `asyncAny`-Task (Zeile 9). Die Laufzeit eines `asyncAny`-Tasks muss groß genug sein, um nicht vom Aufwand der Task Erstellung dominiert zu werden. Weshalb an dieser Stelle eine Aufteilung der Iterationen in N und IN vorgenommen wurde. Sobald ein `asyncAny`-Task seine Arbeit beendet hat, führt er sein Teilergebnis mit den bisherigen Ergebnissen des Places zusammen (Zeile 14). Dazu wird eine neue Instanz der Klasse `PiResult` erstellt. Diese Klasse ist in Quellcode 2.5 dargestellt und erbt von der Klasse `ResultAsyncAny`. Die `mergeResult()` Methode der Klasse `ResultAsyncAny` ist in den Zeilen 8-10 definiert und bildet die Summe aus dem aktuellen Ergebnis `result` und dem neuen Zwischenergebnis `r`. In der letzten Zeile des Quellcodes 2.4 wird das Gesamtergebnis mittels eines `reduceAsyncAny`-Konstruktes bestimmt.

```
1  finishAsyncAny(
2      () -> {
3          final double deltaX = 1.0 / (N*IN);
4          for (int i = 0; i < N; i++) {
5              final int _i = i;
6              asyncAny(
7                  () -> {
8                      double result = 0;
9                      for (int j = 0; j < IN; j++) {
10                         double x = (_i*IN+j + 0.5) * deltaX;
11                         double r = (4.0 / (1 + x * x)) * deltaX;
12                         result += r;
13                     }
14                     mergeAsyncAny(new PiResult(result));
15                 });
16         }
17 });
18 PiResult result = (PiResult) reduceAsyncAny();
```

Quellcode 2.4: Beispiel der Reduktion in APGAS

```
1  public class PiResult extends ResultAsyncAny<Double> {
2
3    public PiResult(double r) {
4      this.result = r;
5    }
6
7    @Override
8    public void mergeResult(ResultAsyncAny<Double> r) {
9      this.result += r.getResult();
10   }
11 }
```

Quellcode 2.5: Ergebnis Klasse

2.3 Lifelinegraph

Zur Bestimmung der Lifelines wird ein spezieller Graph verwendet, in dem alle verfügbaren Places durch einen Knoten repräsentiert werden. Eine Kante von einem Knoten zu einem anderen stellt eine Lifeline dar. Kanten im Lifelinegraphen sind gerichtet und bedeuten, dass der Knoten mit der ausgehenden Kante den Knoten mit der eingehenden Kante als Lifeline-Buddy hat. D. h. der Knoten mit der ausgehenden Kante stiehlt Tasks vom Knoten mit der eingehenden Kante. Die Grundidee hinter dem Lifelinegraphen ist, die Knoten so miteinander zu verbinden, dass möglichst wenig Schritte notwendig sind, um von einem Knoten zu jedem anderen zu gelangen. Dies hat den Vorteil, dass sich die Arbeit schnell verteilt. Dazu könnte man einfach jedem Knoten eine Kante zu jedem anderen Knoten hinzufügen. Dies hätte allerdings den Nachteil, dass dann jeder Knoten im schlechtesten Fall (nur einer oder keiner hat Arbeit) jede Kante überprüfen müsste um Arbeit zu bekommen. Deshalb wird der Lifelinegraph so aufgebaut, dass dieser möglichst wenige Kanten pro Knoten besitzt, aber gleichzeitig die kürzesten Pfade von jedem Knoten zu jedem anderen minimiert. Der Lifelinegraph muss dafür die folgenden Eigenschaften erfüllen:

- Es ist ein gerichteter Graph.
- Jeder Knoten muss mindestens eine ausgehende Kante besitzen.
- Der Graph muss verbunden sein (von jedem Knoten aus muss jeder andere Knoten über einen Pfad beliebiger Länge erreichbar sein).
- Der Graph soll einen geringen Durchmesser besitzen (Es sollen möglichst kurze Pfade von jedem Knoten zu jedem anderen existieren)

Für den Lifelinegraphen wurde die Klasse der zyklischen Hyperkuben gewählt, da diese alle oben genannten Eigenschaften erfüllen. Die formale Beschreibung des Lifelinegraphen entspricht der aus [15]. Ein Lifelinegraph wird über die Parameter **Z** und **L** definiert. Der Parameter Z gibt die Dimension des Hyperkubus an und der Parameter L die Anzahl von Knoten pro Dimension. Die Parameter Z und L ließen sich auch so interpretieren: Z gibt die Menge an Kanten (Lifelines) pro Knoten an und L die maximale Länge eines kürzesten Pfades. Außerdem muss für die Parameter die folgende Bedingung erfüllt sein: $L^{Z-1} < P <= L^{Z}$ mit $P = AnzahlderKnoten$. Diese Bedingung garantiert, dass der Lifelinegraph für die gegebene Menge an Knoten die obigen Eigenschaften erfüllt. Darüber hinaus schreibt die Bedingung vor, dass die geringste Dimension Z bei gegebenem L zu wählen ist. Alle Knoten des Graphen können von 0 bis P-1 durchnummeriert werden. Jede dieser Nummern ließe sich nun ebenfalls durch eine Zahl mit der Basis L und Z Ziffern darstellen. Abbildung 2.1 illustriert einen Lifelinegraphen für die Parameter Z=2, L=3 und P=9 Places mit den Knoten von 0-9 durchnummeriert. In Abbildung 2.2 ist der gleiche Lifelinegraph für Z=2, L=3 und P=9 mit der alternativen Darstellung der Knoten als Zahlen mit der Basis 3 und 2 Ziffern dargestellt. Den Lifelinegraphen der Abbildung 2.2 könnte man sich nun auch als ein Gitter im zweidimensionalen Raum vorstellen mit den Koordinaten (x,y) für jeden der Knoten. Die Darstellung der Knoten als Zahlen mit der Basis L und Z Ziffern soll im weiteren als Komponentendarstellung bezeichnet werden. Die einzelnen Z Ziffern werden als Komponenten bezeichnet.

Der Lifelinegraph ist dezentral und es befindet sich immer nur derjenige Teil des Lifelinegraphen auf einem Place, der dessen Lifelines beschreibt. Die Lifeline-Buddys eines Places werden aus den ausgehenden Kanten des Knotens abgeleitet, der den jeweiligen Place repräsentiert. Der jeweilige für den Place relevante Teil des Lifeline-

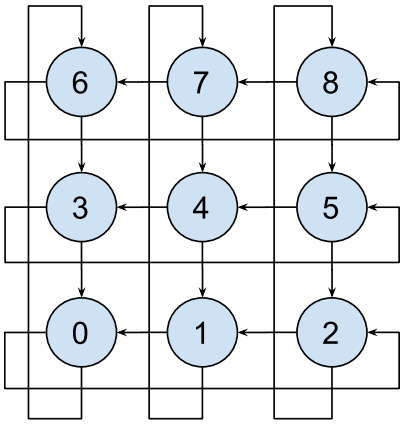

Abbildung 2.1: Lifelinegraph für L=3, Z=2, P=9

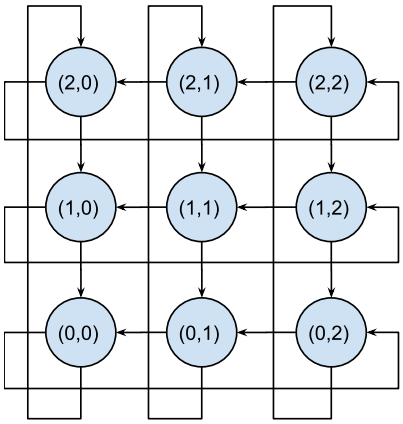

Abbildung 2.2: alternativer Lifelinegraph für L=3, Z=2, P=9

graphen, der dessen Lifelines beschreibt wird während der Initialisierung des Places generiert. Dazu werden im ersten Schritt die Parameter L und Z aus der Anzahl der verfügbaren Places berechnet. Für den Parameter L wird der maximale Wert gewählt, dessen Quadrat kleiner ist als die Anzahl der Places P, formal: $L^2 < P <= (L+1)^2$. Aus L wird anschließend der Parameter Z berechnet, indem der maximale Wert gesucht wird, sodass $L^Z < P <= L^{(Z+1)}$ erfüllt ist. Beide Parameter werden nun in dem Algorithmus aus Quellcode 2.6 verwendet, um die Lifeline-Buddys des Places zu bestimmen.

```
1  int x = 1;
2  int y = 0;
3  int h = here().id;
4  int[] lifelines = new int[Z];
5  for (int j = 0; j < Z; j++) {
6      int v = h;
7      for (int k = 1; k < L; k++) {
8          v = v - v % (x * L) + (v + x * L - x) % (x * L);
9          if (v < P) {
10             lifelines[y++] = v;
11             break;
12         }
13     }
14     x *= L;
15 }
```

Quellcode 2.6: Algorithmus zur Bestimmung des Lifelinegraphen

Der Algorithmus aus Quellcode 2.6 basiert auf der Komponentendarstellung der Knoten des Lifelinegraphen. Die Grundidee des Algorithmus ist, eine Kante (Lifeline) für jede Dimension Z zu erstellen, indem ausgehend von jedem Knoten dessen Komponenten jeweils um eins verringert werden. In Abbildung 2.2 lässt sich die Grundidee sehr gut nachvollziehen. Ausgehend vom mittleren Knoten 4 (1,1) werden Z=2 Kanten erstellt, zu den Knoten 3 (1,0) und 1 (0,1). Der Algorithmus beginnt mit der Initialisierung der Parameter `x`, `y` und `h` in den Zeilen 1-3, sowie der Initialisierung des Lifeline-Buddy Arrays in Zeile 4. `x` ist ein Hilfsparameter in dem die Schrittweite für die aktuelle Dimension von Z gespeichert wird und `y` bestimmt den Index des aktuell berechneten Lifeline-Buddys. Die Schrittweite definiert, wie viele Knoten weiter sich der nächste Knoten für den Schritt von k nach k+1 befindet. Im Parameter `h` wird die ID des Places gespeichert, für den aktuell die Lifeline-Buddys berechnet werden. Zeile 5 verwendet eine Schleife, um über die Anzahl der zu erstellenden Lifeline-Buddys zu iterieren (Parameter Z). In Zeile 6 wird die Hilfsvariable `v` erstellt, die die aktuelle ID des Ziel Places enthält, d. h. den späteren Lifeline-Buddy. `v` wird initial mit der ID des Places initialisiert, für den die Lifeline-Buddys generiert werden. Ausgehend von diesem Place, werden anschließend die möglichen Lifeline-Buddys berechnet und die erste ID gewählt, die geringer ist als die maximale Anzahl an Places. Dazu wird in Zeile 7 über die Gesamtlänge von L iteriert und in Zeile 8 die ID des Places der Lifeline berechnet. Die

Berechnung der ID des Lifeline-Buddys entspricht der Verringerung der Komponente j um eins. Die Komplexe Berechnung mithilfe der Modulo Operanden sorgt dafür, dass der gültige Bereich (0 bis (P-1)) von IDs nicht verlassen wird. Sobald die erste gültige ID gefunden wurde, Zeile 9, wird diese als Lifeline-Buddy im Array `lifelines` an der Position `y` abgelegt (Zeile 10).

3 Konzept der Ressourcen-Elastizität

In Kapitel 1 wurde der Begriff Ressourcen-Elastizität bereits definiert als die Anpassung eines Programms während seiner Ausführung an sich ändernde Ressourcen innerhalb eines HPC Clusters. Die Ressourcen-Elastizität unterscheidet zwei unterschiedliche Prozesse zur Anpassung an sich ändernde Ressourcen. Das Hinzufügen weiterer Places zur Nutzung neuer Ressourcen, das im Abschnitt 3.1 beschrieben wird, und das Entfernen von Places bei sich verringernden Ressourcen. Auf letzteres wird im Abschnitt 3.2 näher eingegangen. Beide Prozesse stützen sich auf neue Konzepte im Bereich der Referenzen. Diese werden in Abschnitt 3.3 erläutert und beinhalten neben allgemeinen Modifikationen die neuen Konzepte der **Migration** und der **Reduktion**. Abschließend legt Abschnitt 3.4 dar, wie der Lifelinegraph während der Ressourcen-Elastizitäts-Prozesse angepasst werden muss, um auch weiterhin eine optimale Lasten-Balancierung zu gewährleisten.

3.1 Hinzufügen von Places

Um dem Programm zur Laufzeit neue Ressourcen in Form von weiteren Knoten des Clusters zur Verfügung zu stellen, müssen diese als neue Places hinzugefügt werden. Dabei muss darauf geachtet werden, dass diese korrekt konfiguriert und initialisiert werden. Konfiguration und Initialisierung stellen dabei zwei unterschiedliche Teilaufgaben dar. Unter Konfiguration versteht man, das Einstellen der Netzwerkparameter, der zu verwendenden Ressourcen (z.b. die Anzahl der Threads) und das festlegen der Place-spezifischen Eigenschaften wie zum Beispiel dessen ID. Die Initialiserung des

Places beschreibt das Kopieren der notwendigen Anwendungsdaten von anderen Places auf den zu initialiserenden Place. Der Begriff der Anwendungsdaten umfasst hierbei alle durch den Programmierer erstellten Variablen, die für die korrekte Ausführung des Programms notwendig sind. Das korrekte Konfigurieren eines Places ist eher von technischer Natur und unterscheidet sich von Implementierung zu Implementierung, sodass in diesem Kapitel nicht näher darauf eingegangen wird.

