UNIKASSEL
VERSITÄT

Fachbereich 16 Elektrotechnik/Informatik
Studiengang Informatik

Übertragung eines fehlertoleranten Algorithmus für Fork/Join-Programme auf reduktionsbasierte Taskpools

Masterarbeit

Maximilian Dratwa
Matrikelnummer: 31213073

Erstprüfer: Prof. Dr. Claudia Fohry
Zweitprüfer: Prof. Dr. Gerd Stumme

Kassel, den 11.12.2017

Selbständigkeitserklärung

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

Kassel, den 11.12.2017

Maximilian Dratwa

Inhaltsverzeichnis

1 Einleitung

Parallelverarbeitung wird häufig eingesetzt, um die Ausführung von Programmen zu beschleunigen. Ein Problem wird dabei in mehrere Teilaufgaben zerlegt, die auch Tasks genannt werden. Diese Tasks werden auf die verfügbaren Recheneinheiten aufgeteilt. Neben Mehrkern-Prozessoren können hierfür mehrere Prozessoren in einem System oder mehrere Computer eines Clusters benutzt werden. Verteilt man die Last effizient auf die verfügbare Hardware, wird durch das gleichzeitige Bearbeiten von Tasks die Laufzeit eines Programmes erheblich verringert.

Es gibt zwei Architekturen der Parallelverarbeitung. Zum einen gibt es Systeme mit einem gemeinsamen Speicher, wie es bei Mehrkernprozessor-Systemen der Fall ist. Die Prozessoren haben Zugriff auf einen gemeinsamen Speicher, wodurch der Austausch von Daten sehr einfach ist. Diese Architektur kommt zum Beispiel bei der OpenMP Programmierschnittstelle zum Einsatz. Eine andere Möglichkeit ist ein geteilter Speicher, d.h. die Prozessoren haben jeweils einen eigenen Speicher. Die Kommunikation untereinander erfolgt durch den Austausch von Nachrichten. Mithilfe des Message Passing Interface (kurz: MPI) kann diese Architektur programmiert werden.

Partitioned Global Address Space (kurz: PGAS) ist ein Programmiermodell, welches die beiden Architekturen miteinander verknüpft und versucht die Komplexität für den Programmierer zu verringern. Die Rechenknoten heißen Places und besitzen einen eigenen Speicher, sowie einen oder mehrere Prozessoren. Ein Place kann auch auf den Speicher anderer Places zugreifen, jedoch sind lokale Zugriffe schneller. Die Programmiersprache X10 basiert auf dem asynchronen PGAS Modell. Tasks heißen dort Activities und können asynchron auf einem vom Programmierer festgelegten Place ausgeführt werden. Die Funktionen der Programmiersprache wurden in einem Framework namens APGAS für Java bereitgestellt, um eine größere Anzahl von Entwicklern zu erreichen. Dies erleichtert die Programmierung paralleler Programme, jedoch bleiben weitere Herausforderungen wie eine effiziente Lastenverteilung und Fehlertoleranz bestehen.

Das Framework Global Load Balancing (kurz: GLB) für APGAS ermöglicht eine effiziente Verteilung von Tasks. Jeder Place besitzt einen Worker, der einen Taskpool besitzt und sich um die Kommunikation mit den anderen Workern kümmert. Jeder Worker arbeitet seinen eigenen Taskpool ab und stiehlt Tasks von anderen Workern, sobald sein eigener Taskpool leer ist. Das Ergebnis wird am Ende per Reduktion aus allen Teilergebnissen berechnet. Dieses Framework sorgt für eine gute Performance und Skalierbarkeit.

Ein Problem bei der Parallelverarbeitung ist die höhere Wahrscheinlichkeit eines Hardwaredefekts. Damit ist der Ausfall einer Hardwarekomponente oder die Nichterreichbarkeit eines Clusterknotens gemeint. Besitzt ein Programm keine Fehlerbehandlung, kann es zu einem Programmabsturz führen oder ein falsches Ergebnis liefern. Das GLB Framework besitzt eine fehlertolerante Variante, die in bestimmten Abständen Backups der Taskpools macht und im Fall eines Fehlers auf die Backups zurückgreift, um von dort aus weiter zu rechnen.

In der Referenz [KKM17] wird ein anderes Konzept der Fehlertoleranz beschrieben. Im Gegensatz zur fehlertoleranten GLB Variante bezieht es sich jedoch auf verteilte Fork/Join Programme. Mithilfe von Datenstrukturen die einen Teil des Stehlbaumes abbilden, kann man bei einem Ausfall eines oder mehrerer Knoten, die verloren gegangenen Tasks identifizieren. Die sich in Bearbeitung befindenden Teilaufgaben können gesichert werden und müssen bei erneuter Ausführung der fehlerhaften Tasks nicht erneut berechnet werden. Noch laufende Berechnungen gehen somit nicht verloren.

Diese Thesis beschreibt die Übertragung des fehlertoleranten Konzepts für Fork/Join Programme auf das GLB Framework für APGAS. Eine Herausforderung dabei war es, die Anpassung auf reduktionsbasierte Taskpools durchzuführen, welche eine andere Herangehensweise bei der Weitergabe und Verarbeitung der Ergebnisse haben. Fork/Join Programme stehlen sich eine Task und besitzen eine Verbindung zum Elternknoten, die für die Ergebnisrückgabe benötigt wird. Bei GLB wird die Hälfte des Taskpools gestohlen und es gibt keine Elternknoten, da das Ergebnis am Ende aller Berechnungen per Reduktion zusammengefasst wird. Ein Ziel war es möglichst wenige Änderungen an der Schnittstelle des Frameworks vorzunehmen, um Anpassungen von Benchmarks und Programmen gering zu halten.

Auf dem Cluster der Universität Kassel wurden Experimente durchgeführt, um die Effizienz im fehlerfreien Fall zu messen. Zum Vergleich wurde eine Variante ohne Fehlertoleranz

und die Variante mit regelmäßigen Backups benutzt. Als Benchmark kam Unbalanced Tree Search (kurz: UTS) zum Einsatz, welche bereits für das GLB Framework implementiert war. Bei einer kleineren Anzahl von Tasks zeigten sich keine spürbaren Auswirkungen in der Laufzeit im Vergleich zur Variante ohne Fehlertoleranz. Bei einer hohen Anzahl an Tasks steigt der Overhead mit der Anzahl der Places. Bis auf eine Ausnahme bleibt er immer niedriger als bei der Backup Variante.

In Kapitel 2 der Thesis werden die benötigten Grundlagen zu APGAS und dem GLB Framework erläutert. Danach wird in Kapitel 3 Fork/Join und dessen fehlertoleranter Algorithmus erklärt. Anschließend wird in Kapitel 4 das Konzept der Übertragung von dem Algorithmus in GLB vorgestellt und erforderliche Änderungen hervorgehoben. Danach folgt die Beschreibung der Implementierung in Kapitel 5. In Kapitel 6 wird der Benchmark UTS erläutert und die Messungen miteinander verglichen. Zum Schluss folgt eine Zusammenfassung der Thesis und ein Ausblick auf mögliche Verbesserungen, die in der Zukunft implementiert werden können.

2 Grundlagen

In diesem Kapitel wird zuerst die Programmiersprache X10 vorgestellt, die das asynchrone PGAS Modell implementiert. Anschließend folgt das APGAS Framework für Java, welches die Funktionen aus X10 übernimmt. Die Struktur des GLB Framework wird erläutert und der interne Ablauf beschrieben. Am Ende des Kapitels wird eine bereits implementierte fehlertolerante Version von GLB vorgestellt, welche in Kapitel 6 mittels Benchmark mit dem in dieser Arbeit entwickelten Algorithmus verglichen wird.

2.1 Programmiersprache X10

IBM Research entwickelt seit dem Jahr 2004 die objektorientierte Programmiersprache X10. Die Sprache ist klassenbasiert, stark typisiert und besitzt eine automatische Speicherbereinigung (Garbage Collector). Der Name X10 steht für das Ziel, eine um den Faktor 10 höhere Effizienz eines Programmierers bei dem Einsatz für Parallelverarbeitung zu ermöglichen.[X10]

Die Programmiersprache basiert auf dem Asynchronous Partitioned Global Address Space Modell. Ein globaler Adressraum wird für die Rechenknoten, die Places genannt werden, partitioniert. Jeder Place besitzt seinen eigenen Speicher, jedoch kann er auf den gesamten Speicherbereich zugreifen. Der Programmier legt fest, wie die Daten verteilt werden. Dabei muss beachtet werden, dass die Zugriffszeit auf lokalen Speicher deutlich geringer ist. Neben dem Speicher besitzt ein Place auch Prozessoren. Sie führen Threads aus, die in diesem Modell Activites genannten Tasks bearbeiten. Eine Activity kann zur Laufzeit dynamisch erzeugt und ausgeführt werden. Auch hierbei muss vom Programmierer der auszuführende Place festgelegt werden.

2.2 APGAS Framework

Das APGAS Framework wird ebenfalls von IBM Research entwickelt und ist Teil des X10 Projektes [Tar15]. Es überträgt das Programmiermodell und die Hauptfunktionen der Programmiersprache X10 in Java. Der Grund für die Entwicklung war eine größere Anzahl von Programmieren einen leichteren Einstieg zu ermöglichen. APGAS ist Open Source und die Version 1.0 wurde im Juni 2015 veröffentlicht. Die Version wird als Bibliothek bereitgestellt. Voraussetzung für die Nutzung ist Java Version 8, da Lambdafunktionen zum Einsatz kommen.

Für die Entwicklungsumgebung Eclipse gibt es ein Plugin namens APGAS Development Tools, das APGAS und weitere Hilfen, wie die Anzeige von Compilerwarnungen, beinhaltet. Die aktuellste Version von APGAS ist im offiziellen Git Repository zu finden. Diese Thesis verwendet den Stand vom 15. Dezember 2015, der einige Fehler im Zusammenhang mit Clustern behebt.

APGAS verwendet einige Java Bibliotheken, die ebenfalls Open Source sind. Das Framework Hazelcast stellt ein In-Memory Data Grid bereit [HAZ]. APGAS verwendet es für die Kommunikation zwischen den Java Virtual Machines (kurz: JVM) und dem verteilten Speicher.

Im nachfolgenden Text werden die wichtigsten Konstrukte von APGAS beschrieben.

Place

Ein Place stellt, wie bereits erwähnt, eine Recheneinheit dar. Die Anzahl der Places wird durch die JVM Option `-Dapgas.places` festgelegt, die beim Programmstart übergeben wird. Mit dem Keyword `at` kann man den Place wechseln. In einer Lambdafunktion wird der auszuführende Code beschrieben und zusammen mit den verwendeten Daten übermittelt. Alle Objekte, die zwischen den Places übertragen werden, müssen serialisierbar sein. Während in X10 bereits alle Objekte serialisierbar sind, muss in Java das Interface `Java.IO.Serializable` für die jeweilige Klasse implementiert werden.

