UNI KASSEL
VERSITÄT

Universität Kassel

Bachelorarbeit
zum Erlangen des akademischen Grades Bachelor of Science

Fehlertolerante Taskpools in der parallelen Programmiersprache X10

Vorgelegt im
Fachbereich Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik

Jonas Posner
30203660

Kassel, 04. April 2014

Erstgutachterin: Prof. Dr. Claudia Fohry
Zweitgutachter: Prof. Dr. Gerd Stumme

Selbständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Überlegungen sind als solche gekennzeichnet. Die Arbeit wurde noch keiner anderen Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Kassel, den 04.04.2014

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen der Sprache X10 **3**

2.1 Allgemeines . . . 3

2.2 APGAS . . . 4

2.3 Fehlertoleranz-Modi . . . 5

2.4 Grundlegende Sprachkonstrukte . . . 5

3 Stand der Forschung zu fehlertoleranten Algorithmen **9**

3.1 Hadoop . . . 9

3.2 Weitere Arbeiten . . . 11

4 Erklärung des verwendeten UTS-Algorithmus **13**

5 Fehlertolerante Algorithmen für UTS **18**

5.1 Verteilte Backups . . . 19

5.2 Zentrales Backup . . . 24

6 Implementierung der Algorithmen in X10 **26**

6.1 Gemeinsamkeiten . . . 26

6.2 Verteilte Backups . . . 28

6.3 Zentrales Backup . . . 31

7 Performancevergleich und Diskussion **33**

8 Zusammenfassung und Ausblick **38**

Literaturverzeichnis **II**

Anhang: Quellcode auf CD **V**

1 Einleitung

Die Entwicklung von Prozessoren konzentriert sich seit Jahren darauf in einem Prozessor mehrere Kerne unterzubringen, anstatt wie früher lediglich den Takt eines einzelnen Kerns zu erhöhen. Um mehrere Kerne gleichzeitig effektiv zu nutzen, werden parallele Programme bzw. Algorithmen benötigt. Besonders wichtig sind diese in großen Rechenzentren, in denen mit sehr vielen miteinander verbundenen Servern oder Clustern parallel gearbeitet wird.

Parallele Programmierung bringt aber nicht nur Vorteile gegenüber der seriellen, sondern birgt in sich auch Schwierigkeiten für den Programmierer und der Entwicklung der Programmiersprachen. Dazu gehört die Aufgabe Algorithmen fehlertolerant gegenüber ausfallenden Knoten zu entwerfen und umzusetzen. Wird dies nicht berücksichtigt und nur ein Knoten stürzt während der Ausführung ab, bricht das gesamte Programm ohne verwertbares Ergebnis ab. Die Bedeutung dieses Falls ist nicht zu unterschätzen, da laut [9] die Chance auf ein erfolgreiches Ausführen eines 24 Stunden dauernden Algorithmus auf 1000 Knoten, die jeweils über eine mittlere Betriebsdauer zwischen Ausfällen (Mean Time Between Failures, kurz MTBF) von 6 Monaten verfügen, geringer als 1% ist.

Die vorliegende Bachelorarbeit beschäftigt sich mit der Weiterentwicklung eines vorhandenen parallelen Algorithmus mit dem Ziel Fehlertoleranz gegenüber ausfallenden Knoten zu integrieren. Dabei werden verteilte Taskpools verwendet. Bei einem Taskpool arbeiten Threads oder Prozesse Tasks ab, welche in einer Warteschlange gespeichert sind. Aus Tasks können neue Tasks entstehen, sodass anfangs nicht bekannt ist, wie viele Tasks zu bearbeiten sind. In der Regel gibt es deutlich mehr Tasks als Threads.

Da sich Fehlertoleranz auf verschiedene Arten mit jeweiligen Stärken und Schwächen implementieren lässt, wurden in dieser Bachelorarbeit zwei

Algorithmen entwickelt. Die Algorithmen wurden erst theoretisch ausgearbeitet und danach praktisch umgesetzt. Die konkrete Implementierung erfolgte in der noch relativ jungen Programmiersprache X10 [7]. Der anzupassende Taskpool-Algorithmus lag bereits in X10 vor. Leider beinhaltete er einen kleinen Fehler, der zur Folge hatte, dass eine wichtige Funktionalität nicht korrekt funktionierte. Dieser Fehler musste daher vor der eigentlichen Arbeit gefunden und behoben werden. Bei der Implementierung der Algorithmen wurde der Schwerpunkt auf die Kommunikation zwischen den Knoten (in X10 Places genannt) gelegt, während die Synchronisation innerhalb der Places weitgehend unberücksichtigt blieb und in zukünftigen Arbeiten hinzugefügt werden sollte. X10 bot sich für die Bachelorarbeit an, da dort das notwendige Feature der Fehlertoleranz im Release 2.4.1 vom Dezember 2013 bereitgestellt wurde. Nach der Umsetzung beider Algorithmen wurden die Programme auf dem Hochleistungsrechner der TU Darmstadt [3] getestet in bezüglich ihrer Performance miteinander verglichen.

Insgesamt zeigten die Messungen, dass beide Algorithmen bei vertretbarem Aufwand Fehlertoleranz beherrschen. Dieser Mehraufwand beträgt zwischen 18% und 63%. Aufgrund der unzureichenden Intra-Place-Synchronisation haben die Resultate allerdings nur einen vorläufigen Charakter.

Im nachfolgenden Kapitel 2 werden Grundlagen zur Programmiersprache X10 erläutert, soweit sie für die Implementierung in Kapitel 6 benötigt werden. Daraufhin gibt Kapitel 3 einen Überblick über diverse veröffentlichte fehlertolerante parallele Algorithmen. Im Kapitel 4 wird der anzupassende Basisalgorithmus beschrieben. Die für die Fehlertoleranz nötigen Anpassungen werden in Kapitel 5 zuerst theoretisch dargelegt, bevor die konkrete Umsetzung im Kapitel 6 erläutert wird. Kapitel 7 beschreibt die Performancemessungen und diskutiert die Ergebnisse. Das Fazit in Kapitel 8 fasst die gesamte Arbeit kurz zusammen und gibt einen Ausblick.

2 Grundlagen der Sprache X10

2.1 Allgemeines

X10 ist eine objektorientierte, parallele Programmiersprache. Sie ist klassenbasiert und bietet Einfachvererbung sowie einen Garbage Collector. Eine wichtige Zielplattform für X10 sind Hochleistungsrechenzentren, wobei diese unterschiedliche Berechnungseinheiten besitzen dürfen (sogenannte NUCC, Non-Uniform-Cluster-Computing Systeme). Die Programmiersprache ist folglich stark auf Parallelität ausgerichtet und für bis zu 100.000 Hardware-Threads ausgelegt.

Entwickelt wird X10 seit 2004 von IBM und einigen Universitäten, finanziert wurde die Entwicklung unter anderem von der DARPA (Defense Advanced Research Projects Agency). X10 befindet sich noch in der aktiven Entwicklung, sodass in naher Zukunft weitere Änderungen, Optimierungen und neue Features hinzukommen können. Die Entwicklung der Sprache verfolgt ähnliche Ziele wie die der Programmiersprache Chapel [2] und konkurriert mit ihr.