Für das korrekte Initialisieren der Anwendungsdaten würde ein einfaches Kopieren der Daten von einem der Places nicht ausreichen. Das Problem beim einfachen Kopieren ist, dass es aufgrund der bereits ausgeführten Berechnungen dazu gekommen sein kann, dass sich diese Daten verändert haben. Ein neuer Place benötigt eine Initialisierung mit den Daten, die bei Programmstart vorlagen. Diese Notwendigkeit ergibt sich aus dem Berechnungsmodell der lokalitätsflexiblen Tasks. Ein Task führt seine Berechnungen immer ohne Annahmen über etwaige Zustände oder Variablen aus, da sich diese potenziell von Place zu Place unterscheiden. Ein Beispiel dazu wäre, eine Variable, in der ein Zwischenergebnis gespeichert wird. Würde diese Variable nach der Ausführung mehrerer Tasks einfach kopiert werden, würde dessen Zwischenergebnis ebenfalls kopiert werden. Dies käme dem doppelten Ausführen der Berechnungen gleich und führt zu einem falschen Ergebnis der gesamten Berechnung. Aus diesem Grund wird der initiale Zustand der Variablen vom Programmstart und vor der Ausführung irgendeiner Berechnung benötigt, denn diese enthält noch keine Zwischenergebnisse. Die Ressourcen-Elastizität verwendet dafür das PGAS-Modell und erweitert dieses um serialisierte Initialisierungsfunktionen in Referenzen, siehe 3.3.

Sobald alle neuen Places korrekt initialisiert wurden, wird der Lifelinegraph und alle Lifelines aller Places aktualisiert, näheres im Kapitel 3.4. Im Anschluss können die neu hinzugefügten Places ihre Lasten-Balancierung aktivieren, sodass diese damit beginnen können, sich Tasks von anderen Places zu stehlen. Der Prozess des Hinzufügens wird vom Master Place initiiert und durchgeführt. Alle anderen Places sind von diesem Prozess nicht betroffen. Die Berechnungen und auch die Lasten-Balancierung aller aktiven Places und des Masters bleiben aktiv, sodass eine möglichst hohe Performance auch während des Hinzufügens von neuen Places gewährleistet ist.

3.2 Entfernen von Places

Der zweite Anwendungsfall der Ressourcen-Elastizität ist das Entfernen von Places zur Laufzeit. Dies erfordert ebenfalls, dass der Lifelinegraph angepasst wird, siehe Abschnitt 3.4. Wie beim Hinzufügen von Places, sind beim Entfernen ebenfalls die vorhandenen Anwendungsdaten und zusätzlich dazu auch die vorhandenen Tasks zu berücksichtigen. Innerhalb einiger Variablen eines Places, der entfernt wird, können relevante Zwischenergebnisse oder auch das Endergebnis einer Berechnung enthalten sein. Beim Entfernen eines Places müssen diese Daten auf einen anderen Place gesichert werden, sodass diese nicht verloren gehen. Ebenfalls muss sichergestellt werden, dass auf diese Daten beim Berechnen des Endergebnisses wieder zugegriffen werden kann. Um diese beiden Punkte zu gewährleisten, wurden ebenfalls die Referenzen des PGAS-Modells benutzt, um die Konzepte der **Migration** und der **Reduktion**. Die Migration verschiebt die Werte der Referenzen auf einen anderen Place und biegt die Zeiger der Referenzen auf den neuen Place um. Auf die Migration wird in Unterabschnitt 3.3 näher eingegangen. Die **Reduktion** dient der Zusammenführung aller Teilergebnisse aller Places zu einem Gesamtergebnis und ist abgeleitet von der Reduktion aus Kapitel 2. Neben den Teilergebnissen der aktiven Places berücksichtigt die Reduktion ebenfalls die Werte der migrierten Referenzen. Eine ausführlichere Beschreibung der Reduktion findet sich in Unterabschnitt 3.3.

Abbildung 3.1 illustriert den Algorithmus, der zum Entfernen von Places verwendet wird. Zu Beginn muss der Lifelinegraph angepasst werden, indem die zu entfernenden Places von diesem entfernt werden, Details dazu siehe 3.4. Der Lifelinegraph muss vor dem Entfernen der Places aktualisiert werden, da auf diese Weise verhindert wird, dass weitere Stehlanfragen an die zu entfernenden Places geschickt werden. Anschließend werden alle zu entfernenden Places darüber informiert, dass diese entfernt werden sollen. Das Informieren der zu entfernenden Places geschieht parallel, indem jedem der Places eine entsprechende Nachricht vom Master Place geschickt wird. Die Nachricht enthält zusätzlich eine Liste mit allen zu entfernenden Places, was im Nachfolgenden noch wichtig wird.

Alle weiteren Schritte, die nun noch für das Entfernen von Places auszuführen sind, werden von den zu entfernenden Places ausgeführt. Ein wichtiger Aspekt an dieser Stelle

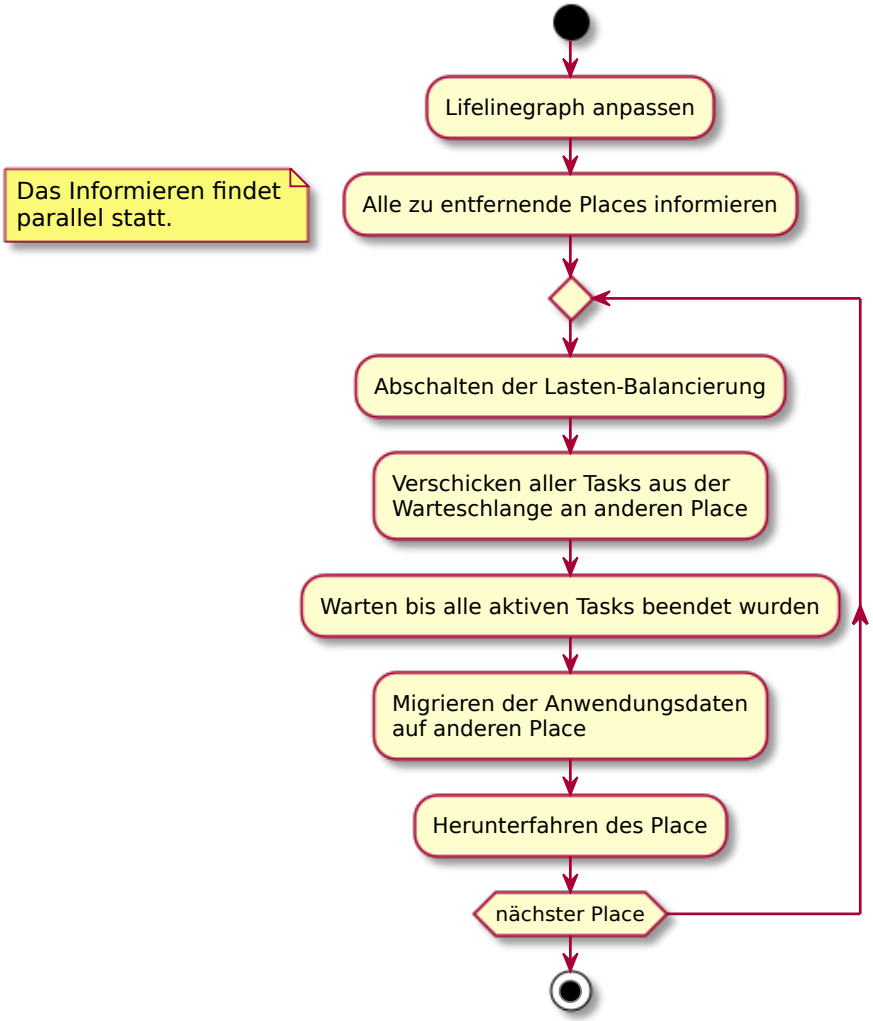

Abbildung 3.1: Algorithmus: Entfernen eines Places

ist, dass die Lastenbalancierung an dieser Stelle angepasst werden musste. Der bisherige Workstealing Algorithmus der Lastenbalancierung hat beim Beantworten einer aktiven Lifeline durch das Opfer direkt Tasks an den Dieb geschickt. Stehlanfragen können allerdings auch zeitversetzt beantwortet werden. Das zeitversetzte Beantworten einer Stehlanfrage geschieht immer dann, wenn ein Opfer zum Zeitpunkt der Stehlanfrage keine Tasks hatte und sich den Dieb gespeichert hat (die Lifeline blieb aktiv). Sollte allerdings der Dieb bis zum Zeitpunkt der Beantwortung der Stehlanfrage entfernt worden sein, kann es zum Verlust der Tasks kommen. Aus diesem Grund wurde die Lastenbalancierung nun so angepasst, dass der Dieb nur über verfügbare Tasks informiert wird, anstatt direkt Tasks zu schicken. Der Dieb kann dann erneut eine Stehlanfrage schicken. Jeder Place, der entfernt werden soll, schaltet seine Lasten-Balancierung ab, um keine weiteren Stehlanfragen an andere Places zu schicken. Als Nächstes werden alle Tasks, die sich in der Warteschlange des Places befinden und aktuell nicht ausgeführt werden, zu einem anderen Place geschickt. Die Auswahl des Empfänger Places berücksichtigt dabei alle zu entfernenden Places aus der obigen Nachricht, sodass die Tasks an keinen dieser Places geschickt wird. Außerdem wird versucht bei multiplen zu entfernenden Places die Tasks all dieser Places nicht zu einem einzigen Place zu schicken, sondern diese auf die verbleibenden Places zu verteilen. Dies wird erreicht, indem die erste Lifeline des jeweiligen zu entfernenden Places verwendet wird, um dem Place der Lifeline die Tasks zu schicken. Die Eigenschaften des Lifelinegraphen garantieren hier eine möglichst gute Verteilung, siehe 2.3. Nachdem nun alle wartenden Tasks verschickt wurden, wird gewartet, bis der letzte aktive Task eines jeden Threads abgearbeitet wurde. Sobald keine Tasks mehr ausgeführt werden, werden die Anwendungsdaten des Places migriert. Als Ziel der Migration wird der gleiche Place verwendet, zu dem bereits die Tasks geschickt wurden. Abschließend wird der jeweilige Place heruntergefahren.

3.3 Ressourcen-Elastische Referenzen

In Kapitel 2.1 wurden bereits die Referenzen eingeführt. Diese wurden nun für die Ressourcen-Elastizität in mehreren Punkten erweitert. In einem ersten Schritt wurden die Referenzen in zwei unterschiedliche Typen aufgeteilt. In die globalen Referenzen

und die lokalen Referenzen. Beide sind angelehnt an die zwei verschiedenen Varianten der `GlobalRef` aus Kapitel 2.1. Die globalen Referenzen stellen eine Referenz auf einen einzelnen Wert dar, der auf nur einem der Places liegt. Jeder Place kann auf diesen Wert über die globale Referenz zugreifen. Die lokalen Referenzen verweisen dagegen immer auf Werte auf dem aktuellen (lokalen) Place. Es kann also jeder Place nur auf seine eigenen Werte über lokale Referenzen zugreifen.

Die lokalen Referenzen wurden zusätzlich um serialisierte Initialisierungsfunktionen erweitert, sodass deren initialer Zustand auf allen Places durch diese Funktion initialisiert werden kann. Diese Funktion dient zusätzlich der späteren Initialisierung neuer Places durch den Master-Place. Ein initialer Wert wäre hier für viele Anwendungsfälle bereits ausreichend, jedoch bietet eine Funktion zum Initialisieren mehr Flexibilität als ein einfacher Wert. Die Möglichkeit von Java, Lambda-Funktionen zu serialisieren, bietet sich hier sehr gut an, um den initialen Zustand zu konservieren. Die Serialisierung der Lambdafunktion ermöglicht zusätzlich Seiteneffekte zu vermeiden, wie der Zugriff auf andere veränderliche Variablen aus der Lambda-Funktion heraus. Lediglich lokale Referenzen benötigen eine solche konservierte Initialisierung. Globale Referenzen jedoch benötigen diese nicht, da deren Wert bereits auf einem der Places gespeichert ist und daher keine weitere Initialisierung mehr benötigt. Sollte eine globale Referenz verschoben werden, ist auch dann keine konservierte Initialisierung notwendig, da die zwischenzeitlichen Veränderungen des Wertes erhalten bleiben müssen.

Migration

Die **Migration** von Referenzen beschreibt das Verschieben ihrer lokal gespeicherten Werte auf einen anderen Place. Dies kann erst zu einem Zeitpunkt geschehen, zu dem sichergestellt ist, dass der Wert der Referenz eines Places sich nicht mehr ändern kann bzw. kein anderer Place mehr darauf zugreifen kann. Ansonsten könnte es zu Datenverlusten kommen. Die Migration wird deshalb erst dann durchgeführt, wenn es keine aktiven Tasks mehr auf einem Place gibt. Damit Zugriffe von außen während der Migration nicht stattfinden können, blockiert die Migration sämtliche Zugriffe auf den Wert der Referenz, die gerade migriert wird. Dazu werden die lokalen Werte der Referenz mittels eines Locks gesperrt. Dieser Lock wird erst dann wieder freigegeben, wenn die Migration

abgeschlossen ist. Globale und lokale Referenzen müssen aufgrund ihrer Eigenschaften unterschiedlich migriert werden:

Migration globaler Referenzen

Globale Referenzen speichern auf jedem Place zu jeder globalen Referenz einen Besitzer ab. Dieser Besitzer gibt an, auf welchem Place sich der Wert der globalen Referenz befindet. Zur Migration einer globalen Referenz werden die folgenden 5 Schritte durchgeführt:

1. Setze Lock
2. Kopiere Wert zum neuen Place
3. Setze den Besitzer der Referenz auf den neuen Place beim ursprünglichen Place und dem neuen Place
4. Entsperre Lock
5. Setze den Besitzer der Referenz auf den neuen Place bei allen übrigen Places

Die Migration ist ein asynchroner Prozess, der über eine Weiterleitung von Nachrichten den Wert und alle Zugriffe darauf zusätzlich zum Lock absichert. Aufgrund des Locks können sich Anfragen von anderen Places an dieser globalen Referenz ansammeln, welche nach der Freigabe des Locks ausgeführt werden müssen. Da der Wert der globalen Referenz allerdings in der Zwischenzeit auf einen anderen Place transferiert wurde, können diese Anfragen nun lokal nicht mehr beantwortet werden. Aus diesem Grund wird bei jeder Anfrage die lokale Referenz überprüft, ob dieser Place der Besitzer des Wertes ist. Hat sich dies inzwischen geändert, wird die Anfrage an den neuen Place weitergeleitet.