Activity

Ein Place kann mehrere Activities besitzen, die wie Tasks sind, und sie auf Threads verteilen. Sie führen einen Code Block aus. Ein Programm startet mit der Root Activity, welche die Main-Methode ausführt. Wenn diese Activity beendet wird, dann terminiert das gesamte Programm. Standardmäßig sind die Anzahl der Threads und die Zahl der verfügbaren Prozessorkerne identisch. Es ist möglich, mit der JVM Option `-Dapgas.threads` eine benutzerdefinierte Anzahl einzustellen. Zur Verwaltung werden weitere Threads von Java und APGAS benötigt, die nicht zur angegebenen Gesamtanzahl zählen.

async / finish

Mit dem Keyword `async` kann man einen Code Block als Activity ausführen. Ist ein Thread auf dem angegebenen Place im Leerlauf, wird der Block direkt ausgeführt. Ansonsten wartet die Activity auf den Scheduler, der sie an einen Thread übergibt.

Mit `finish` kann man eine Barriere für Activities setzen, die auf die Fertigstellung aller Acitvities innerhalb eines Blocks wartet. Erst nach Beendigung der letzten Activity wird das Programm fortgesetzt.

Fehlerbehandlung

Die Fehlerbehandlung kann mit der Option `-Dapgas.resilient=true` aktiviert werden. Dadurch wird das Programm nicht beendet, falls ein Place abstürzt. Es führt nur zu einer `DeadPlacesException` . Jeder Place kann einen Handler registrieren, der im Fall eines Absturzes ausgeführt wird und als Parameter den abgestürzten Place besitzt. Dort kann die Fehlerbehandlung gestartet werden. Die `DeadPlaceException` wird ausgelöst, falls man auf einen bereits abgestürzten Place zugreifen möchte.

Nebenläufigkeit

APGAS stellt keine Konstrukte zur Nebenläufigkeit bereit. Stattdessen können die von Java bekannten Keywords benutzt werden. Kritische Abschnitte werden mit `synchronized` abgesichert. Datentypen wie AtomicBoolean und ConcurrentLinkedQueue sind threadsicher und können ebenfalls verwendet werden.

2.3 GLB Framework

Das Global Load Balancing Framework benutzt APGAS als Grundlage. Es basiert auf dem Lifeline Graph Work-Stealing Algorithmus[SKK+11]. Es dient zum effizienten Verteilen von Tasks auf die Places. Diese Arbeit übernimmt ein sogenannter Worker, welcher auf jedem Place läuft. Er besitzt einen Taskpool, den er abarbeitet. Wird ein Task bearbeitet, wird er aus dem Taskpool gelöscht. Aus dem Task können weitere Tasks entstehen, die ans Ende des Pools angehängt werden.

Tritt der Fall ein, dass ein Worker einen leeren Taskpool besitzt, so wird eine Stehlanfrage an einen anderen Worker gesendet, um einen Teil dessen Taskpools zu übernehmen. Die Auswahl des Opfers erfolgt zuerst per Zufall. Werden nur leere Taskpools gefunden, dann werden Lifelines des Workers nach Tasks befragt. Die Lifelines sind Kanten in einem k-dimensionalen Graphen zwischen den Workern, der beim Programmstart erzeugt wird. Der Worker wird inaktiv, bis er von einer Lifeline wieder aufgeweckt wird.

Dieses Vorgehen ist besonders bei Algorithmen mit einer unbekannten Anzahl von Tasks sinnvoll, um eine gleichmäßige Verteilung der Rechenlast zu erreichen. Das Framework besitzt jedoch eine Einschränkung. Es lassen sich nur Algorithmen einbinden, die ohne Seiteneffekte und Kommunikation zwischen Tasks auskommen. Das Gesamtergebnis muss per Reduktion aus den Ergebnissen der Tasks berechenbar sein. Die Reduktionsfunktion muss kommutativ und assoziativ sein, damit bei unterschiedlichen Reihenfolge der Bearbeitung von Tasks das Ergebnis nicht verändert wird.

Neben dem Worker gibt es einige Klassen, die für den jeweiligen Algorithmus implementiert werden müssen.

Worker

Im Konstruktor werden die Datenstrukturen und der Taskpool initialisiert, sowie die Lifelines berechnet.

Die Hauptfunktion des Workers steckt in der `processStack()` Methode. Solange der Worker als aktiv markiert ist, durchläuft er eine Schleife. Zuerst wird `process(int)` aufgerufen, um n Tasks aus dem Taskpool zu bearbeiten. Ist der eigene Taskpool leer, werden in der Funktion `steal()` Stehlanfragen gesendet. Ist keine Stehlanfrage erfolgreich, bleibt der Worker solange inaktiv, bis er von einer Lifeline aufgeweckt wird oder die Berechnung abgeschlossen ist. Sind jedoch Tasks vorhanden, dann können mit der `distribute()` Funktion Stehlanfragen beantwortet werden. Dabei wird jeweils die Hälfte des Taskpools abgetrennt.

Um Speicherplatz zu sparen, fasst jeder Worker nach der Bearbeitung eines Tasks das Ergebnis zusammen. Am Ende besitzt jeder Place sein Teilergebnis. Place 0 führt eine Reduktion aus, um das Gesamtergebnis zu ermitteln.

Taskbag

Der Taskbag wird zum Übertragen der Tasks zwischen den Workern benutzt. Mit der Funktion `size()` wird die Anzahl der gespeicherten Tasks zurückgegeben. Mit `merge()` wird der übergebene Taskbag an das Ende des Taskbags gehangen.

TaskQueue

Die TaskQueue speichert den eigenen Taskpool. Hier werden Funktionen zum Teilen und Zusammenfügen von Taskpools sowie zur Berechnung von Tasks implementiert und der Reduktionsoperator angegeben.

Die Funktion `process(int):boolean` bearbeitet bis zu n Tasks, die aus dem Taskpool entfernt werden. Teilaufgaben werden an das Ende des Taskpools angehängt. Der Rückgabewert der Funktion gibt an, ob sich noch Tasks im Taskpool befinden.

In der Methode `split():TaskBag` wird definiert, wie der Taskpool aufgeteilt wird und liefert Tasks in einem Taskbag zurück, der an einen anderen Worker gesendet werden kann.

Die Methode `merge(TaskBag)` fügt den übergebenen TaskBag in den Taskpool ein.

Das reduzierte Ergebnis der eigenen TaskQueue liefert die Funktion `getResult()`. Mit `mergeResult(TaskQueue)` kann das Ergebnis von einer anderen TaskQueue reduziert werden. Reduziert man das Ergebnis aller TaskQueue in eine, so erhält man das Gesamtergebnis.

2.4 Fehlertolerante GLB Variante

Das vorgestellte GLB Framework besitzt keine Fehlerbehandlung. Die Referenz [PF17] beschreibt eine fehlertolerante Variante des GLB Frameworks. Jeder Place erstellt in regelmäßigen Abständen Backups der eigenen TaskQueue. Bei der positiven Beantwortung einer Stehlanfrage wird ebenfalls ein Backup erstellt.

Bei dem Versenden eines TaskBags wird dieser zusätzlich in eine Map Datenstruktur geschrieben. Im Falle eines Fehlers bei der Übernahme in die TaskQueue des Diebes kann er dort entnommen werden. Bei erfolgreicher Übertragung wird der TaskBag wieder aus der Map gelöscht. Der TaskBag hat eine LootID, die hochgezählt wird, wenn sich der Taskpool durch eine Stehlanfrage teilt. Die Map und Taskpool Backups können auf mehrere Places gespiegelt werden. Bei einem Ausfall wird das Backup von einem anderen Place geladen und nicht übertragene TaskBags aus der Map übernommen.

Die Implementierung der Backups und der Map Datenstruktur erfolgt mit dem Hazelcast Framework, welches bereits vom APGAS Framework bekannt ist. Mit dem `IMap<K,V>` Interface kann man eine Map Datenstruktur erzeugen, die eine Option besitzt, auf wie viele JVMs sie gespiegelt werden soll. Bei der Initialisierung der Places werden zwei IMaps erzeugt.

`iMapBackup<Integer, TaskQueue>` dient zur Speicherung der Taskpool Backups. Jeder Place kann mit seiner Place Nummer seine TaskQueue eintragen. Jeder Worker hat nun ein Zähler, wie oft die `process(n)` Funktion auf der TaskQueue aufgerufen wurde. Übersteigt der Zähler einen angegebenen Wert, so wird die Methode `writeBackup` ausgeführt und

der Zähler zurückgesetzt. Hierbei wird die Queue in die iMap geschrieben und auf mehrere Places gespiegelt.

Die Map `OpenLootMap<Integer, <Integer, <Pair<Long, TaskBag>>` speichert für jeden Place eine Map, in der TaskBags vor der Übertragung zu einem Dieb gespeichert werden. Die Place Nummer des Diebes ist der Schlüssel und dazu wird ein Paar aus einer LootID und einem TaskBag gespeichert.

Da vor dem Beendigen der Arbeit ein Backup geschrieben wird, sind alle TaskQueues in iMapBackup geschrieben. Die Funktion `CollectResults` lädt alle Ergebnisse der TaskQueues aus der Map und reduziert sie zu einem Ergebnis. Sollte in dieser Phase ein Worker ausfallen, so hat es keinen Einfluss auf die Reduktion und der Ausfall kann ignoriert werden.

Im Fall eines Fehlers beginnt die Fehlerbehandlung, die durch Ausführen eines Handlers erfolgt. APGAS startet auf jedem Place den Handler, sobald es einen Ausfall eines Places bemerkt. Dabei wird die Nummer des abgestürzten Places übergeben. Die Fehlerbehandlung funktioniert nur, solange Place 0 nicht abstürzt. Unbeantwortete Stehlanfragen von dem abgestürzten Place werden gelöscht. Besitzt man die nächstniedrigere Place Nummer, gilt man als Backup Place für den abgestürzten Worker. Als Backup Place darf das Backup des Places geladen und in die TaskQueue übernommen werden. In der `OpenlootMap` wird überprüft, ob an den abgestürzten Place gesendete TaskBags vorhanden sind. Sind die Tasks des TaskBags nicht im Backup der TaskQueue gespeichert, werden sie wieder eingefügt. Ist alles übernommen, wird ein Backup von dem Backup Place hochgeladen und die Informationen vom abgestürzten Place aus den Maps gelöscht. War der Backup Place vor der Fehlerbehandlung inaktiv, so wird er wieder reaktiviert. Am Ende der Methode wird solange gewartet, bis die Fehlerbehandlung aller Places bearbeitet wurde. Anschließend werden die Berechnungen auf allen verfügbaren Places fortgesetzt.