Die aktuelle stabile Version von X10 ist 2.4.2 aus dem Februar 2014. Diese Version wurde anfangs für die vorliegende Bachelorarbeit verwendet. Leider fiel im Rahmen der ersten Tests auf, dass sie einen Fehler enthält. Mit diesem Fehler wäre die Entwicklung der fehlertoleranten Algorithmen nicht realisierbar gewesen. Daher wurde im Rahmen dieser Bachelorarbeit ein offizieller Fehlerreport [5] erstellt. Der Fehler wurde nach kurzer Zeit von einem X10-Entwickler behoben. Deswegen wurde für die Entwicklung der Algorithmen die aktuellste X10-Version vom 21.02.2014 aus dem offiziellen SVN-Repository [4] kompiliert und verwendet.

Eines der Hauptziele von X10 besteht darin die parallele Programmierung der oben angesprochenen Systeme für den Programmierer deutlich zu vereinfachen.

Dadurch sollen mehr Programmierer für die parallele Programmierung gewonnen werden. Der Name X10 weist auf dieses Ziel hin und bedeutet, dass die Produktivität der parallelen Programmierung 10 Mal so effektiv wie mit herkömmlichen parallelen Sprachen gestaltet werden soll. Um die Erlernbarkeit zu verbessern, ist die Syntax von X10 stark an die Syntax des weitverbreiteten Java angelehnt. Ein relativ großer Unterschied zwischen Java und X10 ist allerdings, dass es in X10 keine Threads gibt, sondern nur leichtgewichtige Activities. Nähere Details zu den Sprachkonstrukten werden im Unterabschnitt 2.4 erläutert.

Für X10 existieren zwei Compiler. Der erste Compiler basiert auf C++ (native X10), der zweite auf Java (managed X10). Ein in X10 geschriebenes Programm kann also in C++ oder Java kompiliert werden. Außerdem kann beim Kompilieren angegeben werden, welche Art der Datenübertragung verwendet werden soll. Zur Auswahl stehen unter anderem die Optionen sockets und MPI bereit.

2.2 APGAS

APGAS steht für Asynchronous Partitioned Global Address Space [16] und ist ein Programmiermodell. Es ergänzt das ältere und bekanntere PGAS-Modell um Asynchronität und wurde explizit für X10 entwickelt. PGAS wird von anderen parallelen Programmiersprachen wie zum Beispiel Fortress und UPC benutzt.

Bei dem APGAS-Modell wird der globale Adressraum logisch unterteilt, sodass jeder Prozess seinen eigenen lokalen Speicherbereich besitzt, aber auch auf nicht lokale Speicherbereiche von anderen Prozessen zugreifen kann. Der lokale Zugriff auf den eigenen Speicher ist im Vergleich zum entfernten Zugriff schneller. In bestimmten Situationen ist der entfernte Zugriff allerdings unerlässlich. Der Programmierer ist für die Verteilung der Daten zuständig und bestimmt damit auch darüber wie oft in seinem Algorithmus ein entfernter Zugriff entsteht.

Im APGAS-Modell ist es außerdem möglich während der gesamten Laufzeit des Programms neue Activities dynamisch und asynchron zu erzeugen. So kann der

Programmierer sehr flexibel Algorithmen entwerfen.

2.3 Fehlertoleranz-Modi

Bei X10 gibt es verschiedene Arten wie eine Fehlertoleranz gegenüber ausfallenden Knoten umgesetzt wird. Dies kann über die Umgebungsvariable `X10_RESILIENT_MODE` (Typ Integer) eingestellt werden. Dabei können folgende Optionen ausgewählt werden:

- 0: Das Programm wird ohne Fehlertoleranz ausgeführt.
- 1: Place 0 ist für die Fehlertoleranz zuständig. Dies ist vergleichbar mit dem Master-Slave-Prinzip von Hadoop (siehe auch 3.1). Diese Option ist momentan am stabilsten und wird von den X10-Entwicklern empfohlen.
- 2: Verteilte Fehlertoleranz. Die Fehlertoleranz hängt nicht von einem Place ab, sondern verteilt sich auf mehrere Places. Diese Option ist schon verwendbar, befindet sich allerdings noch im Entwicklungsstadium und funktioniert daher nicht stabil.
- 3: Basiert auf Apache ZooKeeper. Diese Option ist noch in der Planungsphase und momentan nicht einsatzbereit.

2.4 Grundlegende Sprachkonstrukte

In diesem Abschnitt werden die für die Arbeit relevanten Sprachkonstrukte von X10 kurz beschrieben. Als Bezugspunkt dient die bekannte Syntax von Java. Die Beschreibungen basieren auf der Version 2.4.2 aus dem Februar 2014.

Place

Ein Place ist eine lokale Recheneinheit, das heißt typischerweise ein konkreter Prozessor. Ein Place kann entweder physikalisch sein oder auch simuliert werden.

Mit einem `at` kann auf einen gewünschten Place zwecks Ausführung von Anweisungen gewechselt werden. Über eine einfache Schleife kann man über alle vorhandenen Places iterieren und dort gegebenenfalls Activities ausführen: `for (place in Place.places())`.

Activity

Bei X10 gibt es keine Threads wie in Java, da diese von den X10-Entwicklern als schwerfällig angesehen werden. Daher ist in X10 eine leichtgewichtige Variante implementiert, die sogenannten Activities. Die einzelnen Activities besitzen keine konkreten Namen. Innerhalb einer Activity kann mit dem Schlüsselwort `here` geprüft werden, auf welchem Place sich die Activity befindet.

Async

Mit einem `async` wird eine neue Activity asynchron gestartet. So blockieren sich die neue Activity und die initialisierende Activity nicht gegenseitig und können parallel laufen. Zum Beispiel kann eine Activity auf Place 0 vollkommen unabhängig von einer Activity auf Place 1 gestartet, ausgeführt und beendet werden.

Finish

Mit der Anweisung `finish` wird festgelegt, dass mehrere asynchrone Activities gegenseitig auf sich warten. Erst nachdem alle Activities (auch eventuell rekursiv gestartete Activities) beendet sind, werden die nachfolgenden Anweisungen ausgeführt.

Deklaration von Variablen

Variablen können mit den Schlüsselworten `val` oder `var` deklariert werden. Beispiele:

```
val i = 5;
var i: Long = 5;
```

Deklarieren mit `val` ist dabei vergleichbar mit einem `final` in Java. Die Variable darf nur einmal initialisiert werden, kann danach nicht mehr geändert werden und der Variablentyp muss bei einer direkten Initialisierung anders als bei Java nicht zwingend angegeben werden. Eine `var`-Variable ist eine Variable mit den üblichen Eigenschaften. Bei einem Place-Wechsel mittels `at` wird eine `val`-Variable auf den neuen Place kopiert und der Wert kann dort verwendet werden. Dieser Kopiervorgang geschieht mit einer `var`-Variable nicht. Bei dem Kopiervorgang handelt es sich um ein sogenanntes „deep copy". Durch das Kopieren ist die Variable auf dem neuen Place unter dem gleichen Namen ansprechbar und verwendbar, Änderungen sind am Ursprungs-Place also nicht sichtbar.

Rail

Rails sind gleichzusetzen mit Arrays in Java. Der größte Unterschied besteht darin, dass Rails als Indextyp Long verwenden und nicht Integer.

Exception

Eine Exception ist ähnlich wie in Java definiert. Wie in Java können Exceptions mit einem `try-catch`-Block aufgefangen werden. Für die vorliegende Bachelorarbeit ist die sogenannte DeadPlaceException besonders wichtig. Diese wird erzeugt, wenn ein Place beispielsweise auf Grund eines Hardwareausfalls abstürzt und ist für die gewünschte Fehlertoleranz essenziell.