Migration lokaler Referenzen

Bei der Migration einer lokalen Referenz wird der Wert der lokalen Referenz eines Places innerhalb einer speziellen Datenstruktur eines anderen Places gespeichert. Die spezielle Datenstruktur speichert ebenfalls zu welcher Referenz jeder migrierte Wert gehörte. Da bei lokalen Referenzen keine Zugriffe von außen stattfinden können und es zum Zeitpunkt der Migration keine Tasks mehr auf dem aktuellen Place gibt, benötigt die Migration von lokalen Referenzen keine weiteren Sicherungen. Die migrierten Werte einer lokalen Referenz innerhalb der speziellen Datenstruktur werden anschließend nicht mehr verändert. Denn der ursprüngliche Place wird nach Abschluss der Migration heruntergefahren. Da es in den meisten Fällen nicht notwendig ist, die Werte aller lokalen Referenzen zu sichern, können einzelne lokale Referenzen auch als **Transient** markiert werden. Diese als Transient markierten lokalen Referenzen werden von der Migration ausgeschlossen, um den Speicherbedarf so gering wie möglich und die Performance so hoch wie möglich zu halten. Diese gesicherten Werte werden vor allem dann wichtig, wenn das Endergebnis aus allen Zwischenergebnissen jedes Places einer Berechnung bestimmt werden soll, siehe Unterabschnitt 3.3.

Reduktion

Die **Reduktion** wird durchgeführt, sobald eine Berechnung abgeschlossen und das Endergebnis berechnet werden soll. Eine Reduktion kann lediglich auf lokalen Referenzen ausgeführt werden und dient dazu die verteilten Einzelwerte aller Places zu kombinieren. Wie die einzelnen Werte miteinander kombiniert werden sollen, gibt eine Reduktionsfunktion an. Damit ist die Reduktion sehr ähnlich der Reduktion aus Kapitel 2.2. Jedoch mit dem Unterschied, dass diese in die lokalen Referenzen integriert ist. Es ist daher nicht mehr notwendig, die `ResultAsyncAny` Klasse zu implementieren und die Konstrukte `mergeAsyncAny` und `reduceAsyncAny` zu verwenden. Stattdessen wird der lokalen Referenz eine Reduktionsfunktion übergeben und diese kombiniert anschließend alle Werte zu einem Endergebnis. Die Integration der Reduktion in die lokalen Referenzen bringt zusätzlich den Vorteil, dass diese für jede lokale Referenz ausgeführt werden kann. Die ursprüngliche Reduktion aus Kapitel 2.2 legte die Reduktion auf einen einzelnen Wert mit einem spezifischen Typ (siehe `ResultAsyncAny<T>`)

fest. Lokale Referenzen können in einer beliebigen Anzahl verwendet werden. Darüber hinaus berücksichtigt die erweiterte Reduktion ebenfalls die gesicherten Werte aus der Migration.

3.4 Lifelinegraph Modifikationen

Jeder der beiden vorherigen Ressourcen-Elastizitätsprozesse involviert die Modifikation des Lifelinegraphen, auf dem die globale Lasten-Balancierung basiert. Beim Hinzufügen von Places wird der Lifelinegraph nach dem Hinzufügen aktualisiert und beim Entfernen davor. Dies hat den Grund, dass auf diese Weise verhindert wird, dass Stehlanfragen an nicht mehr existente Places oder noch nicht vorhandene Places geschickt werden. Diese Arbeit verwendet einen zentral vom Master Place verwalteten Lifelinegraphen anstelle eines über alle Places verteilten Lifelinegraphen. Die aus dem Lifelinegraphen abgeleiteten Lifelines sind jedoch über alle Places verteilt. Diese Entscheidung wurde aufgrund der hohen Komplexität eines verteilten Lifelinegraphen getroffen. Diese wird im folgenden kurz erläutert.

Die Komplexität bei der Modifikation eines verteilten Lifelinegraphen läge in den Abhängigkeiten zwischen den Places und ihren Lifelines. Das Entfernen eines Places aus dem Lifelinegraphen hätte oftmals Folgen für die Lifelines eines jeden mit diesem Place verbundenen Place im Lifelinegraphen. Hier müsste dann mindestens eine Nachricht zwischen den Places ausgetauscht werden. Eine zusätzliche Hürde wäre hier ebenfalls die asynchrone Eigenschaft der benötigten Nachrichten. Die asynchrone Eigenschaft würde zu erhöhtem Synchronisierungsaufwand führen, um zu garantieren, dass die Sicht auf den Lifelinegraphen von jedem Place aus zu jedem Zeitpunkt identisch ist.

Der zentral vom Master Place verwaltete Lifelinegraph verteilt die Lifelines nach Änderungen am Lifelinegraphen direkt an die jeweiligen Places. Dies hat den Vorteil, dass keine Synchronisierung für die Berechnung oder Modifikation des Lifelinegraphen notwendig ist. Der Berechnungsaufwand für die Erstellung des Lifelinegraphen durch nur einen Place kann vernachlässigt werden, da dessen Berechnung lediglich in O(n) von der Anzahl `n` an Places abhängt. Die korrekte Reihenfolge der Update-Nachrichten für die

Lifelines eines jeden Places wird über eine eindeutige Nummerierung der Nachrichten garantiert.

Generieren und Hinzufügen von Knoten zum Lifelinegraphen

Die Berechnung des Lifelinegraphen im Master Place ist im Aktivitäts-Diagramm in der Abbildung 3.4 illustriert. Der grundlegende Algorithmus aus Kapitel 2.2, der die Lifelines eines Places berechnet, wurde dabei nicht verändert. Ein Knoten referenziert einen Place im Graphen und besitzt eine eindeutige Knoten-ID. Der Lifelinegraph selbst liegt als eine sortierte Liste von Knoten im Speicher des Master Places vor. Eine Kante von einem Knoten zu einem anderen im Lifelinegraphen entspricht einer Lifeline im Lifelinegraphen. Ein jeder Knoten des Lifelinegraphen besitzt eine Menge von Kanten, die seinen Lifelines entspricht. Diese Darstellung des Lifelinegraphen entspricht der Adjazenzlistendarstellung von Graphen. Eine Kante (Lifeline) speichert die ID des Ziel-Knotens und kann folgendermaßen gelesen werden: „Der referenzierte Place des Knotens X besitzt eine Lifeline zum Knoten mit der ID der Lifeline“.

Die Generierung des Lifelinegraphen entspricht der iterativen Ausführung von Hinzufüge-Operationen für jeden der Knoten, beginnend mit einem leeren Lifelinegraphen. In einem ersten Schritt werden zuerst die Parameter L und Z anhand der Anzahl der Places berechnet. Dabei kommt es häufig vor, dass $L^Z > P$ gilt. Wenn dies der Fall ist, entstehen Lifelinegraphen, die nicht voll besetzt sind. In der Abbildung 3.3 ist ein solcher nicht vollständig besetzter Lifelinegraph dargestellt. Die Abbildung 3.2 zeigt hingegen einen voll besetzten Lifelinegraphen. Der unvollständig besetzte Lifelinegraph besitzt für die Knoten 6 und 5 andere Lifeline-Buddys, als der vollständig besetzte Lifelinegraph. Der Knoten 6 hat zum Beispiel im vollständig besetzten Lifelinegraphen die Lifeline-Buddys 8 und 3, im unvollständigen Lifelinegraphen jedoch die Lifeline-Buddys 7 und 3. Diese unterschiedlichen Lifeline-Buddys werden im Folgenden als **Ziel-Alternativen** einer Lifeline bezeichnet.

Im Anschluss an die Berechnung von L und Z, wird über die Menge der Places iteriert, für welche der Lifelinegraph erstellt werden soll. Jede dieser Iterationen entspricht dem

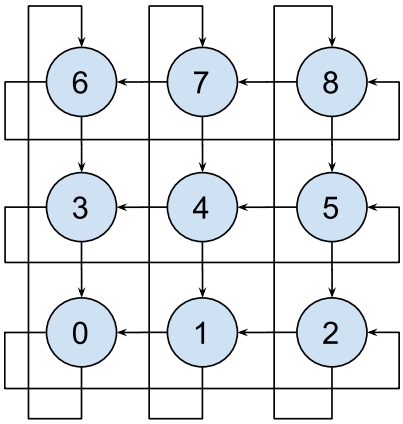

Abbildung 3.2: Lifelinegraph für L=3, Z=2, P=9

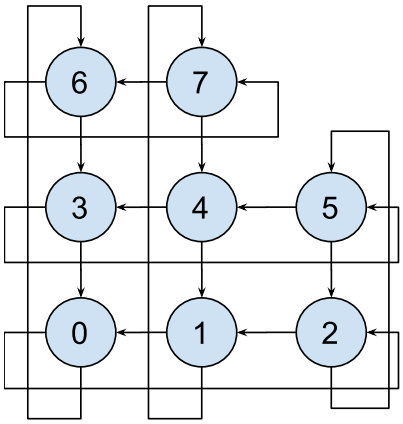

Abbildung 3.3: Lifelinegraph (nicht voll besetzt) für L=3, Z=2, P=9

Hinzufügen eines neuen Knotens zum Lifelinegraphen und befindet sich im Block **Hinzufügen_von_Knoten**. Für jeden dieser Places wird nun ein Knoten im Lifelinegraphen erstellt und diesem hinzugefügt. Jedem Knoten wird bei seiner Erstellung der Place zugeordnet, für den er erstellt wurde. Zusätzlich erhält jeder Knoten eine eindeutige Knoten-ID, die seiner Position (Dezimaldarstellung der Komponentendarstellung) im Lifelinegraphen entspricht. Jeder neu erstellte Knoten wird dabei an das Ende des Lifelinegraphen angefügt. Auf diese Weise ist der Lifelinegraph von Anfang an aufsteigend nach der ID der Knoten sortiert, was für das Entfernen von Knoten noch relevant wird, siehe Unterabschnitt 3.4. Im Anschluss an das Hinzufügen eines neuen Knotens zum Lifelinegraphen werden dessen Kanten erstellt. Das Erstellen einer Kante beinhaltet das Berechnen der Ziel-Alternativen derjenigen Kante. Diese Berechnung entspricht mit einer Ausnahme exakt dem Algorithmus aus Quellcode 2.6 Zeilen 7-13. Lediglich die if-Bedingung aus Zeile 9 wurde entfernt, um alle Ziel-Alternativen für jeden möglichen Place berechnen zu können. Sobald alle Kanten mit ihren Ziel-Alternativen erstellt wurden, wird für jede Kante der Lifeline-Buddy bestimmt. Als Lifeline-Buddy wird die größte Ziel-Alternative (ID) gewählt, die kleiner als P ist.

Der gesamte Block **Aktualisieren_des_Knotens** in Abbildung 3.4 wird als Aktualisierung eines Knotens bezeichnet. Dieser Block wird immer dann ausgeführt, wenn sich die Lifelines eines Knotens geändert haben könnten und sorgt dafür, dass die Kanten gegebenenfalls korrigiert oder ergänzt werden. Sollen während der Laufzeit aufgrund