3 Lokale Fehlerbehebung für Fork/Join-Programme

Im vorherigen Kapitel wurde ein Algorithmus mit reduktionsbasierten Taskpools und eine Möglichkeit der Fehlerbehandlung vorgestellt. Dieses Kapitel behandelt eine lokale Fehlerbehebung für Fork/Join Programme mit Work-Stealing, welche in „Localized Fault Recovery for Nested Fork-Join Programms“[KKM17] beschrieben wird.

Den Ablauf eines verteilten Fork/Join Programmes kann man als Berechnungsbaum darstellen. In diesem Baum sind Tasks die Knoten und durch den fork Befehl werden neue Knoten erzeugt, die eine Kante zum Elterntask erhalten. Geht ein Task durch einen Fehler verloren, so ist der gesamte Teilbaum des Tasks betroffen, da die Verbindung zum Berechnungsbaum verloren geht. Eine lokale Fehlerbehebung speichert in jedem Knoten ein Backup seiner Kindtasks. Bemerkt ein Knoten einen Ausfall eines Kindes, wird der Task aus dem Backup erneut ausgeführt und als neuer Knoten angehangen. Dieser Knoten sucht nach allen Kindknoten des verlorenen Tasks, um deren Teilbäume wieder mit dem Berechnungsbaum zu verbinden. Dadurch wird die Anzahl der wiederauszuführenden Tasks minimiert, da nicht die gesamte Arbeit des Teilbaumes erneut ausgeführt werden muss. Andere Tasks des Programmes sind von der Fehlerbehandlung nicht betroffen und werden weiterausgeführt.

Die Lösung beinhaltet ein fehlertolerantes Kommunikationsprotokoll für Work-Stealing mit verteiltem Speicher. Die Behandlung von mehrfachen Abstürzen sowie von Fehlern während der Fehlerbehandlung werden ebenfalls berücksichtigt.

Im ersten Abschnitt dieses Kapitels wird erklärt, was ein Fork/Join Programm ist und welche Fehlerquellen es bei der Ausführung eines solchen Programmes gibt. Dann wird eine Erweiterung des Fork/Join Programmes für verteilte Speicher mit Work-Stealing

vorgestellt. In Abschnitt 3.3 wird die Fehlertoleranz für solche Programme erläutert. Abschnitt 3.5 zeigt an einem Beispiel den Ablauf des Algorithmus.

3.1 Fork/Join

Ein Fork/Join Programm führt einen Task aus und durch den `fork` Befehl wird ein neuer Task erzeugt, der eine Teilaufgabe erledigt. Die Teilaufgabe kann also parallel ausgeführt werden. Um die Ergebnisse von mehreren `fork` Aufrufen zusammenzuführen, wird ein `join` Befehl ausgeführt. Dieser stellt eine Barriere da, die auf die Terminierung der Tasks wartet. Anschließend wird das Hauptprogramm fortgesetzt und die Ergebnisse der Tasks können verwendet werden.

Anhand der rekursiven Berechnung der Fibonacci Zahlen in Quelltext 3.1 soll die Funktionsweise eines Fork/Join Programmes verdeutlicht werden. Die Fibonacci Zahlen sind definiert durch `FIB(n) = n` für `n < 2` und ansonsten durch `FIB(n) = FIB(n-1) + FIB(n-2)`.

```
1 function FIB(n)
2   if n < 2
3     RETURN n
4   a = fork FIB(n - 1)
5   b = fork FIB(n - 2)
6   join
7   RETURN a + b
```

Quelltext 3.1: Fork/Join Fibonacci Programm

Die Funktion `FIB` gibt den Wert `n` zurück, falls `n` kleiner als 2 ist. Ansonsten wird der Fork/Join Teil aufgerufen. Die Abarbeitung erfolgt nach dem depth-first Prinzip, d.h. bei einem `fork` wird zuerst der neu erstellte Task bearbeitet und die Fortsetzung des alten Tasks kann von einem anderen Thread abgearbeitet.

Um `a` zu berechnen, wird mithilfe des `fork` Befehls der Funktionsaufruf von `FIB(n-1)` in einem neuem Task ausgeführt. Gleichzeitig kann ein anderer Thread den alten Task fortsetzen und berechnet den Wert von `b` mit einem `fork` von `FIB(n-2)`. Der Thread, der den `join` Befehl ausführt, wartet auf die Beendigung der beiden Tasks. Sind die Ergebnisse der Tasks in `a` und `b` gespeichert, kann der Thread beide Werte addieren

und den Wert zurückgeben. Die durch den `fork` erstellten Tasks erzeugen während ihrer Berechnung auch zwei Tasks, sollte der Wert von `n` beim Funktionsaufruf größer als 2 sein. Dadurch entstehen immer mehr Tasks, die parallel abgearbeitet werden können.

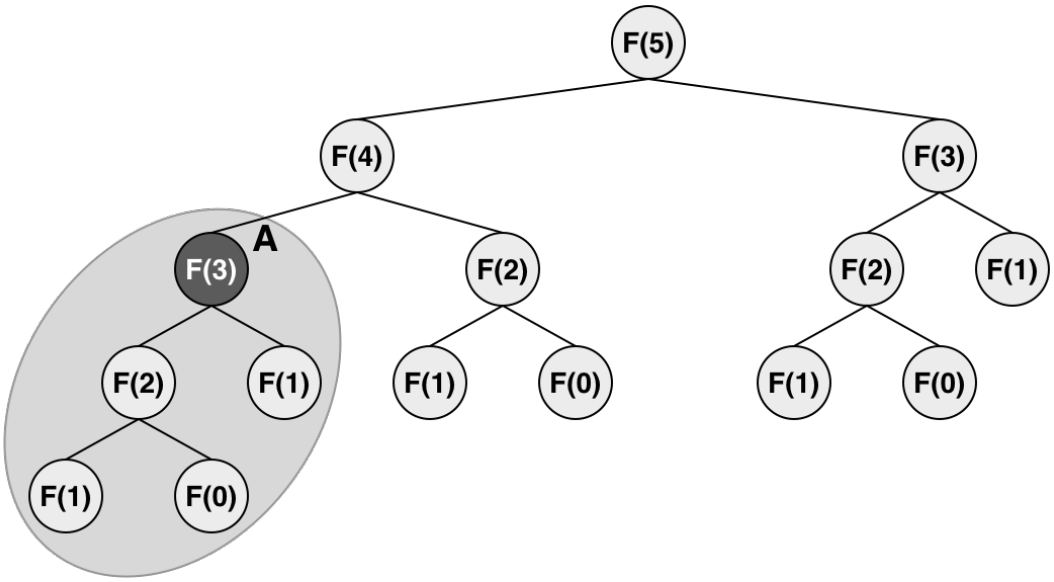

Abbildung 3.1: Berechnungsbaum für FIB(5)

In Abbildung 3.1 sieht man den Berechnungsbaum für `FIB(5)`. Jeder Knoten besitzt in diesem Fall zwei Kinder, die durch die zwei `fork` Befehle der Funktion erstellt wurden. Die Blätter stellen den Funktionsaufruf von `FIB(0)` und `FIB(1)` dar. Nehmen wir an, durch einen Absturz einer Recheneinheit geht der hervorgehobene Task A verloren. Dadurch ist der umkreiste Teilbaum betroffen. Das Programm hat nun die Aufgabe, die Teilberechnungen im Zusammenhang mit der abgestürzten Einheit zu identifizieren. Es ist möglich, dass zum Beispiel `FIB(2)` bereits das Ergebnis zurückgeliefert hat und der Task nicht mehr aktiv ist. Das Ergebnis dieser Teilberechnung ist mit dem Absturz verloren gegangen und muss erneut berechnet werden. Teilberechnungen, die sich noch in Berechnung befinden, können erhalten bleiben. Im letzten Schritt müssen die Verbindungen zwischen wiederhergestellten und erhaltenen Tasks hergestellt werden.

3.2 Fork/Join mit Work-Stealing

Damit das Fork/Join Programm auf einem Cluster ausgeführt werden kann, wird es um Work-Stealing erweitert. Jede Recheneinheit besitzt wie bei dem GLB Framework einen Worker. Alle Worker besitzt eine Main Task. Zu Beginn wird auf dem ersten Worker ein Start-TaskFrame erzeugt. Ein TaskFrame beinhaltet die zu berechnende Aufgabe, die in mehrere Ausführungsschritte unterteilt ist. Bei der Berechnung von einer Fibonacci Zahl repräsentiert ein Ausführungsschritt entweder, die Berechnung von `a, b` oder dem `join` Befehl mit Ergebnisberechnung.

Alle Worker durchlaufen während der gesamten Laufzeit eine Schleife. Wenn kein TaskFrame vorhanden ist, versucht er von einem zufällig ausgewählten Worker zu stehlen. Wenn der Stehlvorgang erfolgreich ist, wird eine Verbindung zum Opfer für die spätere Ergebnisrückgabe gespeichert. Anschließend wird ein TaskFrame berechnet. Bei einem `fork` Befehl, wird ein neuer Kind TaskFrame erzeugt und direkt bearbeitet. Der alte TaskFrame wird in einem Taskpool gespeichert und kann von dort gestohlen werden. Nach Beendigung des Kind TaskFrames wird der Eltern TaskFrame aus dem Taskpool entnommen, um das Ergebnis zu speichern. Wurde der Eltern TaskFrame gestohlen, muss das Ergebnis zwischengespeichert werden. Der Dieb überträgt den Eltern TaskFrame, wenn seine Berechnungen abgeschlossen sind, zurück zum Kind. Das Zwischenergebnis kann dann in den TaskFrame eingefügt werden. Das Programm terminiert, wenn der Main Task bearbeitet wurde.

In den folgenden Abschnitten werden einige Datenstrukturen erläutert, die für die Kommunikation benötigt werden.

TaskFrame

Der TaskFrame bildet eine Teilaufgabe eines Tasks ab. Um eigene Algorithmen zu implementieren, muss man von der Klasse in Quelltext 3.2 erben. Die Berechnungsfunktion ist in mehrere Schritte aufgeteilt. Die Variable `step` speichert die Nummer der durchgeführten Schritte und beim Aufruf der Berechnungsfunktion `compute()` wird mittels Sprunganweisung die nächste Teilaufgabe ausgeführt.

```
1  TaskFrame:
2    int step = 0, stolen = 0
3    T *result
4    ID id
5    owner = here
6
7    virtual compute()=0
8
9    fork(childFrame):
10     this.step++
11     push(this) // zum Taskpool hinzuf gen
12     childFrame.compute()
13     // Kann Eltern TaskFrame aus Taskpool entnommen werden?
14     if pop() == false:
15       // Ergebnis zwischenspeichern
16       parentReturn(childFrame)
17
18   join():
19     if stolen:
20       returnFrame(this) // gestohlenen Frame an Besitzer senden
```

Quelltext 3.2: Fork/Join TaskFrame

Um eine Zuordnung zu ermöglichen, besitzt jeder TaskFrame eine eindeutige Identifikation (kurz: ID). Wenn ein TaskFrame gestohlen wird, um einen weiteren Berechnungsschritt auszuführen, erhält er eine neue ID. Damit kann man nachvollziehen, woher der TaskFrame stammt und welche Teilaufgabe er erledigt. Außerdem wird das Opfer als Besitzer (`owner`) eingetragen.