PlaceLocalHandle

Ein `PlaceLocalHandle` ist ein Verweis auf Place-lokale Informationen. Dafür muss pro Place ein Objekt einer nutzerdefinierten Klasse angelegt werden, auf das man mittels des `PlaceLocalHandle` schnell zugreifen kann.

Atomic

Mit einem `atomic`-Block können Anweisungen atomar ausgeführt werden, `atomic` kennzeichnet somit einen kritischen Abschnitt. Die `atomic`-Blöcke sind für den Zugriff auf gemeinsame Daten nötig.

3 Stand der Forschung zu fehlertoleranten Algorithmen

Zur Thematik der fehlertoleranten Algorithmen gibt es bereits einige veröffentlichte Arbeiten, die jedoch nicht direkt auf den in dieser Arbeit behandelten Taskpool-Algorithmus anwendbar waren. Im nachfolgenden Unterkapitel wird Hadoop vorgestellt und mit X10 verglichen. Anschließend wird ein Überblick über weitere relevante Publikationen gegeben.

3.1 Hadoop

Ein interessantes und verbreitetes System ist Apache Hadoop [1]. Hadoop ist ein Framework von Tools, welche zur Unterstützung der Ausführung von Applikationen auf sehr großen Computerclustern dient. Es ist vollständig Open Source und unter der bekannten Apache Lizenz zugelassen. Der verwendete Algorithmus basiert auf dem MapReduce Algorithmus von Google [11]. MapReduce kann sehr große Datenmengen und Berechnungen auf viele Systeme aufteilen und somit parallelisieren. Die sogenannten Map-Instanzen führen ihre zugeteilten Berechnungen durch und speichern die Ergebnisse auf lokalen Zwischenspeichern ab. Nachdem alle Map-Berechnungen fertig gestellt sind, berechnet ein Reduce-Prozess das Endergebnis aus diesen Zwischenspeichern.

Hadoop verwendet zur Verwaltung der Knoten das Master-Slave-Prinzip. Dabei hat der Masterknoten unter anderem die Aufgabe die Daten und Berechnungen auf die Slaves aufzuteilen. Die Berechnungen werden anschließend von den Slaves durchgeführt. Der Masterknoten ist der einzige Knoten, der Informationen darüber besitzt, welche Datenblöcke auf welchem Slaveknoten gespeichert sind.

Hadoop bietet Fehlertoleranz gegenüber ausfallenden Map-Instanzen. Diese

Toleranz wird durch eine Datenredundanz sichergestellt. Das heißt, dass die zu verteilenden Daten nicht nur auf einem Knoten gespeichert werden, sondern in der Standardkonfiguration auf drei Knoten. So ist eine Datenredundanz gewährleistet und es können die Daten eines ausgefallenen Knotens wiederhergestellt werden. Um einen Knotenausfall zu bemerken, müssen alle Knoten (Slaves) mit dem verwaltenden Knoten (Master) regelmäßig kurzen Kontakt aufnehmen. Kommt dieser Kontakt eine gewisse Zeit lang nicht zu Stande, wird der entsprechende Knoten nicht mehr verwendet und die zugehörigen Daten werden von dem Backup eingespielt. Dies verwaltet der Masterknoten. Da dem Masterknoten so eine elementare und zentrale Rolle zukommt und eine funktionierende Fehlertoleranz vom Masterknoten abhängt, wird dieser vollständig von einem anderem zusätzlichen Knoten gesichert. Dieser Knoten springt nur ein, wenn der Masterknoten ausfällt.

Hadoop wird heute schon von mehreren großen und bekannten IT-Unternehmen produktiv eingesetzt. Dazu zählen unter anderem Twitter, Facebook und eBay.

Unterschiede zwischen Hadoop und X10

Wie oben angesprochen spielt in Hadoop der Masterknoten eine zentrale und elementare Rolle für die Fehlertoleranz. Bei X10 gibt es verschiedene Arten wie eine Fehlertoleranz erreicht wird (siehe 2.3). Da momentan Modus 1 empfohlen wird, ist die Fehlertoleranz ähnlich wie in Hadoop umgesetzt. Allerdings ist in der Zukunft damit zu rechnen, dass auch die Optionen 2 und 3 einen stabilen Status erreichen. Wenn dies der Fall ist, könnte X10 Vorteile gegenüber Hadoop bekommen.

Ein weiterer großer Unterschied besteht im Konzept der Taskpools. Bei Hadoop werden die Aufgaben auf die Slaveknoten verteilt und dort abgearbeitet. In X10 ist es möglich einen verteilten Taskpool zu verwenden. Auf den Places können die dortigen Worker ihre Teile des Taskpools abarbeiten, aber auch zur Laufzeit

selber erweitern. Dieses Erweitern geschieht, wenn sich aus abgearbeiteten Tasks neue Tasks ergeben. Dies ist momentan bei Hadoop so nicht möglich.

Somit haben Hadoop und X10 viele ähnliche Ziele und Features, aber auch einige wesentliche Unterschiede.

3.2 Weitere Arbeiten

In letzter Zeit sind vermehrt interessante wissenschaftliche Paper über die Thematik der Fehlertoleranz in parallelen Programmen erschienen. Dieses Kapitel gibt einen kurzen Überblick über einige dieser Paper, erhebt aber keinen Anspruch auf Vollständigkeit.

Balaji et. al. geben in [13] einen umfangreichen Überblick über viele unterschiedliche fehlertolerante Lösungen. Diese werden in drei Kategorien unterteilt: Hardware, Systemsoftware und Applikation. Die Kategorien werden jeweils einzeln ausführlich beleuchtet, dabei werden Probleme mit passenden Lösungswegen beschrieben.

Vergleichsweise viele Veröffentlichungen ([12], [14], [18]) beschäftigen sich mit fehlertoleranten Algorithmen für parallele Matrix-Berechnungen. Die dafür entwickelte Fehlertoleranz-Techniken sind allerdings nicht direkt auf eine Fehlertoleranz in verteilten Taskpools übertragbar.

Dinan et. al stellen in [10] eine Variante eines fehlertoleranten Taskpools vor. Das Besondere daran ist, dass der Taskpool mit Hilfe einer verteilten Tasküberwachung ausgewählte Tasks neu starten kann. Im Gegensatz zur klassischen checkpoint/restart-Technik verhält sich der Mehraufwand zum Wiederherstellen proportional zur Fehlerrate und nicht zur Systemgröße. Mit der erarbeiteten Methode kann ein System mehrere ausgefallene Knoten unbeschadet überstehen. Um die Verwaltungsdaten möglichst gering zu halten, dürfen allerdings Tasks keine eigenen Tasks erzeugen. Dies ist ein essenzieller Unterschied zu dem verwendeten Algorithmus dieser Bachelorarbeit. Des Weiteren verwendet

das beschriebene System zusätzliche Knoten, die nur einspringen, wenn Knoten ausfallen. Auch dies wird in dieser Bachelorarbeit in dieser Form nicht verwendet. Messungen in [10] ergaben, dass der Mehraufwand bei ausfallenden Knoten bei lediglich 10% liegt.