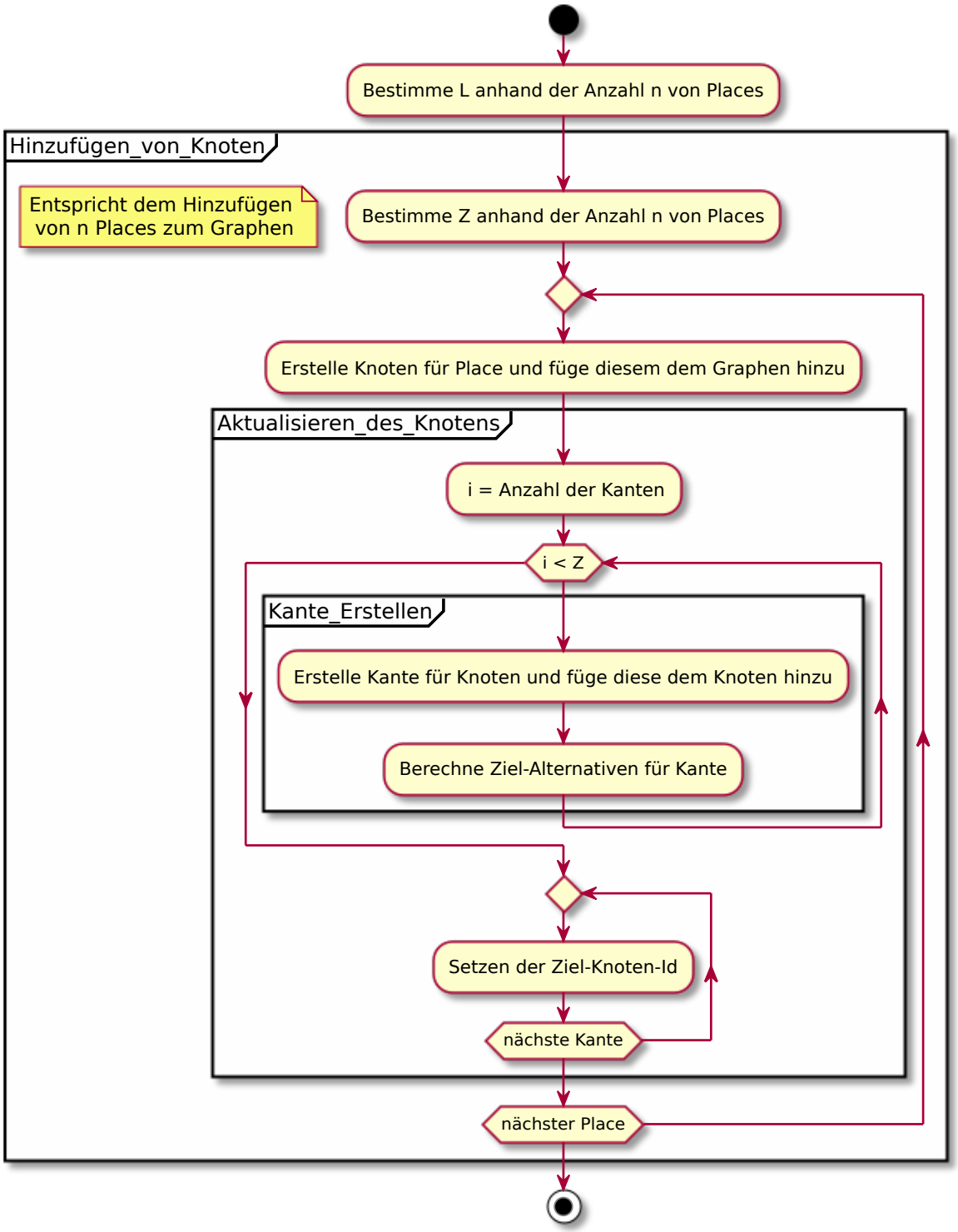

Abbildung 3.4: Erstellen eines Lifelinegraphen, Hinzufügen von Places

eines Ressourcen-Elastizitäts-Prozesses weitere Places dem Lifelinegraphen hinzugefügt werden, wird für diese neuen Places lediglich der Teil des Algorithmus ausgeführt, welcher in der Gruppe **Hinzufügen_von_Places** zu finden ist. Weitere Knoten werden dabei ebenfalls immer am Ende des Lifelinegraphen hinzugefügt, sodass dieser aufsteigend sortiert bleibt.

Entfernen von Knoten

Innerhalb eines Ressourcen-Elastizitäts-Prozesses kann es ebenfalls vorkommen, dass Knoten zu entfernender Places aus dem Lifelinegraphen entfernt werden müssen. Das Aktivitäts-Diagramm in Abbildung 3.5 illustriert den Algorithmus, welcher ausgeführt wird, um Knoten wieder zu entfernen. Für die zu entfernenden Places werden zuerst die Knoten bestimmt, die die Places repräsentieren. Anschließend werden diese Knoten aus dem Lifelinegraphen entfernt. Dieser Schritt hat oftmals zur Folge, dass der Lifelinegraph repariert werden muss. Die Reparatur ist notwendig, da aufgrund des Entfernens von Knoten aus dem Lifelinegraphen ein unvollständig verbundener Graph entstehen kann. Ein unvollständig verbundener Graph enthält Knoten, die von anderen Knoten des Graphen nicht mehr erreicht werden können. Für die globale Lasten-Balancierung würde dies bedeuten, dass es bei der Verwendung eines unvollständigen Graphen Places geben kann, die nie wieder Tasks bekommen würden. Außerdem kann es durch das Entfernen von Knoten dazu kommen, dass der Lifelinegraph nicht mehr einer Hyperkubus-Struktur entspricht.

Aus den genannten Gründen wird nach dem Entfernen von Knoten aus dem Lifelinegraphen eine Reparatur dessen vorgenommen. Diese ist im Aktivitäts-Diagramm in Abbildung 3.5 in der Gruppe **Reparieren_des_Graphen** angegeben. Für jeden entfernten Knoten wird überprüft, ob sich dieser am Ende des Graphen befand. Denn dann muss der Graph nicht weiter repariert werden. Sollte sich der Knoten nicht am Ende des Graphen befunden haben, wird der letzte Knoten des Graphen an die Position des entfernten Knotens verschoben. Hier wird die Eigenschaft des Graphen, dass dieser sortiert ist, genutzt, um den letzten Knoten des Graphen schnell bestimmen zu können. Das Verschieben des letzten Knotens des Lifelinegraphen beinhaltet das Löschen aller seiner Kanten (Lifelines) und das Ändern dessen Knoten-ID auf die Knoten-ID des

entfernten Knotens (dies entspricht der Anpassung an seine neue Position im Lifelinegraphen, siehe Komponentendarstellung 2.3). Die Zuordnung zum ursprünglichen Place bleibt allerdings erhalten. Im Anschluss an die Reparatur des Graphen wird der Parameter Z und jeder Knoten aktualisiert. Dies entspricht der Aktualisierung aus Abbildung 3.4. Während der Aktualisierung werden nun die Kanten (Lifelines) der verschobenen Knoten neu berechnet.

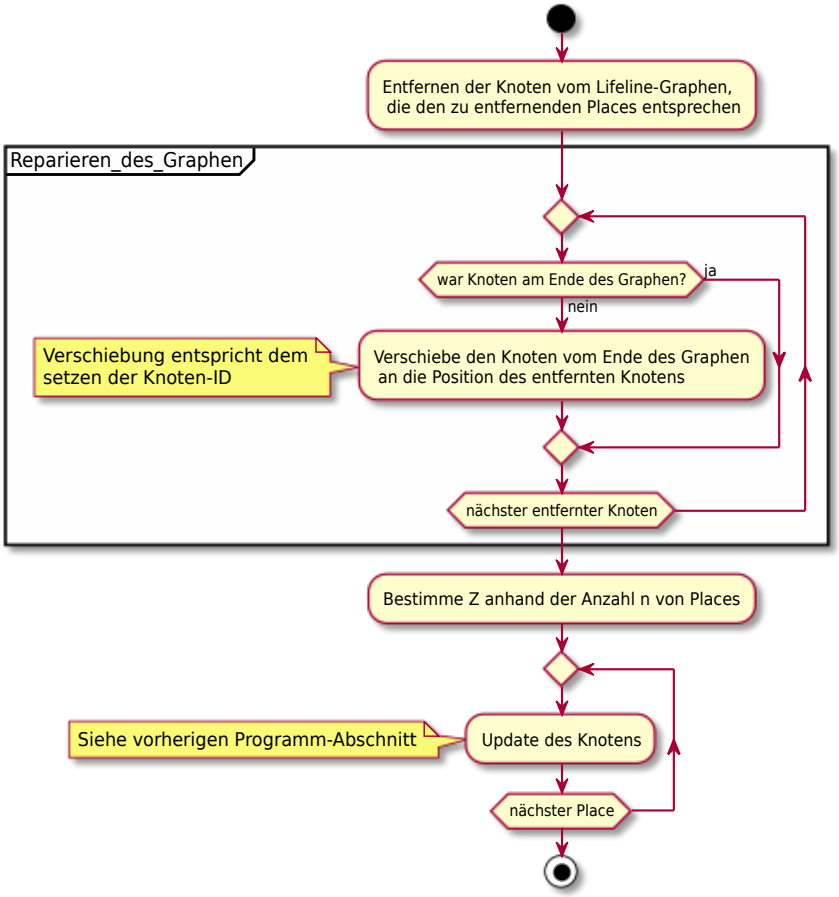

Abbildung 3.5: Entfernen eines oder mehrerer Places vom Lifelinegraphen

Die Reparatur des Lifelinegraphen ist in den folgenden Abbildungen anhand eines Beispiels dargestellt. Aus dem Lifelinegraphen wurde bereits der Knoten (mit der Knoten-ID 4) für den zu entfernenden Place (P4) entfernt, siehe Abbildung 3.6. Der Reparatur-Algorithmus sucht nun den letzten Knoten des Lifelinegraphen heraus. Der

letzte Knoten ist der Knoten mit der Knoten-ID 8. Dieser Knoten wird nun an die Stelle des entfernten Knotens verschoben und dessen Knoten-ID wird zu 4 geändert. Im Anschluss werden die Kanten aller Knoten im Lifelinegraphen aktualisiert. Das Ergebnis ist in Abbildung 3.7 dargestellt. Der Knoten des Places (P8) befindet sich nun an der Stelle, an der sich zuvor der Knoten des Places (P4) befand.

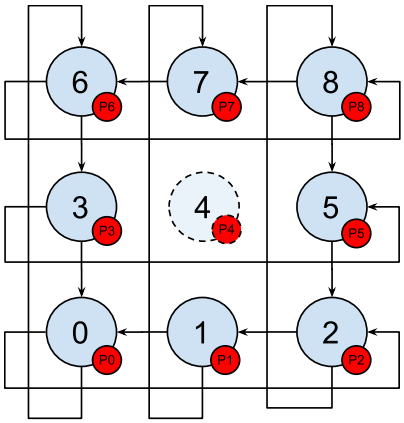

Abbildung 3.6: Lifelinegraph Place 4 entfernt für L=3, Z=2

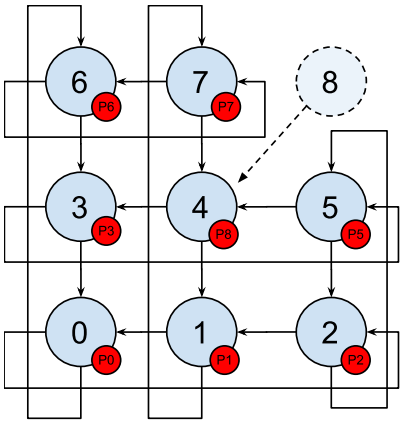

Abbildung 3.7: Lifelinegraph repariert für L=3, Z=2

4 Implementierung

Innerhalb dieses Kapitels wird die Implementierung der Referenzen im Detail erläutert, da diese den Kern der Ressourcen-Elastizität darstellen. Im Abschnitt 4.1, wird auf allgemeine Implementierungsaspekte eingegangen, die allen Referenzen zugrunde liegen. Diese bilden das notwendige Grundwissen über die technischen Aspekte von Referenzen, die für das weitere Verständnis dieses Kapitels notwendig sind. Im Anschluss an die allgemeinen Aspekte von Referenzen wird im Abschnitt 4.2 im Detail auf den Typ der globalen Referenzen eingegangen. Im Vordergrund steht dort vor allem die **Migration** globaler Referenzen. Anschließend wird im letzten Abschnitt 4.3 der Typ der lokalen Referenzen erläutert. Die Hauptaspekte dieses Abschnittes sind die **Migration** und die **Reduktion**.

4.1 Allgemeines

Die für die Ressourcen-Elastizität neu entwickelten Referenzen wurden bereits in Kapitel 3.3 eingeführt. Diese sind von der Klasse `GlobalRef` der APGAS-Bibliothek abgeleitet. Für die globalen und die lokalen Referenzen wurden vollständig neue Klassen implementiert, welche in den Folgenden Abschnitten vorgestellt werden.

Die neue Klasse `GlobalRef` (gleichnamig zu ihrer ursprünglichen Version) implementiert den Typ der globalen Referenzen. Die Abbildung 4.1 illustriert wie eine Instanz einer `GlobalRef` im partitionierten globalen Adressraum abgelegt wird. Die Abbildung zeigt 4 Places, mit jeweils einer CPU, 8 Threads (T1-T8) und den partitionierten gemeinsamen Speicher aller 4 Places. Die Instanz der `GlobalRef` liegt auf jedem Place in identischer Form vor (blauer Bereich der `GlobalRef`), sodass alle Places die gleiche Sicht auf die globale Referenz haben, so wie in Abbildung 4.1 dargestellt. Diese verweist auf exakt

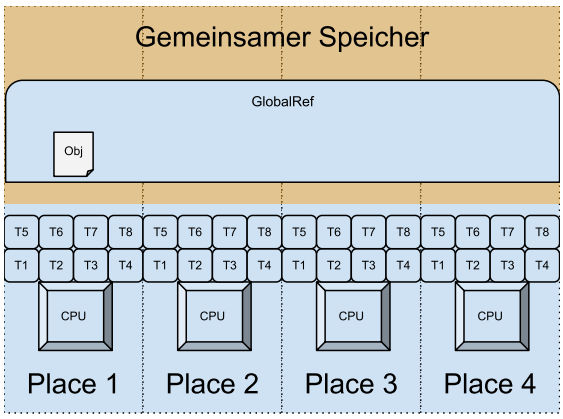

Abbildung 4.1: Speicher: GlobalRef

ein einzelnes Objekt, das den Wert der globalen Referenz abbildet. Dieses Objekt liegt dabei immer auf exakt einem Place, sodass Zugriffe darauf von anderen Places über das Netzwerk durchgeführt werden müssen.