Ist der auszuführende Schritt ein `fork` Befehl, wird der `step` erhöht und der TaskFrame in den Taskpool eingefügt. Ein Kind TaskFrame wird erzeugt und bearbeitet. Ist die Ausführung des Kind TaskFrames abgeschlossen, wird der Eltern TaskFrame aus dem Taskpool geholt (`pop()`). Schlägt dies fehl, da der Frame gestohlen wurde, wird das Ergebnis in einer Map für Teilergebnisse eingetragen. Diese Map speichert zur ID des Eltern TaskFrames das Ergebnis des Kindes, um es zu einem späteren Zeitpunkt einzufügen. Ansonsten wird das Ergebnis in dem Eltern TaskFrame gespeichert und dessen Berechnung fortgesetzt.

Bei dem `join` Befehl wird das Ergebnis des TaskFrames berechnet. Ist der TaskFrame jedoch gestohlen, wird die Weiterleitung an den Besitzer gestartet. Ein TaskFrame merkt

sich die Anzahl der Stehlvorgänge in `stolen`. Der Frame wird so oft an den jeweils vorherigen Besitzer weitergeleitet, bis er bei dem Worker angelangt ist, der den Frame erzeugt hat. Bei jeder Weiterleitung wird in der Map der Teilergebnisse geprüft, ob der Worker Teilberechnungen des Tasks erledigt hat. Dann wird das Teilergebnis in den Frame eingetragen und an den nächsten Worker geschickt.

Fibonacci

Die `FIB` Klasse aus Quelltext 3.3 ist eine Implementierung der Fibonacci Berechnung und erbt von TaskFrame. In einer Variable `n` steht die zu berechnende Fibonacci Zahl. Zur Speicherung der beiden Teilergebnisse von `Fib(n-1)` und `Fib(n-2)` gibt es die Variablen `a` und `b`. Die Berechnungsfunktion ist in 3 Schritte unterteilt. Der erste Schritt überprüft, ob der Spezialfall `n < 2` vorhanden ist und schreibt sofort das Ergebnis. Ansonsten wird ein `fork` ausgeführt, der Task `Fib(n-1)` berechnet. Bei dem zweiten Schritt wird `Fib(n-2)` berechnet. Im letzten Schritt wird `join` ausgeführt, um auf die erzeugten Tasks zu warten. Danach wird das Ergebnis zurückgegeben.

```
1  FIB inherits TaskFrame:
2    int n, a, b
3
4    FIB(n, result):
5      this.n = n
6      this.result = result
7
8    compute():
9      goto step
10     0:  if n < 2
11         *result = n
12         RETURN
13       fork(FIB(n - 1, &a))
14     1:  fork(FIB(n - 2, &b))
15     2:  join()
16       *result = a + b
```

Quelltext 3.3: Fork/Join Fibonacci TaskFrame

Worker

Der Worker speichert für die Ergebnisrückgabe von seinem aktuellen TaskFrame die ID seines Eltern TaskFrames. Es gibt vier Datenstrukturen zur Verwaltung der Stehlvorgänge:

Besitzer Eine Datenstruktur, die zu jedem gestohlenem TaskFrame den vorherigen Worker speichert.

Teilergebnisse Speichert in einer Map Teilergebnisse gestohlener TaskFrames, die zu einem späteren Zeitpunkt eingefügt werden.

Eltern Da von einem TaskFrame der nächste Ausführungsschritt gestohlen wird, ändert sich dessen ID. Eine Verbindung zwischen alter und neuer ID wird gespeichert.

Vereinigung Eine weitere Map dient zur Speicherung von bearbeiteten TaskFrames, die vom Dieb zum vorherigen Worker zurückgesendet werden. Diese TaskFrames können mit den Einträgen aus den Teilergebnissen vereinigt werden.

3.3 Lokale Fehlerbehebung

Um eine lokal begrenzte Fehlerbehandlung durchzuführen, muss man den aktuellen Stand der Berechnungen in einem Stehlbaum speichern. Im Gegensatz zum Berechnungsbaum stellen Worker die Knoten da und ein Knoten kann mehrere Tasks beinhalten. Die Kanten zeigen die Stehlvorgänge des Programmes an. Hat ein Worker seine Aufgabe erledigt, so wird die Verbindung zu seinem Opfer nach Übergabe des Ergebnisses entfernt. Es werden also nur aktive Worker in der Struktur gespeichert, um Speicherplatz zu sparen. In der Abbildung 3.2a ist ein Stehlbaum abgebildet. Worker W0 ist der Startpunkt. Worker W1 und W7 haben von W0 einen Task gestohlen. Die Worker W2, W3 und W4 haben von W1 gestohlen und W3 wurde von W5 und W6 bestohlen.

Bei einem Absturz von W3 müssen alle Berechnungen identifiziert werden, die mit diesem Worker zu tun hatten (siehe Abbildung 3.2b). Worker W1 wird *Recovery Knoten* von W3 genannt, weil W3 von dem Worker gestohlen hat. Ein *Recovery Knoten* startet die Identifizierung der verlorenen Teilberechnungen. Er überprüft, welche Task W3 gestohlen hat. Für jeden dieser Tasks wird geprüft, ob ein anderer Worker Teilberechnungen gestohlen hat.

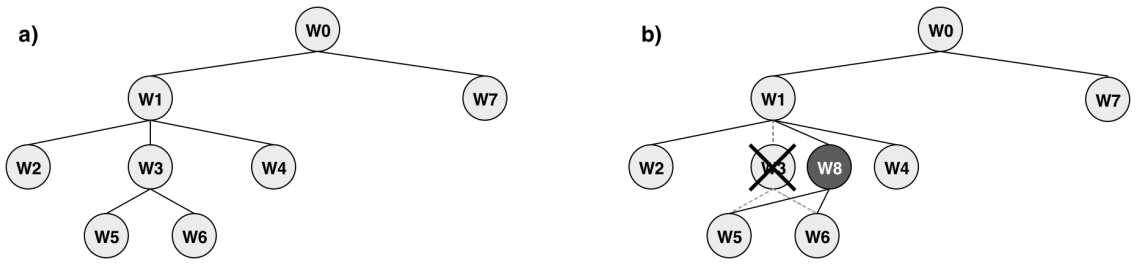

Abbildung 3.2: Stehlbaum a) vor und b) nach einem Absturz

Dies geschieht, indem der *Recovery Knoten* nacheinander Anfragen an die anderen Worker sendet und fragt, ob sie von W3 gestohlen haben. In diesem Beispiel sind W5 und W6 Besitzer von Teilberechnungen. Sie werden als *Frontier Knoten* der Berechnung markiert. Damit wird eine doppelte Ausführung von noch aktiven Berechnungen verhindert. Die gesammelten Informationen werden in einem *Replay Baum* gespeichert. Ein Worker wie W8, der bis jetzt noch keine Aufgabe übernehmen konnte, kann ein Replay stehlen und verarbeiten. Er verbindet sich mit W1 als *Alias Knoten* für den abgestürzten Worker. Er wird der Besitzer von den verlorenen Tasks und wiederholt sie. Der *Alias* setzt Verbindungen zu den *Frontier Knoten*, damit diese das Teilergebnis zurücksenden können. Währenddessen ist der TaskFrame gesperrt und kann nicht gestohlen werden. Erst nachdem alle Verbindungen wiederhergestellt sind, können die Tasks vom *Alias Knoten* gestohlen werden, um die Last wieder zu verteilen. Auch bei mehreren gleichzeitigen Abstürzen, können die *Recovery Knoten* alle Tasks wiederherstellen und es wird sichergestellt, dass verlorene Berechnung nur einmal ausgeführt werden.

3.4 Algorithmus zur lokalen Fehlerbehebung

Der fehlertolerante Algorithmus baut auf dem im Abschnitt 3.2 vorgestellten verteilten Fork/Join Programm auf. Hier werden die Erweiterungen der Datenstrukturen beschrieben, die eine lokale Fehlerbehebung ermöglichen.

3.4.1 Erweiterung der Datenstrukturen

Der Worker durchläuft eine Schleife, in der er zuerst einen TaskFrame stiehlt und diesen anschließend berechnet. Die Anzahl der Durchläufe der Schleife werden im Worker gezählt. Sie gibt die Nummer der Arbeitsphase an. Die Arbeitsphase erhöht sich nur nach einem erfolgreichen Stehlvorgang. Der TaskFrame erhält ein Level, das die Rekursionstiefe der Berechnung angibt. Wie bereits erwähnt, besitzt der TaskFrame eine eindeutige ID. Sie setzt sich zusammen aus dem Besitzer Worker und dessen Phase, dem Level des TaskFrames und dem auszuführenden Schritt. Aus der ID kann man genau ablesen, welche Aufgabe der TaskFrame berechnet.

Da die Berechnungsschritte eines TaskFrames von mehreren Workern ausgeführt und gestohlen werden, benötigt man nicht nur die Informationen von einem TaskFrame zu seinem Eltern TaskFrame sondern auch die Informationen von anderen Stehlvorgängen auf diesen TaskFrame. Dieses wird in Stolen Steps aufgelistet. In einem Stolen Step wird die beim Stehlvorgang generierte ID zusammen mit dem Dieb gespeichert. Die Reihenfolge in der Liste der Stolen Steps repräsentiert das Level im Stehlbaum, bei dem die Fortsetzung eines Tasks gestohlen wurde. Jede Liste enthält immer alle vorherigen Stehloperationen von allen Workern. Diese Eigenschaft wird während der Wiederherstellung zum Erstellen des Replays der verloren gegangen Berechnungen und zur Identifizierung der Verbindungen zu *Frontier Knoten* benutzt.

In Abbildung 3.3 sieht man die zusätzlichen Datenstrukturen nach einem Stehlvorgang. Der Pfad der Stehlvorgängen wird in einer Liste gespeichert (Stolen Path). Ein Eintrag setzt sich aus der ID eines TaskFrames und dessen Stolen Steps zusammen.