4 Erklärung des verwendeten UTS-Algorithmus

Der Kern der vorliegenden Bachelorarbeit handelt von der Anpassung eines konkreten Programms zur Lösung des relativ bekannten Benchmark-Problems UTS. Die Abkürzung UTS steht dabei für Unbalanced Tree Search. Damit die vorgenommenen Modifikationen zur Fehlertoleranz (siehe Kapitel 5 und 6) besser verständlich werden, wird in diesem Abschnitt zuerst das verwendete Basisprogramm erläutert. Unter [6] ist der gesamte Quelltext des originalen UTS-Benchmarks in mehreren Programmiersprachen (unter anderem MPI, OpenMP, Chapel, X10) verfügbar. Der Benchmark ist speziell dafür gedacht Techniken zur Lastbalancierung in parallelen Systemen miteinander zu vergleichen sowie durch eigene Implementierung des Benchmark parallele Programmiersprachen zu bewerten.

UTS beinhaltet das vollständige Durchlaufen eines zur Laufzeit dynamisch generierten unbalancierten Baums. Dabei wird die Anzahl der Knoten gezählt. Die Eigenschaften des Baumes können mit Hilfe von Übergabeparametern beeinflusst werden. Es handelt sich unter anderem um folgende Merkmale:

- Startwert für den root-Knoten,
- Maximale Anzahl von Kindern, die ein Knoten haben darf,
- Verzweigungsfaktor,
- Maximale Tiefe,
- Baumtyp.

Der Baum wird also erst während der Ausführung aus den Parametern berechnet. Zur Berechnung der einzelnen Baumknoten verwendet der Benchmark den

verbreiteten SHA1-Algorithmus. Aus Hash-Werten können die Kindknoten mit Hilfe des SHA1-Algorithmus eindeutig berechnet werden. Dabei entsteht ein unbalancierter Baum, bei dem vor der kompletten Ausführung die jeweilige Knotenzahl der Teilbäume nicht bekannt ist. Somit wird die Verteilung der zu berechnenden Baumknoten auf die physikalischen Knoten zu einer Herausforderung und würde ohne Lastverteilung zu einer ungleichmäßigen Auslastung der Knoten führen. Die verwendete Lastverteilung ist demzufolge maßgeblich wie gut ein System im Laufzeitvergleich bei diesem Benchmark abschneidet.

In dieser Bachelorarbeit wird als Basis nicht der originale UTS-Algorithmus eingesetzt, sondern eine angepasste Variante von Claudia Fohry und Jens Breitbart [8]. Diese Variante basiert auf X10 und verwendet einen selbst implementierten Taskpool auf Nutzer-Level zum Verwalten der Tasks. Eine nähere Beschreibung befindet sich im folgenden Abschnitt.

UTS mit Taskpool

In der vorliegenden X10-Implementierung wird zur Verwaltung der Tasks und zur Lastverteilung ein verteilter Taskpool verwendet. Daher besitzt jeder Worker seine eigene Warteschlange mit Tasks, kann bei Bedarf aber auch von anderen Warteschlangen Tasks stehlen. Ein Task repräsentiert in dieser UTS-Variante einen Baumknoten, der Informationen über seine Kindknoten enthält. Nach der Berechnung der Kindknoten ist der Task vollständig bearbeitet und die Kindknoten befinden sich in Form von Tasks im Taskpool. In den folgenden Unterabschnitten werden die einzelnen X10-Klassen und Strukturen und deren Funktionalität detaillierter beschrieben.

UTS

Dies ist die Hauptklasse des vorliegenden Programms [8]. Hier werden die Übergabeparameter eingelesen und in einer Struktur des Typs `Paras` gespeichert. Danach wird der root-Baumknoten, eine Struktur des Typs `TreeNode`, initialisiert. Zusätzlich wird `PlaceLocalHandle` angelegt und auf allen Places mit Objekten des Typs `InfosPerPlace` initialisiert. So kann jeder Place auf seine eigenen Variablen zugreifen.

Jetzt erzeugt der root-Baumknoten seriell seine Kind- und Enkelkindknoten und verteilt die Enkel fair auf alle verfügbaren Worker. So sind alle Worker Taskpools mit Tasks gefüllt.

Nach der Initialisierungsphase werden alle Worker auf ihren jeweiligen Places asynchron gestartet und fangen nun parallel an ihre Taskpools abzuarbeiten. Nachdem alle Worker-Activities beendet sind, wird die Anzahl der Baumknoten mit einem selbstgeschriebenen Reduce zusammengerechnet. Daraufhin wird das errechnete Ergebnis und die benötigte Zeit auf der Konsole ausgegeben.

Paras

Diese Struktur dient lediglich zum Speichern der Parameter. Im Falle unzureichender Übergabeparameter werden die restlichen Variablen über einen entsprechenden Konstruktor mit Standardwerten initialisiert.

InfosPerPlace

`InfosPerPlace` wird für den `PlaceLocalHandle` verwendet. Hier werden für jeden Place die übergebenen Parameter und die Anzahl der bereits berechneten Baumknoten gespeichert. Zusätzlich wird ein Array vom Typ `SplitQueue` angelegt. Das Array bekommt die Länge der Anzahl der Worker pro Place. In diesem Array wird für jeden Worker eine eigene Struktur des Typs `SplitQueue` angelegt. Diese Struktur stellt alle benötigten Variablen und Funktionen zur

Abarbeitung des Taskpools bereit.

TreeNode

Die Struktur `TreeNode` repräsentiert einen Baumknoten. Sie bietet Variablen zum Speichern der Tiefe und des Hashwertes des Baumknotens. Zusätzlich werden Funktionen zum ersten Initialisieren des Taskpools und zur Expansion eines Knotens bereitgestellt. Bei der Expansion werden Funktionalitäten des SHA1-Algorithmus verwendet.

SplitQueue

In der Klasse `SplitQueue` sind die Variablen und Funktionen für die Verwendung des Taskpools implementiert. Da jedem Worker ein Objekt des Typs `SplitQueue` zugeordnet ist, speichert eine Variable die zugehörige ID des Workers. Außerdem stellt diese Klasse ein Array mit Objekten des Typs `TreeNode` bereit. Diese `TreeNodes` repräsentieren die noch offenen Tasks und müssen von dem zugeordneten Worker bearbeitet werden. Das Array ist unterteilt in einen privaten und öffentlichen Bereich. Die Unterteilung ist über entsprechende Variablen realisiert, die die Bereiche definieren.

Zuerst arbeiten die Worker ihren privaten Bereich im eigenen Array ab. Ist dieser Bereich erschöpft, transferiert der Worker Tasks mit Hilfe der Funktion `tryAcquire` aus dem eigenen public-Bereich in den private-Bereich. Sobald auch der public-Bereich abgearbeitet ist, also `tryAcquire` erfolglos verlief, wird die Funktion `tryStealLocal` aufgerufen. `tryStealLocal` versucht aus dem öffentlichen Bereich andere Worker offene Tasks zu stehlen. Ist auch dies ohne Erfolg, wird versucht Arbeit von anderen Places zu beschaffen. Dabei versucht ein arbeitsloser Worker zyklisch von anderen Workern auf fremden Places offene Tasks zu stehlen. Dieses Feature wird mit der Funktion `tryStealRemote` umgesetzt und bewirkt eine dynamische Lastverteilung zwischen den Places. Wenn ein Worker

keine zu berechnenden Knoten mehr besitzt,und der Versuch auf fremden Places etwas zu stehlen erfolglos war, wird dieser beendet.