Die lokalen Referenzen sind mit zwei neuen Klassen implementiert worden, der **PlacelocalRef** und der **ThreadlocalRef**. Beide Klassen hinterlegen Objekte auf jedem der Places, auf die über die gemeinsame Referenz zugegriffen werden kann. In Abbildung 4.2 wird die Abbildung einer `PlacelocalRef` Instanz auf den partitionierten globalen Adressraum dargestellt. Ähnlich zur `GlobalRef` existiert auf jedem Place eine identische `PlacelocalRef` Instanz, welche global als ein Objekt gesehen werden kann. Außerdem ist auf jedem der Places ein identisches Objekt abgelegt. Auf diese Objekte kann jedoch lediglich von ihrem besitzenden Place aus zugegriffen werden. Im Laufe der Programmausführung können nun lokalitätsflexible Tasks über die gleiche Referenz auf ähnliche aber Place-spezifische Objekte zugreifen und diese manipulieren. Dieser Zugriff über die Referenz ermöglicht eine einfache und allgemeine Syntax für Speicherzugriffe innerhalb von lokalitätsflexiblen Tasks, sodass immer auf die lokale Version des Wertes zugegriffen werden kann. Die zweite Implementierung der lokalen Referenzen ist die `ThreadlocalRef` Klasse. Diese ermöglicht es, nicht nur pro Place einen Wert zur Verfügung zu stellen, sondern für jeden Thread auf jedem Place. Dies ist exemplarisch für den partitionieren globalen Adressraum in Abbildung 4.3 dargestellt. Durch die Bereitstellung eines Wertes für jeden verfügbaren Thread, kann weitestgehend auf Synchronisierung innerhalb des Places beim Zugriff auf die `ThreadlocalRef`

verzichtet werden. Was zu einem enormen Performance-Vorteil führen kann, wenn sich die Berechnung darauf abbilden lässt.

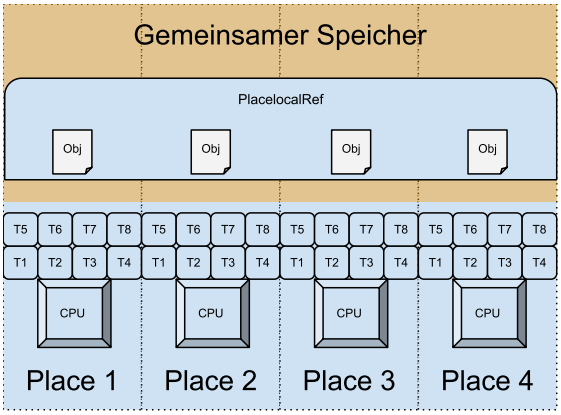

Abbildung 4.2: PlacelocalRef

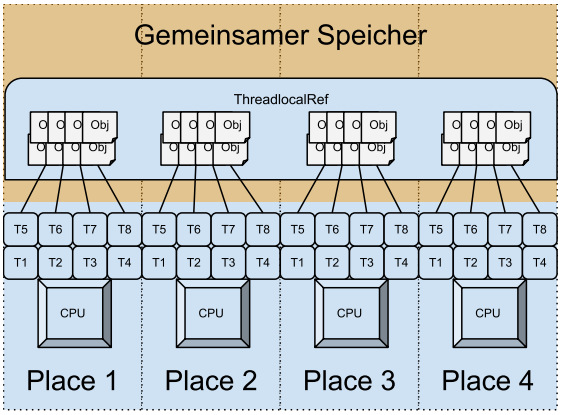

Abbildung 4.3: ThreadlocalRef

Allen Referenzen liegt das neu implementierte gemeinsame Interface **Ref** zugrunde, dass die grundlegenden Methoden jeder Referenz Implementierung festlegt. Dieses ist in Abbildung 4.4 abgebildet. Neben den gemeinsamen Methoden legt das Interface ebenfalls fest, dass die Interfaces **Serializable** und **AutoClosable** zu implementieren sind. `Serializable` ist notwendig, um die Referenzen samt ihrer Werte über das Netzwerk zu anderen Places schicken zu können. `AutoClosable` ermöglicht dem Programmierer einen einfacheren Umgang mit den Referenzen, in dem es Try-With-Ressources Statements von Java unterstützt. Einfacher wird der Umgang an dieser Stelle dadurch, dass Java das manuelle Freigeben der Referenzen durch die Verwendung des Try-With-Ressources Statements anstößt. Das manuelle Freigeben von Referenzen ist notwendig, da die Garbage-Collection von Java hier nicht eingesetzt werden kann. Das liegt wiederum daran, dass die Garbage-Collection kein Wissen über den Zustand der jeweiligen Referenzen auf anderen Places hat. Die folgende Auflistung beschreibt die grundlegenden Methoden der Referenzen im Einzelnen:

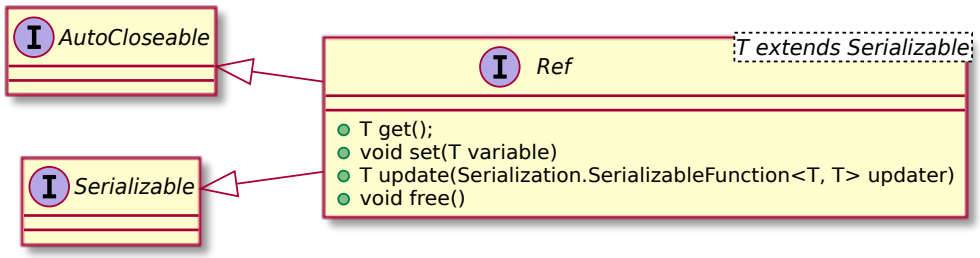

Abbildung 4.4: Das Interface Ref

- **get()**: Gibt den referenzierten Wert zurück.
- **set()**: Überschreibt den referenzierten Wert mit dem übergebenen Wert.
- **update()**: Führt eine Lambdafunktion aus, um den referenzierten Wert zu modifizieren. Gibt den neuen Wert zurück.
- **free()**: Gibt den referenzierten Wert zurück und baut die Referenz auf allen Places ab.

Die Methoden `get()`, `set()` und `free()` sind den gleichnamigen Implementierungen der ursprünglichen Klasse `GlobalRef` aus Kapitel 2 nachempfunden. Ihre Funktionsweise ist identisch, lediglich dessen Implementierung unterscheidet sich geringfügig. Die Methode `update()` ist neu hinzu gekommen und ermöglicht ein synchronisiertes modifizieren des referenzierten Wertes, sodass zwischen lesen und schreiben keine weiteren Zugriffe möglich sind. Diese Synchronisierung schließt damit Lost-Updates aus.

Jede Implementierung einer Referenz besitzt eine Instanz der Klasse **GlobalID**. Diese dient der eindeutigen Identifizierung einer Referenz im globalen gemeinsamen Adressraum. Die Klasse `GlobalID` wird durch die APGAS-Bibliothek bereitgestellt. Eine `GlobalID` besteht aus einer lokal generierten ID, der `lid` und der ID des erstellenden Places, der `placeId`. Die `placeId` wird bei der Instantiierung der `GlobalID` vom erstellenden Place gesetzt. Die *lid* wird ebenfalls während der Instantiierung gesetzt, jedoch durch eine statische atomare Zählervariable, die für jede `GlobalID`, die auf einem Place erstellt wird um eins erhöht wird. Die Werte `lid` und `placeId` identifizieren eine `GlobalID` global eindeutig. Die Klasse `GlobalID` wurde zusätzlich um

eine statische `ConcurrentHashMap` namens **REFERENCE_MAP** erweitert, in der festgelegt wird, wer der aktuelle Besitzer der `GlobalID` ist. Intitial ist dies immer der erstellende Place. Dies kann sich jedoch während einer **Migration** ändern, siehe Unterabschnitt 4.2.

In den folgenden zwei Unterabschnitten wird im Detail auf die Implementierungen der **Migration** und **Reduktion** für die gerade vorgestellten Implementierungen der Referenzen eingegangen.

4.2 Globale Referenzen

Die `GlobalRef` Klasse verwendet intern eine statische `ConcurrentHashMap`, im weiteren als **REFERENCE_MAP** bezeichnet. Die `REFERENCE_MAP` ordnet der `GlobalID` der `GlobalRef` Instanz den referenzierten Wert zu, wenn sich dieser auf dem aktuellen Place befindet. Befindet sich der Wert der `GlobalRef` auf einem anderen Place, enthält diese Map keinen Eintrag. Sämtliche Zugriffe auf einen Wert einer `GlobalRef` über `get()`, `set()` oder `update()` werden über die `ConcurrentHashMap` synchronisiert.

Globale Referenzen können auf andere Places migriert werden. Dazu wird der Besitzer der `GlobalID` der `GlobalRef` auf den neuen Place gesetzt und anschließend das referenzierte Objekt auf diesen Place übertragen. Die Migration wird mithilfe des Algorithmus aus Quellcode 4.1 durchgeführt. In Zeile 3 wird die Methode `compute()` dazu verwendet, die zu migrierende globale Referenz für alle anderen Zugriffe zu blockieren. Der Lambdafunktion werden die `GlobalID` als auch das referenzierte Objekt übergeben. Während der gesamten Ausführung der Lambdafunktion ist der Zugriff auf die globale Referenz blockiert. Innerhalb der Lambdafunktion wird nun der Wert der globeln Referenz zum neuen Ziel-Place kopiert. Dies wird mithilfe eines `uncountedAsyncAt`-Tasks durchgeführt, welcher über einen **GlobalCountDownLatch** synchronisiert wird, Zeilen 5-16. Ein `GlobalCountDownLatch` ist eine Hilfsklasse, welche identisch funktioniert, wie ein herkömmlicher **CountDownLatch**. Dieser wird mit einem CountDown von 1 initialisiert (Zeile 5), auf dem entfernten Place heruntergezählt (Zeile 11) und anschließend darauf gewartet, dass dieser den Wert 0 erreicht (Zeile 16). Auf dem Ziel-Place wird nun der Wert gespeichert (Zeilen 7-11) und die dortige `GlobalID` wird zum Ziel-Place migriert

(Zeile 8). Das Migrieren einer `GlobalID` zu einem anderen Place ändert den Besitzer dieser `GlobalID`. Wird z.B. eine `GlobalID` von Place A zum Place B migriert, dann ist Place B der neue Besitzer. Der Besitzer muss dazu auf allen Places für die `GlobalID` neu gesetzt werden (Zeilen 8,14,27). Anschließend wird der Wert der globalen Referenz vom ursprünglichen Place entfernt (Zeile 17). Das Zurückgeben eines `null` Wertes innerhalb eines `compute()` Aufrufes löscht dessen Eintrag aus der `REFERENCE_MAP`. Als letzter Schritt werden nun noch die übrigen Instanzen der globalen Referenz auf den anderen Places migriert. Dazu muss innerhalb dieser Instanzen lediglich die `GlobalID` migriert werden (Zeile 27). Dies geschieht ebenfalls über einen `uncountedAsyncAt`-Task, der über einen `GlobalCountDownLatch` synchronisiert ist (Zeilen 20-32).

```
1  public void migrate(final Place targetPlace) {
2
3    REFERENCE_MAP.compute(this.globalID, (globalID, storedValue) -> {
4
5      GlobalCountDownLatch countDownRef = new GlobalCountDownLatch(1);
6      uncountedAsyncAt(targetPlace, () -> {
7        GlobalRef.REFERENCE_MAP.compute(globalID, (storedID, oldValue) -> {
8          globalID.migrate(targetPlace);
9          return storedValue;
10       });
11       countDownRef.countDown();
12     });
13
14     globalID.migrate(targetPlace);
15     countDownRef.await();
16     countDownRef.free();
17     return null;
18   });
19
20   ArrayList<Place> otherPlaces = new ArrayList<>(places());
21   otherPlaces.remove(here());
22   otherPlaces.remove(targetPlace);
23
24   countDownRef = new GlobalCountDownLatch(1)
25   otherPlaces.forEach(otherPlace -> {
26     uncountedAsyncAt(otherPlace, () -> {
27       globalID.migrate(targetPlace);
28       countDownRef.countDown();
29     });
30   });
31   countDownRef.await();
32   countDownRef.free();
33 }
```

Quellcode 4.1: Algorithmus: Migration globaler Referenzen

Eine weitere Besonderheit bei der Migration von globalen Referenzen ist das Zusammenspiel mit den Methoden `get()`, `set()` und `update()`. Diese sind so implementiert, dass die Anfragen, die sich während einer Migration ansammeln können, nach der Migration an den neuen Ziel-Place weitergeleitet werden. Das Quellcode 4.2 illustriert diese Funk-

tionalität anhand der `get()` Methode. Sollte der aktuelle Place der Besitzer der `GlobalID` und damit auch der `GlobalRef` sein, wird der gespeicherte Wert zurückgegeben (Zeilen 3-5). Ist der Wert auf einem anderen Place gespeichert, wird ein `uncountedAsyncAt`-Task zu diesem Place geschickt, um den Wert von dort zu holen (Zeilen 7-17). Zum Kopieren des Wertes wird eine temporäre `GlobalRef` (Zeile 7) verwendet und für die Synchronisierung ein `GlobalCountDownLatch` (Zeile 8). Beide verwenden ein Try-With-Ressources Statement, um diese Instanzen wieder korrekt abzubauen, damit keine Speicherlecks entstehen. Die Weiterleitung selbst versteckt sich in Zeile 11, in der ein rekursiver Aufruf auf `get()` verwendet wird. Die Variable `TRANSFERED_REF` ist die globale Referenz von der der Wert abgerufen wird. Sobald diese mittels eines Tasks auf einen anderen Place geschickt wird, wird diese automatisch zur lokalen Instanz. Dies geschieht während der Serialisierung und Deserialisierung.

```
1  public T get() {
2
3    if (this.globalID.getPlace().equals(here())) {
4      return (T) REFERENCE_MAP.get(this.globalID);
5    }
6
7    try (GlobalRef<T> tempRef = new GlobalRef<>()) {
8      try (GlobalCountDownLatch countDownRef = new GlobalCountDownLatch(1)) {
9        final GlobalRef<T> TRANSFERED_REF = this;
10       uncountedAsyncAt(this.globalID.getPlace(), () -> {
11         T storedVariable = TRANSFERED_REF.get();
12         tempRef.set(storedVariable);
13         countDownRef.countDown();
14       });
15       countDownRef.await();
16     }
17   return tempRef.get();
18   }
19 }
```

Quellcode 4.2: Algorithmus: get() globaler Referenzen

4.3 Lokale Referenzen

Äquivalent zu den globalen Referenzen verwenden die lokalen Referenzen ebenfalls eine `REFERENCE_MAP` für das Speichern von Werten. Allerdings wird nicht der Wert direkt abgespeichert, sondern es wird die `PlacelocalRef` bzw. die `ThreadlocalRef` innerhalb der `REFERENCE_MAP` gespeichert. Dies ist notwendig, da eine lokale Referenz zusätzlich zu ihrem Wert ebenfalls die serialisierte Initialisierungsfunktion speichern muss. Die `ThreadlocalRef` ist im Grunde eine `PlacelocalRef`, welche eine Map anstelle eines Wertes speichert. Diese Map enthält alle Werte aller Threads und besitzt spezielle Zugriffsfunktionen. Da die `ThreadlocalRef` Implementierung auf der `PlacelocalRef` basiert, wird im Weiteren nur die `PlacelocalRef` betrachtet. Im Folgenden werden nun die Migration als auch die Reduktion der lokalen Referenzen erläutert.