In der Datenstruktur Stolen Children wird ein Backup der gestohlenen TaskFrames gesichert. Zu jedem Dieb wird eine eigene Map angelegt. Als Schlüssel dient die ID des TaskFrames. Dazu wird eine Sicherungskopie vom gesamten TaskFrames gespeichert und seinem Pfad der Stehlvorgänge, um alle Schritte von Erstellung des TaskFrames bis zum aktuellen Stehlvorgang nachvollziehen zu können.

Damit die Datenstruktur nicht zu groß wird, kann ein Eintrag gelöscht werden, wenn der Dieb den berechneten TaskFrame zurückschickt. Der Dieb steht ab diesem Zeitpunkt nicht mehr für eine Fehlerbehandlung des TaskFrames zur Verfügung, weil er in eine

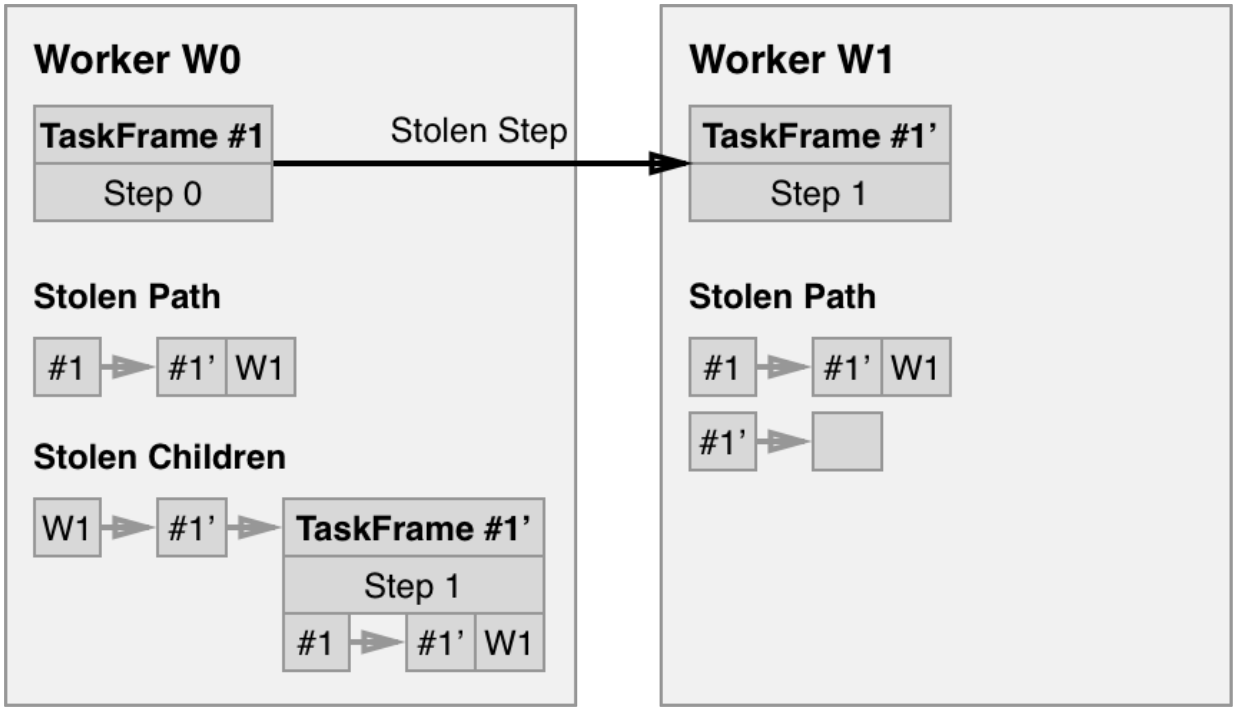

Abbildung 3.3: Datenstruktur nach Stehlvorgang

neue Arbeitsphase eintritt und keine Informationen mehr zu dem gestohlenen Task gespeichert hat. Dieser Frame muss im Falle eines Absturzes des Opfers erneut ausgeführt werden.

3.4.2 Wiederherstellung

Bei einem Ausfall eines Workers werden die noch aktiven Worker benachrichtigt. Sie leiten jeweils eine eigene Wiederherstellung von verlorenen TaskFrames ein, die unabhängig von anderen Workern ausgeführt wird. Die Aufgabe eines Workers ist es, herauszufinden, ob er ein *Recovery Knoten* ist. Dafür wird in der Stolen Children Datenstruktur nach gestohlenen TaskFrames des abgestürzten Workers gesucht. Für jeden gefundenen TaskFrame wird ein Replay erzeugt.

Ein Replay speichert den TaskFrame, den Stolen Path aus Stolen Children und einen *Replay Baum*. Ein *Replay Baum* besteht aus allen Stolen Paths, die die ID des TaskFrames beinhalten. Jeder Worker bekommt die Aufgabe, in seinem eigenen Stolen Path nach der TaskFrame ID

zu suchen. Bei einem Treffer wird der Stolen Path übertragen. Außerdem werden in Stolen Children auch alle Kopien der Stolen Paths durchsucht.

Alle erzeugten Replays eines *Recovery Knotens* werden in einer Replay Liste gespeichert, von der andere Worker stehlen können. Bei einem Stehlvorgang wird zuerst in die Replay Liste geguckt, bevor ein anderer TaskFrame gestohlen wird. In Stolen Path wird der Stolen Step aktualisiert, damit der Eintrag nicht mehr den abgestürzten Worker beinhaltet. Wie bei einem normalen Stehlvorgang wird der Frame zusammen mit dem Stehlpfad in die Stolen Children Datenstruktur eingetragen.

Der Dieb eines Replays ist der *Alias Knoten* für diese Wiederherstellung. Er berechnet die *Frontier Knoten* aus dem *Replay Baum*. Jeder Stolen Step vom fehlenden TaskFrame wird durchlaufen. Es wird überprüft, ob der Dieb noch aktiv ist. Ist dies der Fall, so gilt der Dieb als *Frontier Knoten* für die Berechnungsphase.

Bei der Bearbeitung eines Replays gibt es 3 Phasen. Die erste Phase heißt Privatization und führt die Berechnungsfunktion solange aus, bis das Level des Frames auf ein *Frontier* trifft. Während der Ausführung ist ein Stehlvorgang des Frames verboten. Die Phase 2 heißt Enforced Steal und beginnt, wenn der nächste Ausführungsschritt als *Frontier* identifiziert wurde, aber der *Frontier Knoten* nicht mehr aktiv ist. Die Berechnung des Frontiers muss wiederholt werden. In diesem Fall wird das Replay in die Liste der Replays geschrieben, um es stehlbar zu machen. Die letzte Phase heißt Patching und sie wird ausgeführt, wenn der *Frontier Knoten* noch aktiv ist. Der Worker aktualisiert die Verbindung und kann die nächsten Aufgabe beginnen.

3.5 Beispiel

In Abbildung 3.4 ist im Vergleich zur vorherigen Abbildung noch ein weiterer Stehlvorgang eingezeichnet. Der gestohlene TaskFrame von Worker W0 wird aus W1 von einem weiteren Worker gestohlen. Der Stolen Path von W1 enthält für TaskFrame #1' einen Stolen Step und in Stolen Children ist eine Kopie des TaskFrames #1'' gespeichert.

Nach einem Absturz von W1 sind alle Informationen, die innerhalb des Workers gespeichert wurden, nicht mehr verfügbar. Die Aufgabe der Fehlerbehandlung ist es, den

Abbildung 3.4: Datenstruktur vor Absturz von W1

TaskFrame mit dem Step 1 wiederherzustellen und die Verbindung zwischen Step 0 und Step 2 zu setzen.

Der Worker W0 ist ein *Recovery Knoten* von W1, da er Einträge von diesem Worker in Stolen Children besitzt. Es wird ein Replay für Step 1 erstellt (Abbildung 3.5). Der *Replay Baum* ist die Vereinigung der Stolen Paths von W0 und W2, weil die ID des verlorenen TaskFrames dort auftaucht.

Abbildung 3.5: Replay für Step 1

Ein neuer Worker stiehlt das Replay und ist *Alias* von W1. Die Verbindung zu W0 wird hergestellt, damit das Ergebnis des Replays übermittelt werden kann. Im *Replay Baum* ist in den Stolen Steps von TaskFrame #1' der Stehlvorgang von W2 zu sehen. Da Worker W2 noch aktiv ist, gilt er als *Frontier Knoten* für Step 2.

Der *Alias* beginnt das Replay auszuführen. In der Privatization wird Step 1 ausgeführt. Anschließend wird für die nächste Ausführung ein *Frontier* erkannt. In der Phase Patching wird die Verbindung zwischen *Alias Knoten* und *Frontier* gesetzt. Damit kann das Ergebnis aus Step 2 von Worker W2 in den *Alias Knoten* übertragen werden.

3.6 Mehrfache Abstürze

Ein Fehler kann jederzeit auftreten. Es können nicht nur mehrere Worker gleichzeitig sondern auch während der Recovery ausfallen. Mithilfe von Abbildung 3.6 wird beschrieben, wie mehrfache Abstürze abgefangen werden.

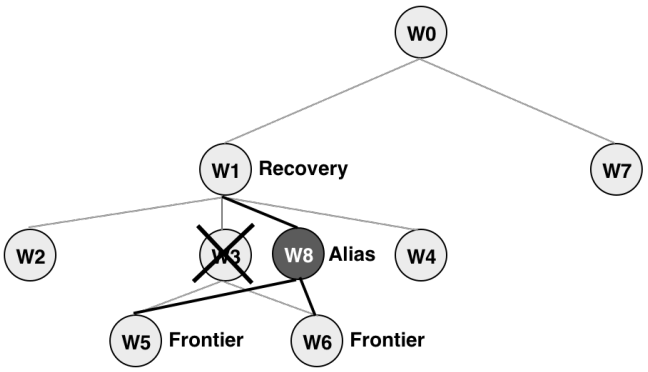

Abbildung 3.6: Stehlbaum nach Fehlerbehandlung

Fällt einer der *Frontier Knoten* W5 oder W6 aus, gibt es zwei Möglichkeiten. Hat der *Alias Knoten* W8 noch keine Berechnung des *Frontiers* durchgeführt, dann ist der Ausfall genauso als wäre der *Frontier Knoten* zusammen mit dem abgestürzten Worker W3 ausgefallen. Wenn W8 bereits das *Frontier* berechnet hat, so sind W5 und W6 Kindknoten von ihm. Dadurch ist der *Alias Knoten* W8 auch ein *Recovery Knoten* für die beiden Kinder und kann deren Arbeit wiederherstellen.

Stürzt der *Recovery Knoten* W1 ab, bevor W8 sein Replay übernimmt, dann passiert das gleiche, wie bei einem gleichzeitigen Absturz. Ist bereits die Verbindung zum *Alias Knoten* W8 vorhanden, so gilt dieser für den neuen *Recovery Knoten* W0 als *Frontier*.