SHA1Rand

Die X10-Struktur `SHA1Rand` bindet benötigte C-Header und Klassen in das Projekt ein. Somit können Funktionen aus den C-Klassen des SHA1-Algorithmus verwendet werden.

5 Fehlertolerante Algorithmen für UTS

In diesem Kapitel werden zwei fehlertolerante Algorithmen beschrieben, die im Rahmen dieser Bachelorarbeit für das Programm aus Kapitel 4 entwickelt wurden. Die Umsetzung im Quellcode wird im Anschluss in Kapitel 6 erläutert.

Beide Algorithmen beginnen nach der Initialisierungsphase, also wenn sich die Initialtasks bereits in den jeweiligen Taskpools befinden. Die Initialisierungsphase wurde ausgeklammert, da ihre Laufzeit vernachlässigbar ist und im Fehlerfall ein Neustart des gesamten Programms in Kauf genommen werden kann.

Bei der Entwicklung der Algorithmen wurde davon ausgegangen, dass in der Regel nur wenige Places während der Laufzeit abstürzen.

Um Fehlertoleranz zu erreichen, werden in regelmäßigen Abständen Sicherungskopien aller wesentlichen Daten von den Places erstellt. Damit kann gewährleistet werden, dass bei einem Place-Ausfall keine relevanten Daten verloren gehen. Diese Sicherungskopien können auf verschiedene Arten implementiert werden. Im Rahmen dieser Bachelorarbeit wurden zwei Arten entwickelt, welche in den nachfolgenden Unterkapiteln vorgestellt werden. Sie verwenden als Grundidee ein zentrales bzw. ein verteiltes Backup.

X10 besitzt unterschiedliche Modi um eine Ausfallsicherheit umzusetzen (siehe Kapitel 2.3). Bei beiden Algorithmen wird Modus 1 verwendet, da er momentan am stabilsten läuft und von den X10-Entwicklern empfohlen wird. Der Modus birgt den Nachteil, dass Place 0 nicht ausfallen darf, da eine solche Exception nicht aufgefangen werden kann. In diesem Fall würde das Programm abstürzen.

Einschränkung

Die Entwicklung der beiden Algorithmen im Rahmen dieser Bachelorarbeit beschränkt sich auf die Kommunikation zwischen den Places, die sogenannte

Inter-Place-Kommunikation. Damit die Algorithmen lückenlos korrekt arbeiten, muss allerdings auch die Synchronisation verschiedener Activities innerhalb eines Places, die sogenannte Intra-Place-Kommunikation, fehlerfrei arbeiten. Aus Zeitgründen wurde diese bei der Bachelorarbeit ausgeklammert.

Die Intra-Place-Kommunikation erfordert ähnlich viel Entwicklungszeit und -aufwand wie die Inter-Place-Kommunikation. Durch die in Abschnitt 5.1 und 5.2 beschriebene Funktionalität werden zeitweise neue zusätzliche Activities auf den Places gestartet. Diese müssen mit den ursprünglichen Activities synchronisiert werden. X10 bietet für kritische Abschnitte ein `atomic` (siehe 2.4), jedoch keine selbst benannten `atomic`-Blöcke (wie zum Beispiel in OpenMP). Um zu vermeiden, dass alle Zugriffe auf gemeinsame Variablen in den gleichen `atomic`-Blöcken erfolgen und sich somit gegenseitig blockieren, müsste mit `lock`-Variablen gearbeitet werden.

5.1 Verteilte Backups

In diesem Abschnitt wird die erste Variante des Algorithmus beschrieben. Zuerst werden die Funktionalitäten des Algorithmus erklärt. Da einige kritische Situationen auftreten können, werden diese im Anschluss erläutert.

Die Backups der einzelnen Places werden nicht auf einem zentralen Place gespeichert, sondern jeder Place hält zusätzlich zu seinem eigenen Taskpool auch mindestens ein (und höchstens zwei) konsistente(s) Backup(s) eines anderen Places. Auf jedem Place können mehrere Worker arbeiten, wobei die Anzahl der Worker per Übergabeparameter festgelegt wird und auf den verschiedenen Places gleich ist. Die Worker speichern ihre relevanten Variablen jeweils unabhängig voneinander. Auf Grund dieser Tatsache sichert jeder Worker auch einen Worker eines anderen Places, siehe Abbildung 5.1.

Standardmäßig sichert jeder Place zyklisch seinen Vorgänger (Place z sichert also Place z - 1, Place 0 sichert den letzten Place). Auch wenn ein Place ausfällt,

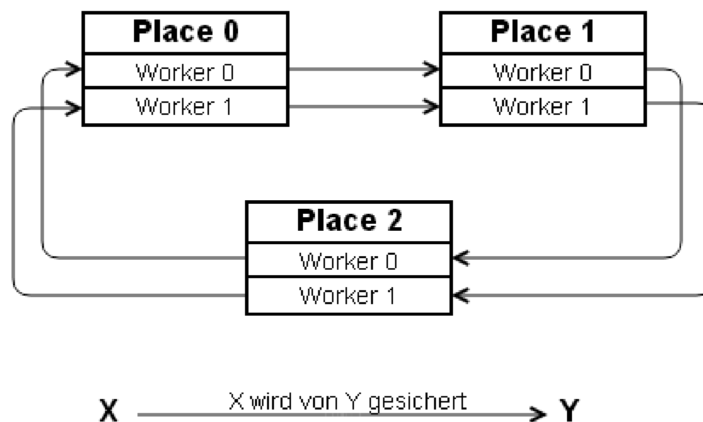

Abbildung 5.1: Backupzyklus

wird diese Reihenfolge logisch wieder hergestellt: Angenommen Place *z* fällt aus, dann wird *z - 1* danach auf *z + 1* gesichert. Um von Beginn an ein konsistentes Backup von jedem Place zu haben, wird bereits vor dem Starten der einzelnen Worker ein Backup erstellt. Dies geschieht unkompliziert beim Initialisieren der Taskpools. So bekommt dort der Place *z* einfach noch zusätzlich sämtliche Tasks, die auch Place *z - 1* zugewiesen bekommt. Diese werden getrennt von den eigenen Tasks abgespeichert. Es gibt also drei unabhängige Taskpools: für die eigenen Tasks, das aktuelle Backup und das vorige Backup. Damit wird erreicht, dass immer ein valides Backup vorhanden ist, auch wenn der zu sichernde Place während eines Sicherungsvorgangs ausfällt. In diesem Szenario wäre zwar ein Backup invalide, das andere aber valide.

Für das regelmäßige Backup ist Worker 0 der jeweiligen Places zuständig. Worker 0 initiiert ein Backup, welches aber alle Worker eines Places sichert. In welchem Rhythmus, also nach jeweils wie vielen berechneten Tasks, dies geschehen soll, kann über einen Übergabeparameter eingestellt werden.

Dabei wird immer abwechselnd in die zwei verschiedenen Backup-Variablen gesichert, um die oben beschriebene Integrität zu gewährleisten. Es wird kein inkrementelles Backup erstellt, da dies aufgrund der vielen Änderungen – aus

Tasks werden neue Tasks erzeugt – eine zu hohe Komplexität hätte.