Migration

Die Migration lokaler Referenzen verwendet eine zusätzliche `ConcurrentHashMap`, den **RESTORE_STORE**. Innerhalb dieser Map werden die `PlacelocalRefs` anderer Places gespeichert, welche migriert wurden. Die Map speichert dazu die Abbildung der `GlobalID` der `PlacelocalRefs` auf eine Liste von `PlacelocalRefs`. Dies ist notwendig, da ein Place mehrere `PlacelocalRefs` anderer Places erhalten kann, die zu der gleichen `GlobalID` gehören, sprich der gleichen `PlacelocalRef` entsprechen.

Da die Implementierung der Migration von `PlacelocalRefs` technisch sehr komplex ist, wird der Algorithmus im Folgenden lediglich mithilfe von Pseudocode illustriert, siehe Quellcode 4.3. Die Migration von `PlacelocalRefs` wird immer direkt für alle `PlacelocalRefs` des aktuellen Places durchgeführt. Die gleichzeitige Migration aller `PlacelocalRefs` ist deutlich effizienter als einzelne Migrationen. Außerdem müssen beim Entfernen eines Places auch immer alle `PlacelocalRefs` migirert werden. Aus diesem Grund werden in den Zeilen 2-3 sämtliche `PlacelocalRefs` des Places kopiert. Dies schließt die `PlacelocalRefs` anderer Places, die bereits zu diesem Place migriert wurden, mit ein (Zeile 3). Nachdem alle `PlacelocalRefs` kopiert wurden, werden die lokalen Einträge der Maps alle gelöscht (Zeilen 4-5). Anschließend werden die kopierten

PlacelocalRefs zum neuen Ziel-Place transferiert (Zeilen 6-17). Dies geschieht unter der Verwendung eines uncountedAsyncAt-Tasks (Zeile 7) und eines GlobalCountDownLatch (Zeile 6). Auf dem Ziel-Place werden nun die kopierten PlacelocalRefs nach ihrer GlobalID gruppiert (Zeile 8) und anschließend dem lokalen RESTORE_STORE hinzugefügt. Dazu wird die merge() Funktion der ConcurrentHashMap verwendet, die die Listen von PlacelocalRefs anhand ihrer GlobalIDs zusammenführt (Zeile 9-13). Ist bereits eine Liste für eine GlobalID im RESTORE_STORE enthalten, wird die neue Liste von PlacelocalRefs vom anderen Place dieser Liste hinzugefügt. Ist noch keine Liste im RESTORE_STORE enthalten, wird die neue Liste vom anderen Place dem RESTORE_STORE hinzugefügt.

```
1  public static void migrate(final Place targetPlace) {
2    final List<PlacelocalRef> placelocals = copy(REFERENCE_MAP);
3    RESTORE_STORE.values().forEach(placelocals::addAll);
4    REFERENCE_MAP.clear();
5    RESTORE_STORE.clear();
6    try (GlobalCountDownLatch countDownRef = new GlobalCountDownLatch(1)) {
7      uncountedAsyncAt(targetPlace, () -> {
8        Map<GlobalID, List<PlacelocalRef>> groupedPlaceLocals = placelocals.
             groupingBy(PlacelocalRef::getGlobalID);
9        groupedPlaceLocals.forEach(
10         (globalID, placelocals) -> RESTORE_STORE.merge(globalID, 
               placelocals, (placelocals1, placelocals2) -> {
11         placelocals1.addAll(placelocals2);
12         return placelocals1;
13       }));
14       countDownRef.countDown();
15     });
16     countDownRef.await();
17   }
18 }
```

Quellcode 4.3: Pseudo-Algorithmus: Migration von PlacelocalRefs

Reduktion

Die Reduktion lokaler Referenzen kombiniert deren verteilte Werte von allen Places zu einem einzelnen Wert. Dies ermöglicht zum Beispiel das Berechnen von Endergebnissen aus einer Menge von Zwischenergebnissen. Aufgrund der bereits erläuterten Migration von lokalen Referenzen muss die Reduktion neben den Werten in den `REFERENCE_MAP`s jedes Places auch die Werte der `RESTORE_STORE`s berücksichtigen.

Die Implementierung der Reduktion ist ebenfalls wie die Implementierung der Migration technisch sehr komplex, deshalb wird auch hier auf eine Beschreibung mittels Pseudocode zurückgegriffen, siehe Quellcode 4.4. Die Reduktion ist syntaktisch angelehnt an die Funktion `reduce()` der Java Stream API, was dessen Verwendung intuitiver machen sollte. Als Parameter wird der `reduce()` Funktion ein initialer Wert als auch eine `Accumulator` Funktion übergeben (Zeile 1). Ein initialer Wert hilft dabei `NullPointerException`s zu vermeiden und garantiert einen Rückgabewert, der nicht `null` ist. Die `Accumulator` Funktion muss vom Programmierer angegeben werden und definiert wie zwei Einzelwerte miteinander zu kombinieren sind. Das globale Endergebnis wird mithilfe einer temporären `GlobalRef` gebildet (Zeile 3). Zur Kommunikation mit den anderen Places wird hier ebenfalls eine Kombination von `uncountedAsyncAt`-Tasks und eine Instanz der Klasse `GlobalCountDownLatch` verwendet. Allerdings wird hier der Countdown auf die Anzahl der Places gesetzt (Zeile 4). Auf jedem der Places wird nun der Code ausgeführt, um die dortigen lokalen Werte des `RESTORE_STORE` und der `PlacelocalRef` zu einem lokalen Gesamtergebnis zu kombinieren (Zeilen 8-25). Dazu werden im ersten Schritt alle Werte des `RESTORE_STORE`, die zur aktuellen `PlacelocalRef` gehören über deren `GlobalID` abgerufen (Zeilen 8-9). Anschließend werden diese, falls vorhanden, mithilfe der Java Stream API Methode `reduce()` zu einem Wert kombiniert (Zeilen 10-14). In den Zeilen 16-18 wird nun der lokale Wert der `PlacelocalRef` bestimmt. Falls dieser nicht existiert, wird der initiale Wert verwendet. Nachfolgend wird nun in Abhängigkeit davon, ob ein Ergebnis für den `RESTORE_STORE` existiert, die Kombination aus dem lokalen Wert der `PlacelocalRef` und dem Ergebnis des `RESTORE_STORE`s gebildet (Zeilen 19-20). Das lokale Endergebnis der Reduktion, entweder der lokale Wert oder das Ergebnis aus lokalem Wert und Ergebnis des `RESTORE\_STORE`s, werden anschließend dem globalen Gesamtergebnis hinzugefügt (ent-

weder Zeilen 22-24 oder Zeilen 19-22). Schlussendlich kann das globale Gesamtergebnis der Reduktion über die temporäre `GlobalRef` abgerufen und zurückgegeben werden (Zeile 29).

```
1  public T reduce(T initialValue, Accumulator<T> accumulator) {
2
3    try (GlobalRef<T> result = new GlobalRef<>(initialValue)) {
4      try (GlobalCountDownLatch countDownRef = new GlobalCountDownLatch(
           places().size())) {
5        final PlacelocalRef<T> TRANSFERED_REFERENCE = this;
6        places().forEach(place -> uncountedAsyncAt(place, () -> {
7
8          Optional<List<PlacelocalRef>> restoredValues =
9            getRestoredValues(TRANSFERED_REFERENCE.getGlobalID());
10         T restoredResult = restoredValues.map(
11           placelocals -> placelocals
12             .stream()
13             .reduce(initialValue, accumulator)
14         ).orElse(null);
15
16         final T localResult = TRANSFERED_REFERENCE.get() == null ?
17           initialValue :
18           TRANSFERED_REFERENCE.get();
19         if (restoredResult != null) {
20           final T endResult = accumulator.apply(localResult, restoredResult
               );
21           result.update(globalResult -> accumulator.apply(globalResult,
               endResult));
22         } else {
23           result.update(globalResult -> accumulator.apply(globalResult,
               localResult));
24         }
25         countDownRef.countDown();
26       }
27     }
28   }
29   return result.get();
30 }
```

Quellcode 4.4: Pseudo-Algorithmus: Reduktion von PlacelocalRefs

5 Experimente

Die Ressourcen-Elastizitäts-Prozesse haben einen Einfluss auf die Performance der ausgeführten Programme. Das Aktualisieren des Lifelinegraphen auf allen Places und das Hinzufügen und Entfernen von Places zur Laufzeit benötigen Prozessorzeit und verursachen erhöhten Netzwerkverkehr. Zur Ermittlung der Performance der Implementierung wurden die beiden Benchmarks UTS [16] und BC [17] verwendet. Dieses Kapitel erläutert zunächst die beiden Benchmarks in ihrer Funktionsweise und bezüglich der notwendigen Anpassungen zur Verwendung der Ressourcen-Elastizität. Dieser Abschnitt beschreibt außerdem die verwendete Experimentierumgebung, die zur Ausführung der Benchmarks verwendet wurde. Anschließend werden im Abschnitt 5.2 die Laufzeiten der beiden Benchmarks präsentiert und die spezifischen Zeiten, die die einzelnen Hinzufüge- und Entfernungs-Operationen innerhalb der Benchmarks benötigt haben präsentiert und diskutiert.

5.1 Experimentierumgebung

Die Benchmarks wurden im hochschuleigenen Cluster ausgeführt. Die dabei verwendete Hardware besteht aus Doppelprozessorsystemen mit je Zwei 16-Kern Opteron Prozessoren aus dem Jahr 2012. Diese verfügen über einen Arbeitsspeicher von 32GB bis 128GB. Die Konfiguration von 32GB war bereits ausreichend für die Ausführung der Benchmarks. Die einzelnen Knoten sind über 1GBit/s Ethernet miteinander verbunden. Das verwendete Betriebssystem war ein CentOS 7, das über ein global verfügbares GPFS Dateisystem verfügte.

Der Benchmark Unbalanced Tree Search (UTS) berechnet das Problem, die Anzahl an Knoten in einem implizit konstruierten Baum zu zählen. Diese Bäume können sich in

Form, Tiefe, Größe und Balance untereinander unterscheiden. Die Balance gibt dabei an, wie stark sich die Größe der Unterbäume eines Baumes unterscheiden. Die starken Größenunterschiede der Unterbäume machen eine einfache Aufteilung der Traversierung des Baumes in einzelne Tasks nicht möglich. Diese müssen viel mehr dynamisch und rekursiv aufgeteilt werden. Aus diesem Grund eignet sich dieser Benchmark besonders gut für die Bewertung der Ressourcen-Elastizität. Denn zu der möglichst gleichmäßigen Verteilung der Arbeit, muss ebenfalls garantiert werden, dass während Ressourcen-Elastizitäts-Prozessen die Arbeit korrekt umverteilt wird. Die verwendete Variante des UTS Benchmarks verwendet Bäume mit einem einstellbaren Parameter **d**, der die maximale Tiefe des Baumes angibt.

Der Betweeness Centrality (BC) Benchmark berechnet einen Centrality Messwert für jeden Knoten innerhalb eines Graphen. Der Benchmark wird mit einem Parameter **n** parametrisiert, der angibt, wie viele Knoten der Graph enthalten soll. Der Centrality Messwert bestimmt, wie häufig sich dieser Knoten auf kürzesten Pfaden zwischen zwei anderen Knoten befindet. Die Bestimmung des Messwertes setzt voraus, dass die kürzesten Pfade eines jeden Knoten zu jedem anderen Knoten berechnet werden. Die Verwendete Benchmarkimplementierung berechnet innerhalb eines Tasks alle kürzesten Pfade von einem bestimmten Knoten zu jedem anderen Knoten und zählt anschließend für jeden Knoten zwischen Start- und Endknoten auf den berechneten kürzesten Pfaden den zugehörigen Centrality Messwert hoch. Für jeden Knoten des Graphen wird ein solcher Tasks erstellt. Aus diesem Grund muss der zugrunde liegende Graph auf jedem der Places vorhanden sein. Am Ende der Berechnung existieren auf jedem der Places eine Menge von Ergebnissen. Jeder Place besitzt die Centrality Messwerte der kürzesten Pfade von allen Knoten, die auf diesem Place berechnet wurden. Das Gesamtergebnis der Centrality Messwerte über alle kürzesten Pfade des Graphen kann anschließend über eine Summenbildung aller Ergebnisse aller Places bestimmt werden. Die Menge der Ergebnisse pro Place als auch die Notwendigkeit, auf jedem Place den Graphen vorliegen zu haben bieten eine gute Ausgangslage zur Evaluation der **Migration** und der **Reduktion**. Denn während der Ressourcen-Elastizitäts-Prozesse müssen größere Datenmengen zwischen den Places transferiert werden. Außerdem kommt in diesem Benchmark die *transiente* Version der **PlacelocalRef** zum Einsatz, sodass der Graph, auf den nur lesend zugegriffen wird, beim Entfernen von Places nicht migriert wird.