Wenn ein unbeteiligter Worker abstürzt, wie zum Beispiel W7, dann wird eine unabhängige Recovery gestartet. Diese Recovery kann gleichzeitig ausgeführt werden.

4 Konzept

In diesem Kapitel werden zuerst die Unterschiede von dem GLB Framework und dem Fork/Join Algorithmus aufgezählt. Anschließend wird die Übertragung des fehlertoleranten Fork/Join Algorithmus in das GLB Framework beschrieben und nötige Anpassungen erläutert.

Der größte Unterschied der beiden Algorithmen ist der Aufbau und die Abarbeitung der Tasks. Bei einem Fork/Join gibt es TaskFrames, die mehrere Ausführungsschritte besitzen und Kinder TaskFrames erzeugen können. Die Kinder leiten ihre Ergebnisse an die Eltern weiter, bis der Starttask sein Ergebnis berechnet hat. Im Gegensatz dazu beinhaltet ein Task in GLB alle Ausführungsschritte eines TaskFrames. Neu erzeugte Tasks werden an das Ende eines Taskpools angehangen und sind unabhängig von anderen Tasks. Ein Programm endet, nachdem alle Taskpools leer sind und das Ergebnis per Reduktion berechnet wurde.

Dadurch kommen in diesem Konzept keine Datenstrukturen zum Einsatz, die für das Einsetzen von Ergebnissen in Eltern Tasks zuständig sind. Nach Fertigstellung eines Tasks wird keine Kommunikation benötigt, um das Ergebnis bzw. den Task an den vorherigen Worker zurückzuschicken.

Fork/Join verwendet das work-first Prinzip (siehe Abschnitt 3.1). Bei GLB werden neue Tasks an das Ende des Taskpools angehängt und der erste Task abgearbeitet. Beide Algorithmen benutzen Work-Stealing zur Verteilung der Last. Bei Fork/Join wird von einem TaskFrame der nächste Bearbeitungsschritt gestohlen. Stattdessen wird bei GLB die Hälfte des Taskpools eines Workers gestohlen, wodurch die Anzahl der Stehlanfragen verringert wird.

In den folgenden Abschnitten werden die Anpassungen der GLB Konstrukte TaskBag, TaskQueue und Worker erläutert, die in Abschnitt 2.3 vorgestellt wurden. Danach wird die

neue Fehlerbehandlung beschrieben und der Ablauf eines Replays. Im letzten Abschnitt werden die Änderungen der Reduktionsfunktion erklärt, um Fehler bei der Ergebnisberechnung abzufangen.

4.1 TaskBag

Ein TaskBag benötigt eine eindeutige ID, wie der TaskFrame aus dem Fork/Join Programm. Der Aufbau der ID muss angepasst werden, da sich die Bearbeitung unterscheidet. Der Worker und die Anzahl der Aufrufe seiner Berechnungsfunktion können weiterhin in der ID gespeichert werden. Anstatt die Tiefe der Rekursion als Level zu benutzen, wird die Anzahl der abgearbeiteten Task aus dem Taskpool verwendet. Zusätzlich werden die Teilungen des Taskpools gespeichert, da keine Ausführungsschritte gestohlen werden (Stolen Steps).

4.2 TaskQueue

Die TaskQueue, die den Taskpool verwaltet, speichert sich die ID vom zuletzt erhaltenen TaskBag. Wird ein Task aus dem Pool bearbeitet, wird das Level erhöht. Die Teilung des Taskpools in einen TaskBag, erzeugt eine ID, dessen Teilvorgänge um eins erhöht werden.

Da sich Diebe in eine Liste eintragen und somit die Beantwortung mehrerer Stehlanfragen gleichzeitig möglich sind, können mehrere TaskBags mit dem selben Level verschickt werden. Bei einer Fehlerbehandlung müssen also mehrere *Frontier Knoten* pro Level unterstützt werden.

4.3 Worker

Ein Worker zählt nun auch die Aufrufe der Berechnungsfunktion. Die Anzahl steht für die Arbeitsphase des Workers.

Der Worker überprüft in seiner Hauptfunktion, ob er ein Replay von einem anderen Worker gestohlen hat. Ist sein Taskpool leer und ist ein Replay vorhanden, dann wird der Worker ein *Alias Knoten*. Er fügt das Replay, welches einen TaskBag enthält, in seinen Taskpool ein. Der Taskpool wird während der Ausführung gesperrt, um zu verhindern, dass von anderen Workern Tasks gestohlen werden. Nach Fertigstellung des Replays kann wieder auf den Taskpool zugegriffen werden. Sollte kein Replay verfügbar sein, so wird der Taskpool wie bisher abgearbeitet.

Die Datenstrukturen Stolen Path und Stolen Children wurden aus Unterabschnitt 3.4.1 übernommen und an TaskBags angepasst. Der Stolen Path speichert die IDs von gestohlenen Taskbags. Die Stolen Steps repräsentieren nun die Teilungen des Taskpools. Stolen Children speichert einen TaskBag statt eines TaskFrames, der bei einem Stehlvorgang übertragen wird.

Die Stehlanfragen werden wie bisher verschickt. Bei der Beantwortung erhält der Dieb ein Replay statt einem TaskBag, wenn auf dem Opfer eine Fehlerbehandlung durchgeführt wurde. Das Opfer eines Stehlvorgangs trägt seine Place Nummer in die ID eines TaskBags ein. Bevor eine Antwort an den Dieb gesendet wird, wird der Stolen Path im Opfer aktualisiert und das Replay bzw. der TaskBag in die Stolen Children eingetragen.

Der Dieb bekommt den Stolen Path übergeben. Anstatt den eigenen Stolen Path zu überschreiben, werden nur die neuen Einträge übernommen, da bei GLB mehrere TaskBags gestohlen werden können. Sollte ein Listeneintrag mehr Stolen Steps beinhalten, als in der eigenen Liste vorhanden sind, werden die fehlenden Einträge ergänzt. Damit soll der Speicherverbrauch bei mehrfachen Stehlvorgängen verringert werden, da immer nur die aktuellste Liste der Stolen Steps gespeichert wird.

Ein Problem bei GLB ist, dass durch die fehlende Kommunikation zu den Elterntasks niemals ein Teilergebnis zu einem anderen Worker übertragen wird. Deshalb kann in der Stolen Children Datenstruktur kein Eintrag entfernt werden. Da mehrere Tasks in Form eines TaskBags übertragen werden, enthält die Datenstruktur jedoch weniger Einträge als bei Fork/Join, bei der jeder TaskFrame einzeln gestohlen wird.

4.4 Lokale Fehlerbehebung

Bei einem Absturz kann das System durch eine Fehlerbehandlung die Arbeit mit weniger Rechenknoten fortsetzen. Man geht davon aus, dass ein abgestürzter Knoten bis zum Ende der Laufzeit nicht mehr zur Verfügung steht. Bei einem Ausfall vom ersten Rechenknoten, der für die Initialisierung der Tasks zuständig war, funktioniert die Fehlerbehandlung nicht und das Programm stürzt ab.

Für die Fehlerbehandlung wird in jedem Worker ein Handler registriert, der im Falle eines Absturzes eines Places ausgeführt wird. Jeder Worker besitzt eine Liste, in der er abgestürzte Worker einträgt, dessen Handler bereits ausgeführt wurde. Damit wird sichergestellt, dass jede Fehlerbehandlung nur einmal ausgeführt wird.

Die Generierung des Replays läuft analog zu der in Unterabschnitt 3.4.2 vorgestellten Methode. Statt der TaskFrame IDs werden TaskBag IDs verwendet. In der Berechnung des *Frontiers* gibt es eine Anpassung für TaskBags. Für ein Berechnungslevel kann es, wie bereits erwähnt, mehrere *Frontier Knoten* geben. Deshalb muss nicht nur gespeichert werden, ob es bei einem Level ein *Frontier* gibt, sondern auch wie viele. Am Ende der Fehlerbehandlung wird der Status des Workers auf aktiv gesetzt und die Hauptschleife ausgeführt, um erzeugte Replays zu verteilen oder von anderen Workern zu stehlen. Ist nur noch ein Worker vorhanden, so bearbeitet er seine eigenen Replays.

4.5 Replay

Der TaskBag des Replays wird in den eigenen leeren Taskpool eingefügt. Würden noch Tasks im Taskpool vorhanden sein, so würden bei einer Teilung die falschen Tasks verworfen werden. Ist kein *Frontier Knoten* vorhanden, kann das Programm normal fortgesetzt werden. Gibt es jedoch *Frontier Knoten*, darf nur die Wiederherstellungsfunktion auf den Taskpool zugreifen.

Die Berechnungsfunktion wird solange ausgeführt, bis man an ein *Frontier* gelangt. An dieser Stelle wird der Taskpool geteilt und der Stolen Path sowie die Stolen Children aktualisiert. Ab diesem Zeitpunkt gilt der Worker als *Recovery Knoten* für das *Frontier*.

Sind alle *Frontier Knoten* abgearbeitet, kann der Taskpool entsperrt und Stehlanfragen beantwortet werden.

4.6 Reduktion

Vor der Reduktion wird auf allen Workern gleichzeitig überprüft, ob alle Fehlerbehandlungen abgeschlossen sind. Dafür wird die Anzahl der laufenden Worker und die Anzahl der abgestürzten Worker addiert und mit der Anzahl der Worker zum Programmstart verglichen. Sollte ein Absturz noch nicht registriert worden sein, wartet der Worker auf die Fehlerbehandlung und arbeitet anschließend die erzeugten Replays ab. Danach werden die Ergebnisse der Worker gespeichert. Sollte es bei der Übertragung des Ergebnisses zu einem Fehler kommen, beginnt die Überprüfung erneut. Die Reduktion wird zum Schluss ausgeführt und das Ergebnis zurückgegeben.

5 Implementierung

Das Kapitel Implementierung beschreibt die wichtigsten Details der Implementierung der Fehlerbehandlung für GLB. Hierbei handelt es sich um die Erstellung der Backups vor der Kommunikation, sowie der Erstellung und Ausführung von Replays. Am Ende wird auf die Probleme während der Entwicklung eingegangen.

In Quelltext 5.1 kann man die Übertragung eines TaskBags zu einem Dieb sehen. Der Worker holt sich eine Anfrage aus der `thieves` Queue. Dann wird die aktuelle Phase des Workers und das Level der TaskQueue ausgelesen und in die ID des zu verschickenden TaskBags eingetragen. Die ID besteht zusätzlich aus dem Zähler der Splits der TaskQueue. Die ID und der Dieb werden in dem neusten Listeneintrag des Stolen Paths als Stolen Step angehängt. Die `updateChildren` Funktion speichert zu der neuen ID eine Sicherungskopie des TaskBags und dem Stolen Path in den Stolen Children. Sollte noch keine Map Datenstruktur angelegt worden sein, so wird das in dieser Methode erledigt.