Da nur Worker 0 das Backup des gesamten Places anstößt, wäre denkbar, dass es längere Zeiträume ohne Sicherungen gibt. Dies wäre der Fall, wenn Worker 0 bereits beendet ist und die restlichen Worker noch arbeiten. Da aber durch die Funktionalität des lokalen Stehlens alle Worker eines Places ungefähr gleichzeitig ihre Taskpools fertig bearbeitet haben, kommt es nicht zu dieser Situation. In einer typischerweise kurzen Zeit zwischen der Beendigung von Worker 0 und den restlichen Workern ist es nicht unbedingt notwendig ein neues Backup zu erstellen.

Entferntes Stehlen

Eine Besonderheit stellt die Funktion des entfernten Stehlens dar, bei der ein Worker Tasks von einem anderen Worker auf einem anderen Place stiehlt. Durch diese Funktionalität kann leicht eine Inkonsistenz entstehen. Ein Backup von Place z wird beispielsweise inkonsistent, wenn Place x dort etwas stiehlt und kurze Zeit später Place z abstürzt ohne in der Zwischenzeit sein Backup zu aktualisiert zu haben. In diesem Falle enthält das Backup von z offene Tasks, die allerdings auch auf x vorhanden sind. Diese werden dann doppelt berechnet und das Endergebnis wird in der Folge falsch sein. Um diese Problematik zu lösen, erstellen beide beteiligten Places nach erfolgreichem Stehlen jeweils ein neues Backup.

Der zweite Fall von Inkonsistenz, der bei einem entfernten Stehlen auftreten kann, ist ein Absturz von Place x während er die gestohlenen Tasks von z in seinen eigenen Taskpool übernimmt. In diesem Moment wurden die Tasks im Place z schon entfernt und im Backup von x sind sie noch nicht vorhanden. Um diesem Fall vorzubeugen, hält der bestohlene Worker vorerst die gestohlenen Tasks in einem temporären Array vor. Bei einem Absturz von x während des Stehlens kann z auf die ausgelöste Exception reagieren und die Tasks aus dem temporären Array wieder in seinen richtigen Taskpool zurück kopieren. So können keine Tasks

verloren gehen.

Unkritisch ist dagegen der Fall, dass Place x von Place z Tasks stehlen möchte und Place z schon bei der Auswahl der zu stehlenden Tasks abstürzt. In diesem Fall erhält Place x keine Tasks von seinem Stehlversuch, da Place z eine Exception ausgelöst hat. Place 0 reagiert auf die Exception entsprechend und initiiert, dass Place $z + 1$ das Backup von z einspielt. Da x keine Tasks von z stehlen konnte, ist das eingespielte Backup valide.

Reaktion auf Absturz

Bei einem Absturz wird eine Exception ausgelöst, aus der zu erkennen ist, welcher Place abgestürzt ist. Da alle Worker auf ihren jeweiligen Places von Place 0 aus gestartet wurden, wird die Exception immer an Place 0 weitergeleitet. Place 0 ist daher für den Aufruf der Methode zur Exception-Behandlung zuständig.
In dieser Methode wird zuerst eine Liste der aktuellen Places aktualisiert, damit kein Place mehr auf den toten Place fälschlicherweise zugreifen kann. Danach wird das Backup des toten Places z direkt auf dem entsprechenden Backup-Place $z + 1$ eingespielt. Das heißt, dass alle offenen Tasks in den privaten Taskpool kopiert werden und die Zahl der bereits berechneten Tasks mit der entsprechenden Variable des Backups addiert wird. Dies erfolgt durch eine zusätzliche Activity auf Place $z + 1$.
Da der abgestürzte Place ein Backup von Place $z- 1$ hielt, bekommt $z - 1$ einen neuen Backup-Place zugewiesen. Daher wird ein Backup von $z - 1$ auf seinen neuem Backup-Place, $z + 1$, initiiert. Anschließend ist wieder ein vollständig konsistenter Zustand hergestellt.

Problemsituationen

Es gibt einige Situationen, bei denen die entwickelte Fehlertoleranz versagt und das Programm ohne verwertbares Ergebnis abstürzt. Diese treten allerdings nur

selten auf, da zwei in Verbindung stehende Places in einem kurzem Zeitabstand hintereinander ausfallen müssen, was statistisch unwahrscheinlich ist.

- Die erste Problemsituation entsteht, wenn ein Place z abstürzt und Place $z + 1$ beim darauf folgenden Einspielen des passenden Backups selber abstürzt. Dann ergibt sich eine Konstellation, bei der Daten von Place z und ihr Backup verloren gehen. Da kein weiteres Backup existiert, sind diese Daten endgültig verloren und das Programm bricht an dieser Stelle ohne ein Ergebnis ab. Eine mögliche Lösung dieses Problems wäre, nicht nur eine Sicherheitskopie jedes Places anzulegen, sondern zwei oder mehr. Da die Wahrscheinlichkeit von zwei oder mehr Place-Abstürzen allerdings gering ist, wurde bei der Entwicklung davon abgesehen mehrere Sicherungskopien pro Place zu implementieren.

- Eine weitere Problemsituation ergibt sich, falls Place z abstürzt und bevor ein neues Backup von $z - 1$ komplett angelegt wurde, Place $z - 1$ auch noch abstürzt. Da zusammen mit z auch das ursprüngliche Backup von $z - 1$ verloren gegangen ist, sind auch hier Daten unwiderruflich verloren. Lösen ließe sich dieses Problem ebenfalls mit mehreren Sicherheitskopien, analog zur oben beschriebenen Lösung.

- Des Weiteren kann eine Problemsituation in der Funktion des entfernten Stehlens auftreten. Beispielsweise stiehlt Place x bei Place z Tasks. Nach dem erfolgreichen Stehlen führen normalerweise beide Places ihre Backuproutine aus um ihre Backups zu aktualisieren. Wenn aber eines dieser Backups wegen einem Absturz von Place z fehlschlägt, beinhaltet das Backup von z bereits gestohlene Tasks, die im Zweifelsfall doppelt berechnet werden. Mit einigem Aufwand wäre es auch möglich diesen Fall fehlerfrei zu behandeln. Dies würde allerdings den Rahmen dieser Bachelorarbeit sprengen.

- Eine nicht auffangbare Problemsituation ist, wie schon anfangs erwähnt, ein Ausfall von Place 0. Darauf kann im verwendeten Modus nicht reagiert werden und das Programm stürzt sofort ohne verwertbares Ergebnis ab.

5.2 Zentrales Backup

In diesem Abschnitt wird der zweite entwickelte Algorithmus beschrieben. Er verwendet statt des zuvor beschriebenen verteilten Backups ein zentrales und ist daher leichter zu implementieren. Der größte Unterschied zwischen den beiden Algorithmen besteht darin, dass bei diesem Algorithmus alle Sicherheitskopien auf Place 0 getätigt werden. Place 0 wurde für diese Aufgabe ausgewählt, da dieser aufgrund des verwendeten Modus nicht abstürzen kann ohne dass das Programm sowieso „verloren“ ist.

Auch hier wird bereits in der Initialisierungsphase das erste Backup jedes Places erstellt. Allerdings bekommt Place 0 alle Backups zugewiesen.

Place 0 hält also zusätzlich zu den ursprünglichen Variablen noch jeweils bis zu zwei komplette Backups von jedem Place. So wird auch hier in jedem Fall gewährleistet, dass immer ein valides Backup vorhanden ist.

Äquivalent zum verteilten Backup werden die Backups jeweils von den Workern mit der ID 0 auf den Places nach einer gewissen Anzahl berechneter Tasks initiiert. Da der Berechnungsaufwand eines Tasks von der Anzahl extrahierter Kindknoten abhängt und somit die Worker auf den verschiedenen Places unterschiedlich schnell ihre Tasks bearbeiten, kommt es normalerweise nicht zu der Konstellation eines Flaschenhalses, bei dem alle zeitgleich auf Place 0 sichern.