Die Verwendung der transienten `PlacelocalRef` spart zusätzlichen Netzwerkverkehr ein.

Beide Benchmarks wurden um einen weiteren Parameter ergänzt. Dieser dient dazu automatische Ressourcen-Elastizitäts-Prozesse während der Ausführung der Benchmarks zu simulieren. Dazu wurde ein festes Zeitintervall von 10 Sekunden definiert, in dem immer wieder Hinzufüge- und Entfernungs-Operationen durchgeführt werden. Es wird immer erst eine spezifische Anzahl an Places hinzugefügt und exakt nach der Hälfte des Zeitintervalls werden diese wieder entfernt. Die Anzahl der Places, die hinzugefügt und wieder entfernt werden, können mit dem Parameter **m** angegeben werden. Auf diese Weise werden in regelmäßigen Abständen neue Places hinzugefügt und wieder entfernt. Das Entfernen wählt dabei immer zufällige Places aus, die entfernt werden sollen, mit Ausnahme des Master Places.

5.2 Evaluation der Ergebnisse

Beide Benchmarks wurden auf dem bereits beschriebenen Cluster in unterschiedlichsten Konfigurationen ausgeführt, um die Laufzeiten der Benchmarks und die Zeiten der Hinzufüge- und Entfernungs-Operationen zu messen. Die Benchmarks wurden für verschiedene Parametrisierungen von **m** und die benchmarkspezifischen Parameter **d** für UTS und **n** für BC durchgeführt. In der nachfolgenden Betrachtung der Ergebnisse wurden diese auf **d=17** und **n=17** eingeschränkt. Andere Parametrisierungen von **d** und **n** führen zwar zu absolut gesehen höheren bzw. niedrigeren Laufzeiten, jedoch bleiben die Zeiten der Hinzufüge- und Entfernungs-Operationen und auch deren relativer Einfluss auf die Laufzeit der Benchmarks nahezu identisch. Die Menge der zu verwendenden Threads pro Knoten wurde auf 32 Threads und damit auf die Anzahl der verfügbaren Prozessoren pro Knoten festgelegt. Diese Festlegung sorgt für eine volle Auslastung auf jedem Knoten, sodass der Einfluss der Ressourcen-Elastizitäts-Prozesse sichtbar wird und nicht durch die Verwendung eines ungenutzten Prozessors verschleiert wird. Nachfolgend werden nun beide Benchmarks mit unterschiedlichen Werten für die Parameter **m** und der Anzahl von Places **p** betrachtet und diskutiert.

In Abbildung 5.1 sind die Laufzeiten von UTS mit verschiedenen Parametrisierungen für **m** und **p** in einem Diagramm aufgetragen. Die Laufzeiten entsprechen dem Mittelwert von insgesamt 10 Durchläufen für jede der Parametrisierungen. Die niedrigste Laufzeit ergibt sich für **m=0**, also den Durchläufen, in denen gar keine Ressourcen-Elastizität stattfindet. Da alle anderen Durchläufe, in denen eine Ressourcen-Elastizität stattfindet eine höhere Laufzeit besitzen und die Laufzeiten für höhere Werte des Parameters **m** ebenfalls immer ansteigen, lässt sich ein definitiver Overhead durch die Ressourcen-Elastizität feststellen. Dieser wird besonders deutlich, wenn die Laufzeiten für **p=6** betrachtet werden. Betrachtet man hingegen die Laufzeiten gegen **p=16** Places, nähern sich diese sehr stark an die Laufzeit ohne Ressourcen-Elastizität an. Hierbei sei angemerkt, dass die Skala des Diagramms bei 40 beginnt und nicht bei 0. Dies kann dazu führen, dass der Abstand der Laufzeiten größer erscheint, als dieser tatsächlich ist.

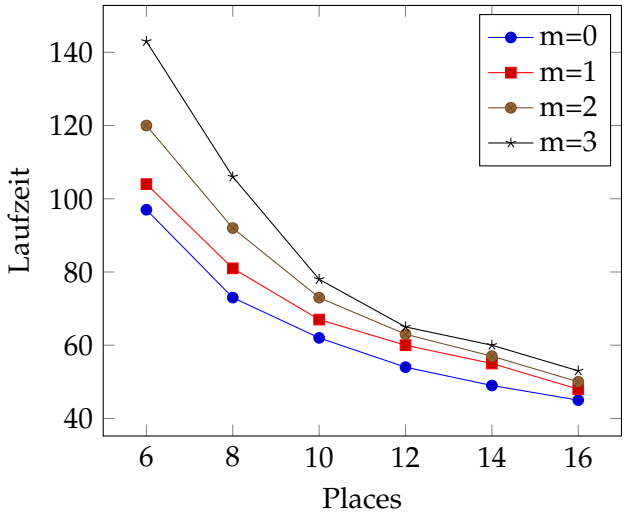

Abbildung 5.1: UTS Ergebnisse für d=17, m=1-3 und p=6-16

Analysiert man nun die Ausführung der Benchmarks im Bezug auf die Simulation der Ressourcen-Elastizität, fällt auf, dass die Places, die hinzugefügt und wieder entfernt werden, einen Teil der Laufzeit nicht zur Verfügung stehen. Denn die Simulation lässt die Places lediglich in der ersten Hälfte des Zeitintervalls aktiv und entfernt diese anschließend für die Dauer der zweiten Hälfte des Zeitintervalls. Unter der Annahme, dass die Verfügbarkeit der Places **m** aus diesen Gründen nur zu 50% besteht, ließen sich

nun die Laufzeiten entsprechend korrigieren, siehe dazu Abbildung 5.2. Die Gleichung zur Korrektur der Laufzeit lässt sich wie folgt definieren:

$$L_{neu} = L_{alt} - L_{alt} \cdot \frac{m}{2 \cdot p}$$

Diese Formel lässt sich herleiten, indem man von der Gesamtlaufzeit denjenigen Teil der Laufzeit abzieht, der den 50% der `m` Places an der Gesamtlaufzeit entspricht. Diese korrigierte Laufzeit ist allerdings nur als Annäherung zu sehen, denn diese vernachlässigt einige Einflüsse auf die Laufzeit des Benchmarks. Hier wäre z.B. das Implementierungsdetail anzuführen, dass Places, welche entfernt werden, den aktuell ausgeführten Task noch zu Ende berechnen dürfen. Betrachtet man nun die korrigierten Laufzeiten, fällt schnell auf, dass die Laufzeiten der Durchläufe mit Ressourcen-Elastizität nun sehr nah an der Laufzeit ohne Ressourcen-Elastizität liegen. Der Overhead scheint also eher gering zu sein. Der geringe Overhead ließe sich gut durch die Implementierung der Ressourcen-Elastizität erklären. Denn diese wurde so konzipiert, dass sie weder die Berechnungen noch die Lastenbalancierung der betroffenen Places oder anderer Places beeinflusst. Außerdem sind das Hinzufügen und das Entfernen von Places so implementiert, dass diese parallel für jeden der betroffenen Places ausgeführt werden können.

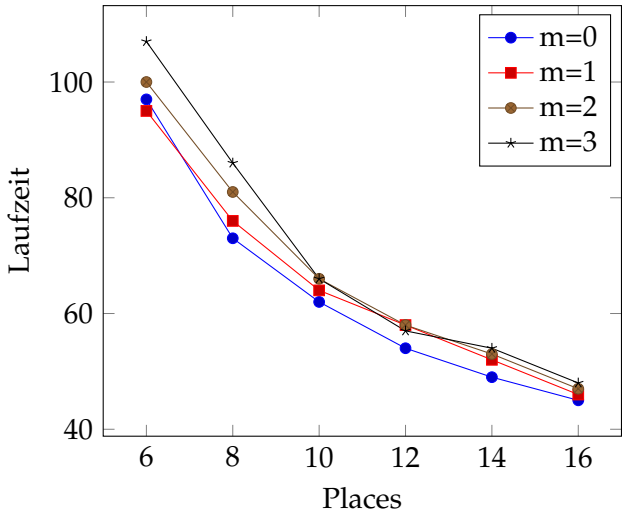

Abbildung 5.2: UTS Ergebnisse bereinigt für d=17, m=1-3 und p=6-16

Der Benchmark BC liefert ähnliche Laufzeiten wie der Benchmark UTS, siehe dazu Abbildung 5.3. Hier fällt allerdings auf, dass die Laufzeiten stärker in Abhängigkeit zum Parameter **m** steigen. Dies geht ebenfalls aus den korrigierten Laufzeiten aus Abbildung 5.4 hervor. Ursache dieser stärkeren Abhängigkeit ist in der deutlich höheren Datenmenge zu vermuten, die beim Hinzufügen und dem Entfernen von Places übertragen werden muss.

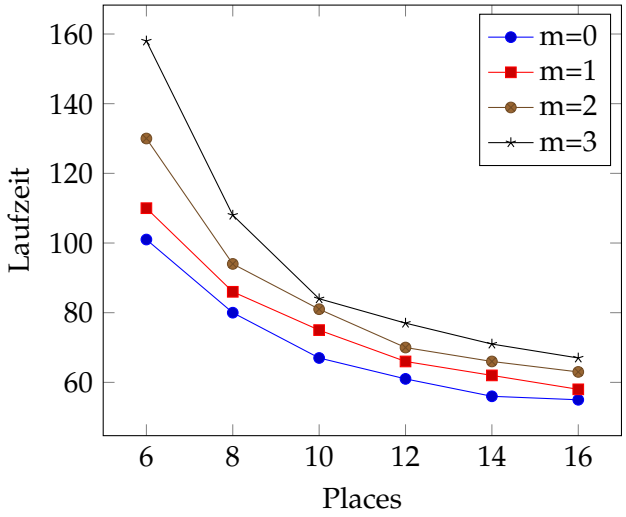

Abbildung 5.3: BC Ergebnisse für n=17, m=1-3, p=6-16

Die nachfolgende Tablle 5.1 zeigt die durchschnittlich benötigte Zeit für Hinzufüge- (Add) und Entfernungs-Operationen (Remove) des Benchmarks UTS. Diese wurden pro Durchlauf eines Benchmarks gemittelt. Die Zeit zum Hinzufügen eines Places beinhaltet das Starten der JVM, das Hochfahren von Hazelcast, das Initialisieren der APGAS-Laufzeitumgebung und die notwendigen Schritte des Ressourcen-Elastizitäts-Prozesses. Für einen neuen Place werden im Schnitt 5 Sekunden benötigt. Jeder zusätzliche Place benötigt etwa 2 Sekunden mehr. Die benötigte Zeit des Hinzufügens eines Places teilt sich auf, in das Hochfahren des Places (JVM, Hazelcast und Laufzeitumgebung starten) und dem Initialisieren des Places. Das Hochfahren der Places wird komplett asynchron ausgeführt, sodass sich die Places gegenseitig nicht beeinflussen. Die Initialisierung der Places hingegen wird allein vom Master-Place vorgenommen, weshalb dieser Anteil an der benötigten Zeit zum Hinzufügen von Places mit jedem weiteren Place steigt. Dies legt

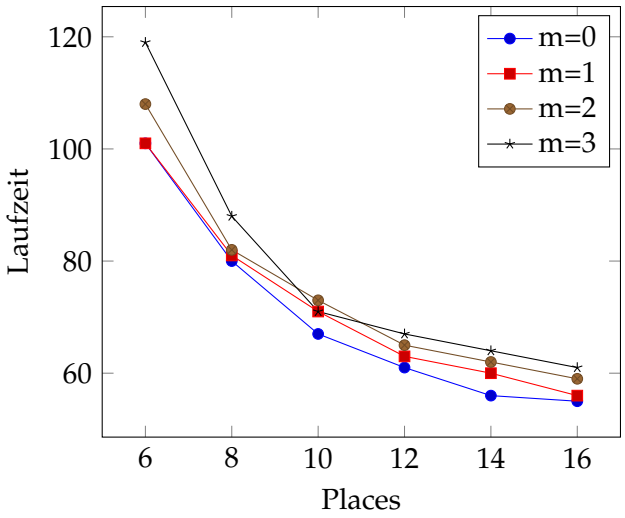

Abbildung 5.4: BC Ergebnisse bereinigt für n=17, m=1-3, p=6-16

eine Schätzung von ca 2 Sekunden pro Initialisierung eines Places und ca 3 Sekunden für das Starten einer beliebigen Menge an neuen Places nahe.

Das Entfernen von Places hingegen scheint unabhängig von der Anzahl an Places zu sein, da dessen Zeit für die Entfernungs-Operation bis auf die Ausnahme von 3,11 Sekunden für 2 Places immer ca 1-1,5 Sekunden beträgt. Diese Ausnahme wiederum lässt sich durch den hohen Ausreißer von 11,43 Sekunden einer einzelnen Operation erklären. Dieser Ausreißer selbst ist vermutlich auf das Warten eines aktiven Tasks zurückzuführen, der besonders lang gedauert hat. Dass die Zeit zum Entfernen von Places nicht von der Anzahl zu entfernender Places abhängt ist durch den asynchronen Prozess des Entfernens zu erklären. Denn beim Entfernen werden die notwendigen Schritte auf den zu entfernenden Places ausgeführt, mit Ausnahme der Aktualisierung des Lifelinegraphen, das vom Master-Place durchgeführt wird.