Zum Schluss wird zur Übertragung des TaskBags eine Activity auf dem Dieb erzeugt. Der Dieb aktualisiert seinen Stolen Path und mit der `deal` Funktion wird der TaskBag und seine ID in die TaskQueue eingefügt.

In Quelltext 5.2 wird die Erstellung der Replays ausgeführt. Die Funktion wird von dem Handler der Fehlerbehandlung aufgerufen. Im ersten Schritt werden die Stolen Children des abgestürzten Workers als Map geladen. Ist die Map `null` , so ist der Worker kein *Recovery Knoten* und er kann die Fehlerbehandlung verlassen. Ansonsten wird durch alle gestohlenen IDs iteriert und für jede ID zuerst ein Replay Baum erzeugt. In einer Schleife werden alle Places durchlaufen und in einer Activity werden der Stolen Path und die Stolen Children in einem Replay Baum übertragen, falls die ID dort auftaucht. Die Methode `contains` des Stolen Path durchsucht die Liste der Stolen Steps nach der ID. In Zeile 16 werden die Sicherungskopien von Stolen Path nach der ID durchsucht. Die

Activity liefert einen Teil des Replay Baums zurück an den *Recovery Knoten*. Dort wird er mit den anderen Teilbäumen zusammengefügt, wodurch der gesamte Replay Baum entsteht. Anschließend muss nur noch die Sicherungskopie des TaskBags und des Stolen Paths zusammen mit dem Replay Baum in ein Replay Objekt gespeichert werden und in die Liste der Replays hinzugefügt werden.

Die Funktion executeReplay aus Quelltext 5.3 wird nach einem Stehlvorgang vom *Alias* ausgeführt. Zuerst wird die `computeFrontier(Replay)` Methode aufgerufen, um die *Frontier Knoten* zu berechnen. Die Funktion liefert die Liste der Stolen Steps der *Frontiers* zurück (`frontiers`) und eine Map `has_frontiers` , die für jedes Level speichert, wie viele *Frontiers* es gibt.

Anschließend wird der TaskBag des Replays in den TaskPool mit der Funktion `processLoot` eingefügt. Falls es *Frontier Knoten* gibt, werden die Tasks ausgeführt, bis für das Level ein Eintrag in der Map `has_frontiers` gefunden wurde. Die Variable `splits` speichert die Anzahl der Teilvorgänge und in einer Schleife werden die *Frontiers* gepatcht. Der Stolen Path erhält einen Stolen Step, welches auf den *Frontier* zeigt und speichert den *Frontier Knoten* in seinen Stolen Children. Wurde das letzte *Frontier* erreicht, wird die Funktion beendet und die normale Ausführung kann fortgesetzt werden.

Während der Entwicklung sind einige Fehler aufgetreten, die bis zum Schluss nicht vollständig gelöst werden konnten. Es kann passieren, dass APGAS den Handler zur Fehlerbehebung nicht aufruft, bevor das Programm terminiert. Dadurch kann ein abgestürzter Place nicht erkannt werden und ein falsches Ergebnis wird ausgegeben. Eine Ausführung mit einer sehr hohen Anzahl von Places kann dazu führen, dass das Programm nicht terminiert. Dadurch konnte die Fehlerbehandlung beim Absturz mehrerer Places nur begrenzt getestet werden.

```
1  public void give(final TaskBag loot) {
2    final int victim = here().id;
3    ...
4    if (this.thieves.size() > 0) {
5      final Integer thief = this.thieves.poll();
6      ...
7      final long victimPhase = this.phase;
8      final long level = queue.getLevel();
9      final ID id = new ID(victim, victimPhase, level,
           loot.getId().getSplits());
10     loot.setID(id);
11     this.st_path.updateLast(new SStep(id, t));
12
13     final StPath path = new StPath(this.st_path);
14     this.updateChildren(t, id, loot, this.st_path);
15     ...
16     try {
17       uncountedAsyncAt(place(thief), () -> {
18         synchronized (this.waiting) {
19           this.st_path.map(path);
20           this.st_path.add(new StPathEntry(id));
21
22           this.deal(loot, victim, id, true);
23
24           this.waiting.set(false);
25           this.waiting.notifyAll();
26         }
27       });
28     } catch (Throwable throwable) {
29       throwable.printStackTrace(System.out);
30     }
31   }
32   ...
33 }
```

Quelltext 5.1: give(TaskBag) aus der Worker Klasse

```
1  private void generateReplay(int deadPlace) {
2    HashMap<ID, Pair<TaskBag, StPath>> hashMap =
         this.st_children.get(deadPlace);
3    if (hashMap == null) return;
4    // generate replay tree
5    for (final ID id : hashMap.keySet()) {
6      StPath rtree = new StPath();
7
8      for (Place place : places()) {
9        try {
10         StPath currentTree = at(place, () -> {
11           StPath localTree = new StPath();
12           if (st_path.contains(id)) localTree.map(st_path);
13           for (int thief : st_children.keySet()) {
14             HashMap<ID, Pair<TaskBag, StPath>> thiefMap =
                   st_children.get(thief);
15
16             for (ID stolenId : thiefMap.keySet()) {
17               Pair<TaskBag, StPath> pair = thiefMap.get(stolenId);
18               StPath stPath = pair.getSecond();
19               if (stPath.contains(id)) localTree.map(stPath);
20             }
21           }
22           return localTree;
23         });
24         rtree.map(currentTree);
25       } catch (Throwable t) {
26         t.printStackTrace(System.out);
27       }
28     }
29     Pair<TaskBag, StPath> pair = hashMap.get(id);
30     replays.add(new Replay(pair.getFirst(),pair.getSecond(),rtree));
31   }
32 }
```

Quelltext 5.2: generateReplay(in) aus der Worker Klasse

```
1  private void executeReplay(Replay replay) {
2    if (replay == null) return;
3
4    TaskBag loot = replay.getLoot();
5
6    Pair<LinkedList<SStep>, HashMap<Long, Integer>> frontierPair =
         computeFrontier(replay);
7    LinkedList<SStep> frontiers = frontierPair.getFirst();
8    HashMap<Long, Integer> hasFrontier = frontierPair.getSecond();
9    // no frontier, calculate all
10   if (frontiers.isEmpty()) {
11     this.processLoot(loot, false, loot.getId(), 1);
12   } else {
13     // split at frontier
14     boolean process = true;
15     this.processLoot(loot, false, loot.getId(), 1);
16
17     while (process && !frontiers.isEmpty()) {
18       long currentLevel = this.queue.getLevel();
19       Integer splits = hasFrontier.get(currentLevel);
20       // frontier found
21       if (splits != null) {
22         for (int i = 0; i < splits; i++) {
23           SStep frontier = frontiers.poll();
24           TaskBag newLoot = this.queue.split();
25           newLoot.setID(frontier.getId());
26           this.st_path.updateLast(frontier);
27           this.updateChildren(frontier.getThiefID(), frontier.getId(),
                 newLoot, st_path);
28         }
29       }
30       process = this.queue.process(this.n);
31     }
32   }
33 }
```

Quelltext 5.3: executeReplay(Replay) aus der Worker Klasse

6 Experimente

Auf dem Cluster des IT Servicezentrum der Universität Kassel wurden Experimente durchgeführt, um die Performance des Algorithmus zu messen. Dafür wurden 12 Rechenknoten verwendet. Jeder Rechenknoten besitzt 2 Intel Xeon E5-2643 v4 Prozessoren mit 6 Kernen, die jeweils eine Taktfrequenz von 3,40 GHz besitzen, und 256GB Hauptspeicher[1]. Zum einen wurde ein Benchmark im fehlerfreien Fall ausgeführt mit mehreren GLB Varianten und anschließend wurde eine Fehlerbehandlung gemessen.

6.1 Unbalanced Tree Search

Bei der Unbalanced Tree Search werden die Knoten eines unbalancierten Suchbaumes gezählt. Der Baum wird während der Laufzeit erstellt und kann mit verschiedenen Parametern beeinflusst werden. Man kann unter anderem die maximale Tiefe oder den Verzweigungsgrad festlegen, welche die Laufzeit stark beeinflussen. Zur Erzeugung von Kindknoten wird ein Hash Algorithmus verwendet. Bei gleichen Startwerten erhält man bei jeder Ausführung den selben Baum. Das Ergebnis kann erst nach der Berechnung jedes Knotens ermittelt werden. Durch die ungleichmäßige Verteilung ist eine dynamische Lastenbalancierung, wie es GLB durchführt, sehr wichtig.[OHL+06]

Für den Benchmark wurde der Tiefenparameter auf 13 festgelegt, wodurch mehr als 250 Millionen Knoten berechnet werden müssen. Da dies auf dem Cluster sehr schnell abgearbeitet wird, wurde ein weiterer Test mit der Tiefe 17 durchgeführt. Die Laufzeit verlängert sich erheblich, da über 67 Milliarden Knoten erzeugt werden, und es werden deutlich mehr Stehlvorgänge durchgeführt.

1 https://www.uni-kassel.de/its-handbuch/daten-dienste/wissenschaftliche-datenverarbeitung/hardware-und-software/hardware-des-clusters.html

6.2 Ergebnisse

Im ersten Schritt wurden Zeitmessungen mit dem GLB Framework (GLB), der fehlertoleranten Backup Variante (BackupFT) und der Implementierung der neuen Fehlerbehandlung (LocalFT) durchgeführt. Als Benchmark kam Unbalanced Tree Search zum Einsatz. Dieses Experiment vergleicht den Overhead im fehlerfreien Fall, der durch die Fehlertoleranz erzeugt wird.

Places	LocalFT	GLB	BackupFT
1	31,2646	31,3025	32,4273
2	16,5858	17,7734	18,4950
4	10,1959	10,2006	11,7945
6	8,2307	8,2125	8,5730
12	6,3024	6,0730	7,9526
24	4,8073	4,4153	6,6205
48	4,7285	2,6118	5,8913

Tabelle 6.1: UTS Laufzeit in Sekunden - Tiefe 13

Tabelle 6.1 zeigt die Laufzeit des UTS Benchmarks mit der Tiefe 13 in Sekunden. Die Zahlen wurden aus dem Mittelwert von 5 Ausführungen berechnet. Bis 6 Places ist kein Overhead bei der lokalen Fehlerbehebung vorhanden. Der Algorithmus ist sogar minimal schneller als die Standard Variante. Anschließend wächst der Overhead mit steigender Anzahl der Places, bleibt jedoch immer unter dem Wert der Backup Variante. Aufgrund der begrenzten Anzahl von Tasks gibt es ab 48 Places keine weitere Verbesserung in der Laufzeit. In Abbildung 6.1 ist für diese Messung der Overhead in Prozent abgebildet.