Die Besonderheiten des entfernten Stehlens müssen bei einem zentralen Backup ebenso beachtet und auch analog zum verteilten Backup gelöst werden.

Eine ausgelöste Exception durch einen abgestürzten Place landet auch hier bei Place 0, sodass dieser darauf reagieren kann. Demzufolge spielt er das entsprechende Backup in seinen eigenen Taskpool ein. Dieses Vorgehen ist deutlich

simpler als beim verteilten Backups. Es muss kein Backup eines anderen Place initiiert werden, da der Zustand jetzt schon wieder vollständig konsistent ist. Des Weiteren wird zur Verteilung die schon implementierte Lastbalancierung über das entfernte Stehlen genutzt.

Problemsituationen

Da die Komplexität des Algorithmus deutlich geringer ist als bei einem verteilten Backup, können auch weniger problematische Situationen entstehen. Wenn nach einem erfolgreichem Stehlen ein Place bei seinem planmäßig initiierten Backup abstürzt, können Tasks doppelt berechnet werden. Ansonsten nimmt selbstverständlich auch hier Place 0 eine unabdingbare Rolle ein. Im Falle eines Absturzes des Place 0 wird das Programm sofort beendet.

6 Implementierung der Algorithmen in X10

Dieses Kapitel beschreibt die konkrete Umsetzung der beiden beschriebenen Algorithmen aus Kapitel 5 in X10. Zuerst werden die Gemeinsamkeiten der beiden Algorithmen ausgeführt. Danach folgt eine Erläuterung der spezifischen Aspekte der Algorithmen in den jeweiligen Unterkapiteln.

6.1 Gemeinsamkeiten

Exception-Handling

Die erste simple Gemeinsamkeit bei beiden Algorithmen besteht darin, dass nach der Initialisierungsphase alle Place-Wechsel durch ein `at` nur noch innerhalb eines `try-catch`-Blocks erfolgen. Ein abstürzender Place erzeugt eine DeadPlaceException, die dann im zugehörigem `catch`-Block aufgefangen wird. So kann auf jeden abstürzenden Place (außer Place 0, siehe 2.3) reagiert werden.

Liste mit aktuell lebenden Places

Die von X10 bereitgestellte Funktionalität `Place.places()` enthält zwar alle Places, wird allerdings nicht automatisch aktualisiert und könnte daher auch bereits abgestürzte Places enthalten. Daher wird in der Klasse `InfosPerPlace` eine ArrayList `livePlaces` mit Objekten vom Typ `Place` angelegt. `livePlaces` wurde mit einer ArrayList realisiert, da diese leicht mittels `remove(object)` aktualisiert werden kann. Da während der Initialisierungsphase nicht auf Fehlertoleranz geachtet wird, wird diese ArrayList in der Methode `init` mit allen Places gefüllt. Somit besitzen nach der Initialisierungsphase alle Places in ihrem `InfosPerPlace`-Objekt eine Liste mit allen Places. Nach der Initialisierungsphase wird im restlichen Programm statt des klassischen `Place.places()` noch die

ArrayList `livePlaces` verwendet.

Die ArrayList muss nur aktualisiert werden, wenn ein Place im laufenden Betrieb abstürzt. Da in der Klasse `Uts` Place 0 über eine Schleife auf allen Places eine Activity startet, fängt im Falle eines Place-Absturzes Place 0 die geworfene DeadPlaceException auf. Im entsprechenden `catch`-Block wird dann die Methode `processException` aufgerufen. Diese Methode entfernt zuerst aus der eigenen `livePlaces` ArrayList den toten Place und ruft dann die Methode `refreshLivePlaces` auf. Diese iteriert über das aktuelle `livePlaces` und aktualisiert auf allen anderen Places die zugehörige ArrayList.

Durch ungünstige Zustände kann es theoretisch vorkommen, dass auf einem Place die `livePlaces`-Liste noch nicht aktualisiert wurde und dieser Place auf einen toten Place wechseln möchte. Daher wird an den relevanten Stellen vor einem `at` mit der Methode `isDead()` nochmals überprüft, ob der Place noch erreichbar ist. Ist er nicht mehr erreichbar, wird `livePlaces` sofort an dieser Stelle aktualisiert.

Backup-Rhythmus

Während der gesamten Laufzeit des Programms werden in regelmäßigen Abständen mit einer selbst geschriebenen Methode Backups erstellt. Die Abstände werden dabei über die bereits berechneten Tasks von Worker 0 des Places definiert. Sie können über den Übergabeparameter `z` beeinflusst werden. Dafür wurde in der Klasse `Paras` die Variable `periodBackup` vom Typ Long ergänzt.

Um die bereits berechneten Tasks mitzuzählen, wurde eine zusätzliche Variable `backupCount` vom Typ Long in der Klasse `InfosPerPlace` angelegt. `backupCount` wird zu Beginn mit 0 initialisiert und von Worker 0 bei jedem Aufruf der Methode `pop` inkrementiert. Nach dem Inkrementieren in `pop` wird geprüft, ob ein Backup erstellt werden soll. In der dann aufgerufenen Backup-Methode `makeBackupFull` wird `backupCount` anfangs auf 0 zurückgesetzt. Da `makeBackupFull` auch nach

einem erfolgreichen entfernten Stehlen aufgerufen wird (also unabhängig vom eigentlichen Rhythmus), verhindert dies, dass Backups unabsichtlich in einem unverhältnismäßig kurzen Abstand wiederholt ausgeführt werden.

Es kann außerdem der Fall eintreten, dass nach einem Absturz eines Places und dem erfolgreichen Einspielen des zugehörigen Backups Worker neugestartet werden müssen, da sie bereits ihre ursprünglichen Tasks fertig berechnet haben und schon beendet wurden. Auch hier wird `backupCount` auf 0 zurückgesetzt.

Anzahl berechneter Tasks

In der Klasse `SplitQueue` wurde eine Variable `workerN` vom Typ Long ergänzt. Diese zählt die Anzahl der bereits berechneten Tasks pro Worker mit. In der nicht fehlertoleranten Version des Programms wurde dies in der Methode `runWorker` der Klasse `Uts` realisiert. Um das Sichern zu vereinfachen, ist allerdings eine Variable in der Klasse `SplitQueue` vorzuziehen.

6.2 Verteilte Backups

In diesem Unterkapitel werden die relevanten Stellen aus dem ergänzten oder veränderten Quellcode für den Algorithmus des verteilten Backups beschrieben.

Da jeder Place einen anderen Place als Backup-Place zugeordnet bekommt, wurde in der Klasse `InfosPerPlace` eine entsprechende Variable `backupPlace` angelegt. Initialisiert wird sie in der bereits vorhandenen `init`-Methode. Da jeder Place x durch $x + 1$ (zyklisch) gesichert wird, kann `backupPlace` mit einem `here.next()` initialisiert werden.