Abschließend zeigt die Tabelle 5.2 die durchschnittlichen Zeiten der Ressourcen-Elastizität für den Benchmark BC. Auch für den Benchmark BC zeigt sich beim Hinzufügen von Places eine Abhängigkeit zur Menge der hinzugefügten Places. Beim Entfernen hingegen scheint dies ebenfalls unabhängig von der Anzahl der entfernten Places zu sein. Allerdings fallen beim Benchmark BC die höheren Zeiten auf, die für die Operationen gebraucht wurden. Beim Hinzufügen zwischen 5,72-10,25 Sekunden und beim Entfernen

Operation	m (in Places)	Zeit (in Sekunden)	Minimum Zeit	Maximum Zeit
Add	1	5,21	5,19	5,24
Add	2	7,36	7,20	7,59
Add	3	8,98	7,18	9,66
Remove	1	1,22	0,91	1,41
Remove	2	3,11	0,91	11,43
Remove	3	1,53	0,93	2,28

Tabelle 5.1: UTS Ressourcen-Elastizität Zeiten

3,73-4,95 Sekunden. Dies lässt sich durch die deutlich höhere Menge an Daten erklären, die der Benchmark BC benötigt.

Operation	m (in Places)	Zeit (in Sekunden)	Minimum Zeit	Maximum Zeit
Add	1	5,72	5,64	5,83
Add	2	7,73	6,56	8,54
Add	3	10,25	9,26	11,06
Remove	1	3,73	0,90	6,34
Remove	2	5,09	4,48	5,68
Remove	3	4,95	0,93	8,93

Tabelle 5.2: BC Ressourcen-Elastizität Zeiten

Bei der Betrachtung der Minimum- als auch Maximum- Zeiten in beiden Tabellen fällt zusätzlich auf, dass diese sehr stark beim Entfernen von Places vom Mittelwert abweichen. Aus der Konzeption der Ressourcen-Elastizität geht hervor, dass weder Berechnungen noch die Lastenbalancierung von der Ressourcen-Elastizität beeinflusst werden sollen. Daher ist der Grund für die Schwankungen an anderer Stelle zu vermuten. Der hohe Ausreißer von 11,43 Sekunden in der Tabelle 5.1 weißt daraufhin, dass diese Schwankungen auf das Warten noch aktiver Tasks eines Places zurückzuführen ist.

6 Verwandte Arbeiten

Es existiert ein Interesse im Bereich des HPC an malleable Jobs zur Erhöhung der Auslastung eines HPC Clusters. Einige der bisherigen Ansätze, um die Ausführung von Programmen mit malleable Jobs kompatibel zu machen werden im Folgenden vorgestellt.

In [7] wird Job-Folding verwendet, um moldable Jobs sich wie malleable Jobs verhalten zu lassen. Das Job-Folding sorgt dafür, dass pro Prozessor mehrere Prozesse gestartet werden. Sobald mehr Prozessoren verfügbar sind, werden die überschüssigen Prozesse auf die zusätzlichen Prozessoren verteilt. Der Ansatz des Job-Folding funktioniert allerdings nur auf Supercomputern und nicht auf HPC Clustern, so wie der Ansatz dieser Arbeit. Ein ähnlicher Ansatz wird in [5] verfolgt, bei dem ein rigid Job mit sogenannten best effort Jobs ergänzt wird. Die best effort Jobs helfen dem rigid Job bei der Berechnung, wenn zusätzliche Ressourcen verfügbar sind. Sollten Ressourcen wegfallen, werden die best effort Jobs wieder beendet.

Die PCM-Bibliothek (Process Checkpointing and Migration) [6] [18] [19] stellt zusätzliche Funktionen für iterative MPI Programme zur Verfügung, um diese Ressourcen-elastisch zu machen. Das iterative MPI Programm wird dazu in regelmäßigen Abständen unterbrochen (Checkpointing) und bei sich veränderten Ressourcen durch Verschieben von Daten oder dem Starten und Beenden von Prozessen angepasst (Migration). Die Ressourcen-Elastizität der APGAS-Bibliothek benötigt im Gegensatz zu PCM keine Unterbrechungen des Programms.

Das Dynaco Framework [9] ermöglicht Ressourcen-Elastizität von Programmen in einem HPC Cluster. Die zu verwendende Bibliothek zur Parallelisierung wird nicht vorgegeben. Allerdings muss vom Programmierer angegeben werden, wann Anpassungen notwendig sind, welche Probleme bei einer Anpassungen auftreten können und wie die

Anpassungen durchzuführen sind. Die Ressourcen-Elastizität der APGAS-Bibliothek nimmt dem Programmierer diese Aufgaben ab.

In [20] wird eine API präsentiert, die den Workload-Manager Slurm in ein Ressourcenelastisches Programm integriert. Die API informiert das Programm über Änderungen in den verfügbaren Ressourcen, woraufhin sich das Programm anpassen kann. Dies entspricht exakt dem entgegengesetzten Part dieser Arbeit, die sich ausschließlich mit der Anpassung des Programms selbst an sich ändernde Ressourcen beschäftigt. Die vorgestellte API in [20] liegt als C++ Bibliothek vor, weshalb sich diese nicht ohne Weiteres mit der in dieser Arbeit entwickelten Ressourcen-Elastizität kombinieren lässt.

Das Charm++ Framework [8] verwendet zur Abstraktion von Daten- und Arbeitseinheiten sogenannte Chares. Chares sind Objekte, die wie Klassen definiert werden können und vom Charm++ Framework verwaltet und auf die verfügbaren Ressourcen verteilt werden. Die Ressourcen-Elastizität wird vom Charm++ Framework durch ein umverteilen der Chares bei Veränderungen der Ressourcen erreicht, ähnlich zur Ressourcen-Elastizität bei APGAS. Allerdings ist aufgrund der Verwendung von Chares in Charm++ im Unterschied zu den Tasks bei APGAS eine Unterbrechung der gesamten Ausführung des Programms erforderlich. Diese Unterbrechung ist notwendig, da die Chares, die umverteilt werden für die Berechnung notwendig, zum Zeitpunkt der Umverteilung jedoch nicht erreichbar sind.

7 Zusammenfassung

In dieser Arbeit wurde ein Konzept zur Anpassung eines Programms während seiner Ausführung an sich ändernde Ressourcen innerhalb eines HPC Clusters vorgestellt. Die Anpassung eines Programms an sich ändernde Ressourcen während seiner Ausführung wurde innerhalb dieser Arbeit als Ressourcen-Elastizität bezeichnet, um diese von *malleable Jobs* besser zu differenzieren. Das Konzept der Ressourcen-Elastizität wurde auf Basis der APGAS-Bibliothek entwickelt und baut auf dessen APGAS-Programmiermodell, seiner globalen Lastenbalancierung und dessen Lifelinegraphen auf. Das APGAS-Programmiermodell mit seinem Task-basierten Berechnungsmodell bietet eine sehr gute Grundlage für das Verschieben von Berechnungen von einer Ressource zu einer anderen. Darüber hinaus sorgt die globale Lastenbalancierung in Verbindung mit dem Lifelinegraphen für eine hohe Auslastung der Verfügbaren Ressourcen, auch bei Anpassungen an sich ändernde Ressourcen.

Neben dem umverteilen von Berechnungen, mussten ebenfalls die auf den Ressourcen befindlichen Daten berücksichtigt werden. Dazu wurde das APGAS-Modell um globale und lokale Referenzen erweitert, die eine Weiterentwicklung der in der APGAS-Bibliothek verwendeten `GlobalRef` darstellen. Die globalen und lokalen Referenzen ermöglichen durch die beiden neuen Konzepte der Migration und der Reduktion ein Verschieben von Daten von einer Ressource zu einer anderen. Die Migration erlaubt es Daten von einer Ressource auf eine andere zu Verschieben und die Reduktion kann im Anschluss an eine Berechnung verwendet werden um alle auf den Ressourcen verteilten Daten zusammenzufassen. Diese Zusammenfassung schließt dabei die in der Migration verschobenen Daten mit ein. Vervollständigt wurde das Konzept der Ressourcen-Elastizität mit der Anpassung des Lifelinegraphen an sich ändernde Ressourcen, durch die Möglichkeit zur Laufzeit weitere Knoten dem Graphen hinzuzufügen oder zu entfernen.

Die entwickelten Konzepte wurden mithilfe der beiden Benchmarks UTS und BC auf dem universitätseigenem Cluster evaluiert und die Ergebnisse in dieser Arbeit präsentiert. Die Ergebnisse haben gezeigt, dass der Overhead, der durch die Anpassungen des Programms an sich verändernde Ressourcen entsteht, relativ gering ist. Des Weiteren konnte festgestellt werden, dass der Overhead bei zunehmender Anzahl an Ressourcen, die dem Programm zur Verfügung stehen, relativ gesehen immer geringer wird. Dieser Vorteil folgt aus dem Design des Konzeptes der Ressourcen-Elastizität, das zum Ziel hatte, so wenig Ressourcen wie möglich in die Prozesse der Ressourcen-Elastizität mit einzubeziehen. Aus diesem Grund wird beim Hinzufügen neuer Ressourcen fast ausschließlich der Master zusätzlich belastet, indem dieser die neuen Ressourcen initialisiert. Während des Entfernens von Ressourcen werden lediglich die zu entfernenden Ressourcen und zum geringen Anteil der Master-Place belastet. Auf diese Weise bleiben alle anderen Ressourcen nahezu unbelastet und können weiterhin die Berechnungen ausführen.

Die Ergebnisse haben ebenfalls gezeigt, dass die Laufzeit des Hinzufügens neuer Ressourcen linear mit der Anzahl der hinzu zu fügender Ressourcen steigt. Der linear ansteigende Teil der Laufzeit ergibt sich aus der benötigten Zeit, die der Master benötigt um die neuen Ressourcen zu initialisieren. Für hohe Zahlen von Ressourcen, kann dies ein Problem werden, sodass zukünftig das Konzept an dieser Stelle angepasst werden könnte, um die Initialisierung neuer Ressourcen auf mehrere Ressourcen zu verteilen. Die Verteilung der Initialisierung hätte den Vorteil, dass diese parallel ausgeführt wird und die neuen Ressourcen damit schneller zur Verfügung stehen würden um Berechnungen auszuführen. Der Berechnungsaufwand würde im Vergleich zur Initialisierung allein durch den Master gleich bleiben. Jedoch könnte der Ressourcenengpass durch die einzelne Netzwerkschnittstelle des Masters negiert werden.

Literaturverzeichnis

[1] D. G. Feitelson and L. Rudolph, "Toward convergence in job schedulers for parallel supercomputers," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–26.

[2] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, and et al., "Theory and practice in parallel job scheduling," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 1–34.

[3] S. Guo and L. Kang, "Online scheduling of malleable parallel jobs with setup times on two identical machines," *European Journal of Operational Research*, vol. 206, no. 3, pp. 555 – 561, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S037722171000175X

[4] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Languages and Compilers for Parallel Computing*, L. Rauchwerger, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322.

[5] M. C. Cera, Y. Georgiou, O. Richard, and et al., "Supporting malleability in parallel architectures with dynamic CPUSETsMapping and dynamic MPI," in *Distributed Computing and Networking*, K. Kant, S. V. Pemmaraju, K. M. Sivalingam, and J. Wu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 242–257.

[6] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and et al., "Dynamic malleability in iterative MPI applications," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, 2007, pp. 591–598.

[7] G. Utrera, J. Corbalan, and J. Labarta, "Implementing malleability on MPI jobs," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 2004, pp. 215–224.

[8] B. Acun, A. Gupta, N. Jain, and et al., "Parallel programming with migratable objects: Charm++ in practice," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 647–658.

[9] J. Buisson, O. Sonmez, H. Mohamed, and et al., "Scheduling malleable applications in multicluster systems," in *2007 IEEE International Conference on Cluster Computing*, 2007, pp. 372–381.

[10] P. Charles, C. Grothoff, V. Saraswat, and et al., "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, p. 519–538, Oct. 2005. [Online]. Available: https://doi.org/10.1145/1103845.1094852

[11] O. Tardieu, B. Herta, D. Cunningham, and et al., "X10 and APGAS at petascale," *SIGPLAN Not.*, vol. 49, no. 8, p. 53–66, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2692916.2555245

[12] O. Tardieu, "The APGAS library: Resilient parallel and distributed programming in Java 8," in *Proceedings of the ACM SIGPLAN Workshop on X10*, ser. X10 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 25–26. [Online]. Available: https://doi.org/10.1145/2771774.2771780

[13] V. Kumar, Y. Zheng, V. Cavé, and et al., "HabaneroUPC++: A compiler-free PGAS library," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2676870.2676879

[14] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, 2000, pp. 36–43.

[15] V. A. Saraswat, P. Kambadur, S. Kodali, and et al., "Lifeline-based global load balancing," *SIGPLAN Not.*, vol. 46, no. 8, p. 201–212, Feb. 2011. [Online]. Available: https://doi.org/10.1145/2038037.1941582

[16] S. Olivier, J. Huan, J. Liu, and et al., "UTS: An unbalanced tree search Benchmark," in *Languages and Compilers for Parallel Computing*, G. Almási, C. Caşcaval, and P. Wu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 235–250.

[17] L. Freeman, "A set of measures of centrality based on Betweenness," *Sociometry*, vol. 40, pp. 35–41, 03 1977.

[18] K. El Maghraoui, T. J. Desell, B. K. Szymanski, and et al., "Malleable iterative MPI applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 3, pp. 393–413, 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1362

[19] T. Desell, K. Maghraoui, and C. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, vol. 10, pp. 323–337, 09 2007.

[20] S. Iserte, R. Mayo, E. S. Quintana-Ortí, and et al., "DMR API: Improving cluster productivity by turning applications into malleable," *Parallel Computing*, vol. 78, pp. 54 – 66, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118302229