Places	LocalFT	GLB	BackupFT
1	7565,745	7602,155	8020,815
6	1376,710	1377,896	1484,678
12	682,358	677,520	689,893
24	345,238	342,872	352,588
48	179,682	175,994	183,122
72	124,192	119,032	127,502
96	99,506	91,837	100,919
144	76,556	64,650	74,809

Tabelle 6.2: UTS Laufzeit in Sekunden - Tiefe 17

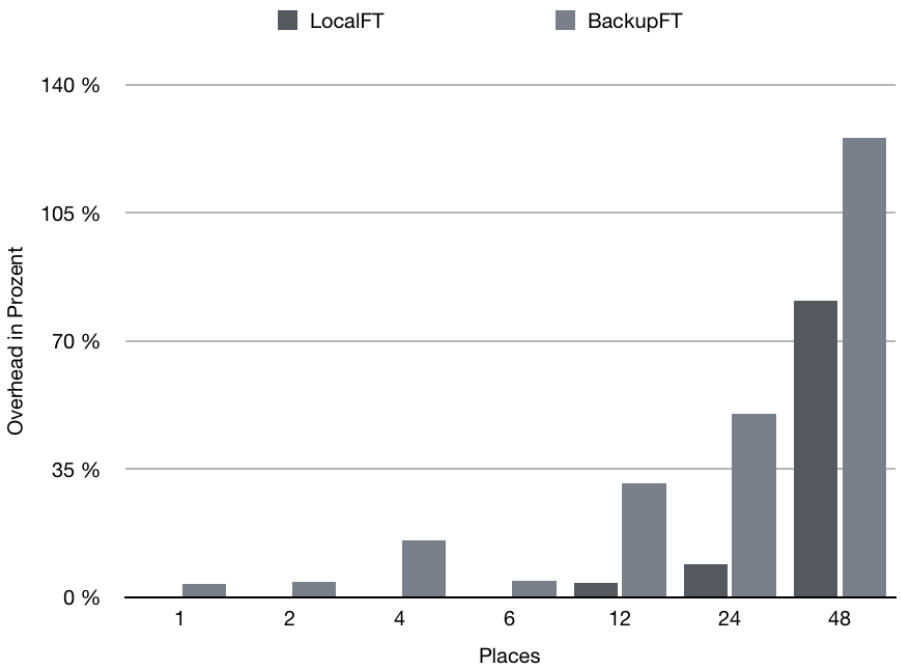

Abbildung 6.1: Overhead der Fehlertoleranz - Tiefe 13

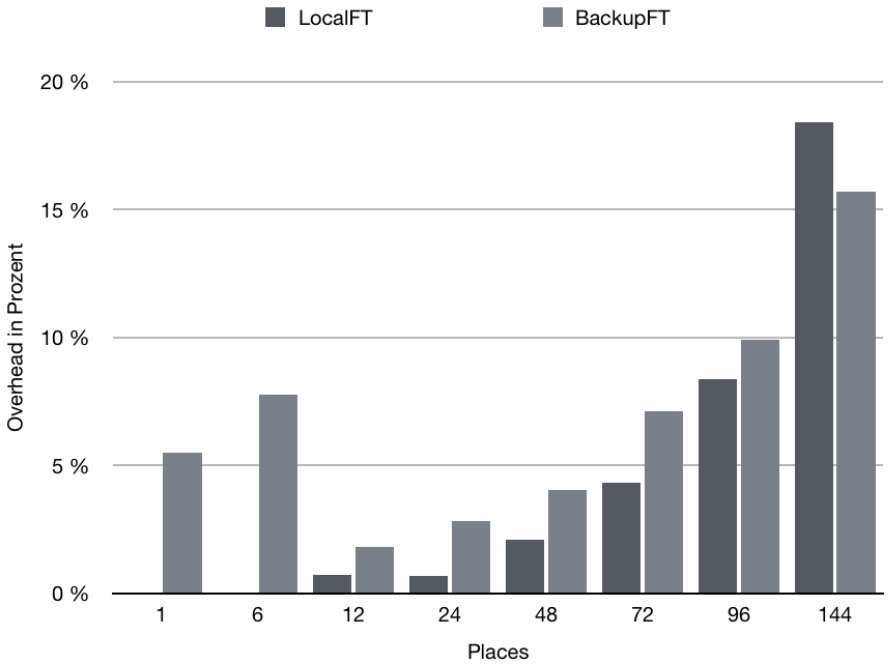

Abbildung 6.2: Overhead der Fehlertoleranz - Tiefe 17

Der Test mit einer Tiefe von 17 lastet alle Places aus (siehe Tabelle 6.2). Die verwendetete Hardware besitzt insgesamt 144 Prozessorkerne und bei diesem Test wurde für jeden Kern ein Place erstellt. Auch in Abbildung 6.2 ist ein Overhead bis 6 Places nicht erkennbar. Bei einer so hohen Anzahl von Tasks ist der Overhead der Backup Variante zuerst deutlich höher. Mit steigender Anzahl von Places nähern sich die beiden fehlertoleranten Varianten an. Bei 144 Places ist die Backup Variante zum ersten Mal schneller als die Implementierung der lokalen Fehlerbehebung.

Bei den Tests ist aufgefallen, dass der lokale Fehlerbehandlungsalgorithmus bei einer sehr hohen Anzahl von Places manchmal auch nach Stunden nicht terminiert. Diese Vorgänge wurden nicht in den Messungen berücksichtigt.

Nachdem die Laufzeit im fehlerfreien Fall gemessen wurde, wurde die Geschwindigkeit der Fehlerbehandlung untersucht. Dazu wurden erst die Laufzeiten für den fehlerfreien Fall mit 6 bzw. 8 Places gemessen, anschließend wurde der Benchmark mit 8 Places ausgeführt und 2 Places sind abgestürzt, nachdem sie gestohlene Tasks weitergeleitet haben. In Tabelle 6.3 kann man ablesen, dass die reine Berechnungszeit (process) sich um 2 Sekunden erhöht, was der Laufzeit mit 6 Places entspricht. Die Abstürze sind relativ früh passiert, um sicherzustellen, dass die Fehlerbehandlung vor Beendigung des Programms auf allen Places ausgeführt wurde. Geht man von einem Absturz kurz vor Ende der Laufzeit aus, muss man mit einer höheren Laufzeit rechnen, da zu diesem Zeitpunkt eine höhere Anzahl von Berechnungen erneut ausgeführt werden müssen. In zukünftigen Arbeiten könnten an dieser Stelle weitere Optimierungen vorgenommen werden.

	6 Places	8 Places	8 Places FT
Setup	0,1208	0,2208	0,2548
Process	8,0037	6,0125	8,0255
Reduce	0,1060	0,2910	0,21614

Tabelle 6.3: Laufzeit in Sekunden - Fehlerbehebung

7 Zusammenfassung und Ausblick

Diese Masterarbeit hat die Übertragung eines fehlertoleranten Algorithmus für Fork/Join Programme auf reduktionsbasierten Taskpool beschrieben. Dafür wurde das Konzept der lokalen Fehlerbehebung [KKM17] in das GLB Framework für APGAS implementiert.

Die lokale Fehlerbehebung beschreibt Datenstrukturen, die einen Stehlbaum abbilden und Sicherungskopien von versendeten Tasks speichern. Aus diesen Datenstrukturen lassen sich bei einem Absturz von einem oder mehreren Rechenknoten die verloren gegangenen Tasks und Verbindungen von Knoten wiederherstellen.

Das Hauptproblem lag in der unterschiedlichen Struktur der Taskbearbeitung. Während Tasks in Fork/Join Programmen Kindertasks erzeugen, die ihre Ergebnisse an die Eltern zurückgeben, so sind die Tasks bei GLB unabhängig voneinander. Auch die Ergebnisberechnung ist durch eine kommutative und assoziative Reduktion unabhängig von der Reihenfolge der Abarbeitung. Beide Algorithmen setzen auf Work-Stealing zur Lastenbalancierung. Bei Fork/Join wird nur ein Task gestohlen, jedoch versendet GLB TaskBags, die mehrere Tasks enthalten. Dies musste bei der Sicherung und Wiederherstellung von Tasks berücksichtigt werden.

Zum Schluss wurden Perfomancetests ausgeführt, um die Implementierung mit einer weiteren fehlertoleranten Variante und dem Standard GLB Framework zu vergleichen. Es zeigte sich, dass die Implementierung im fehlerfreien Fall weniger Overhead produziert. Bei einem weiteren Test wurde die Geschwindigkeit der Fehlerbehandlung getestet. Bei einem Absturz von 2 Places bei einer Gesamtzahl von 8 Places, entsprach die Laufzeit ungefähr der von 6 Places.

In weiteren wissenschaftlichen Arbeiten könnten die im Kapitel Implementierung beschrieben Probleme gelöst und weitere Benchmarks implementiert werden. Außerdem

könnten zum Beispiel Zwischenergebnisse in das GLB Framework eingefügt werden, um den Speicherverbrauch der Fehlertoleranz zu verringert und im Fehlerfall weniger Tasks wiederherstellen zu müssen.

Quelltextverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[HAZ] HAZELCAST, INC.: *Hazelcast IMDG - The Leading Open Source In-Memory Data Grid*. `http://hazelcast.org`, Abruf: 01.12.2017

[KKM17] Kestor, Gokcen; Krishnamoorthy, Sriram; Ma, Wenjing: Localized Fault Recovery for Nested Fork-Join Programs. In: *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International* IEEE, 2017, S. 397–408

[OHL+06] Olivier, Stephen; Huan, Jun; Liu, Jinze; Prins, Jan; Dinan, James; Sadayappan, P; Tseng, Chau-Wen: UTS: An unbalanced tree search benchmark. In: *LCPC* Bd. 4382 Springer, 2006, S. 235–250

[PF17] Posner, Jonas; Fohry, Claudia: Fault Tolerance for Cooperative Lifeline-Based Global Load Balancing in Java with APGAS and Hazelcast. In: *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International* IEEE, 2017, S. 854–863

[SKK+11] Saraswat, Vijay A.; Kambadur, Prabhanjan; Kodali, Sreedhar; Grove, David; Krishnamoorthy, Sriram: Lifeline-based global load balancing. In: *ACM SIGPLAN Notices* Bd. 46 ACM, 2011, S. 201–212

[Tar15] Tardieu, Olivier: The APGAS library: Resilient parallel and distributed programming in Java 8. In: *Proceedings of the ACM SIGPLAN Workshop on X10* ACM, 2015, S. 25–26

[X10] X10: *Introducing X10*. `http://x10-lang.org/articles/79.html`, Abruf: 01.12.2017

Anhang: Quelltext CD