Da bei diesem Algorithmus jeder Worker das Backup eines anderen Workers hält, wurden in der Klasse `SplitQueue` neue Variablen für diese Funktionalität angelegt. Dazu zählen `backupHead, backupTail, backupNpublic, backupNprivate` und `backupWorkerN` vom Typ Long und ein `Rail` mit Namen `backupSQ` und Elementtyp `TreeNode`. Die Länge des `Rails`

ist gleich der Länge des bereits vorhanden `Rails sq`. Diese Variablen existieren nochmals, mit dem Zusatz „Second". So kann ein Worker bis zu zwei vollständige Backups bereithalten. Um zu unterscheiden welches Backup aktuell und valide ist, wurde eine `Boolean`-Variable `validBackup` angelegt. Ist diese `true`, ist das erste Backup gültig, bei `false` das zweite.

In der Methode `initPool` der Klasse `TreeNode` werden die ersten Tasks auf die verschiedenen Places und Worker verteilt. Daher werden dort auch die Backup-Variablen initialisiert. Die Backup-Variablen von Place z enthalten nach der Initialisierung die gleichen Werte wie die normalen Variablen von Place z - 1. `validBackup` bekommt `true` zugewiesen, sodass deutlich wird, dass das erste Backup valide ist. So existiert von allen Workern ein valides Backup zum Zeitpunkt 0.

Erstellen eines Backups

In der Klasse `SplitQueue` wurde die Methode `makeBackupFull` ergänzt. Diese, wie auch alle weiteren Methoden, wird mit dem `PlaceLocalHandle` als Parameter aufgerufen, damit innerhalb der Methoden auf `InfosPerPlace` zugegriffen werden kann. Die Methode gibt bei Erfolg `true` zurück, ansonsten `false`. Da `makeBackupFull` immer ein komplettes Backup vom gesamten Place erstellt und nicht nur eines bestimmten Workers, sammelt die Methode anfangs alle relevanten Daten der lokalen Worker ein. Diese Daten werden in `Rails` geschrieben, welche mit dem Schlüsselwort `val` deklariert sind. Dadurch werden diese Rails bei dem Place-Wechsel auf `tp().backupPlace` mit kopiert. Die jeweiligen globalen Variablen sind mit einem `var` deklariert und werden daher nicht mit kopiert. Auf dem Backup-Place werden dann die `Rails` wieder passend auf die Worker aufgeteilt. Außerdem wird mit Hilfe von `validBackup` auf die richtige Position des Backups geachtet. Bei Erfolg wird `validBackup` negiert, damit das nächste Backup in die anderen Variablen geschrieben wird.

In der Klasse `SplitQueue` wurde außerdem die bereits vorhandene Methode `tryStealRemote` modifiziert. Mit dieser Methode versucht ein Worker von Place x offene Tasks auf einem anderen Place zu finden und zu stehlen. Um einen Place mit offenen Tasks zu finden, wird über die ArrayList `livePlaces` iteriert und mit einem `at` auf den Place gewechselt. Wenn auf einem Place z offene Tasks im public-Bereich gefunden werden, werden diese in ein temporäres `val Rail` namens `stolenArray` kopiert. Danach wird mittels `at` wieder auf x gewechselt und die gestohlenen Tasks werden in den Taskpool von x übernommen. Die erwähnten `at`-Blöcke befinden sich jeweils innerhalb eines `try-catch`-Blocks. Wenn während der Phase des Einspielens des `stolenArray` auf x eine DeadPlaceException auftritt, wird diese von z aufgefangen. In diesem Moment sind die gestohlenen Tasks in keinem richtigen Taskpool mehr vorhanden. Daher wird im `catch`-Block das `stolenArray` wieder in den Taskpool von z eingespielt.
Läuft dagegen alles ohne Komplikationen ab und das Stehlen war erfolgreich, wird auf beiden beteiligten Places die Methode `makeBackupFull` aufgerufen, um die Backups zu aktualisieren.

Einspielen eines Backups

Um nach einem Place-Ausfall das passende Backup einzuspielen, wurden in der Klasse `Uts` die Methoden `processException` und `goBackup` ergänzt. Die Methode `goBackup` wird mit der ID des toten Places als Parameter zeitlich nach den oben auf Seite 27 beschriebenen Schritten in `processException` aufgerufen. Da Place 0 immer die DeadPlaceExceptions auffängt, wird auch die Methode `goBackup` auf Place 0 ausgeführt. Daher wechselt `goBackup` zunächst auf den Place, der das Backup vom abgestürzten Place hält und startet dort eine neue Activity. Diese Activity kopiert die relevanten Tasks aus dem validen Backup in die privaten Taskpools der Worker. Außerdem werden die Variablen `workerN, head` und `nprivate` passend inkrementiert. Nach einem erfolgreichen Wiederherstellen des

Backups muss abschließend noch ein Backup des Places vor dem abgestürzten Place initiiert werden. Auch dies geschieht mittels eines `at`, welches sich in einem `try-catch`-Block befindet. Stürzt z - 1 während der Sicherung ab, wird das gesamte Programm im `catch`-Abschnitt beendet, da nicht mehr gewährleistet werden kann, dass das Gesamtergebnis korrekt ist.

6.3 Zentrales Backup

Dieses Unterkapitel beschreibt die Implementierung des zweiten Algorithmus, welcher ein zentrales Backup verwendet. Da dieser Algorithmus ähnlich wie der vorige umgesetzt wurde, wird bei der Beschreibung nur auf die Unterschiede der beiden Implementierungen eingegangen.

Der entscheidende Unterschied besteht darin, dass jetzt alle Places auf Place 0 gesichert werden. Daher brauchen die Worker von Place 0 entsprechende Variablen. Diese Variablen sind wie bei dem vorigen Algorithmus benannt. Da sie allerdings n-mal (für n Places) benötigt werden, wurde dies mit `Rails` umgesetzt. Im Konstruktor der Klasse `SplitQueue` werden die `Rails` mit der Länge n initialisiert. So kann Place 0 bis zu zwei Backups für jeden anderen Place speichern.

In der Methode `initPool` in der Klasse `TreeNode` wird auch bei diesem Algorithmus das erste Backup vor den ersten parallelen Berechnungen jedes Places geschrieben. Dafür musste der Quellcode vom vorigen Algorithmus nur leicht angepasst werden, sodass alle Sicherheitskopien auf Place 0 erstellt werden.

Auch die Methode `makeBackupFull` in der Klasse `SplitQueue` musste nur leicht verändert werden. Beim Backup muss lediglich in die richtigen Stellen der `Rails` geschrieben werden, anstatt in die klassischen Variablen des verteilten Backups.

Die Methode `tryStealRemote` wurde unverändert aus dem Quellcode des verteilten Backups übernommen.

Die Klasse `Uts` beinhaltet wieder die relevanten Funktionen `processException`

und `goBackup`. `processException` brauchte nicht verändert zu werden. In `goBackup` mussten dagegen nur wenige Änderungen vorgenommen werden. Dadurch ist die Komplexität der Methode etwas gesunken. Da Place 0 immer die DeadPlaceExceptions auffängt und bei diesem Algorithmus alle Sicherheitskopien hält, braucht `goBackup` nicht mehr den Place zu wechseln, um die richtige Sicherheitskopie wiederherzustellen. Stattdessen spielt Place 0 direkt aus der richtigen Position der `Rails` die benötigten Tasks in seinen Taskpool ein. Da bei einem Absturz von Place x das Backup von x - *1* nicht verloren geht, muss dementsprechend auch kein neues Backup von x - *1* angestoßen werden.

7 Performancevergleich und Diskussion

Nach der Entwicklung der Algorithmen wurde ein Performancevergleich zwischen dem ursprünglichen, nicht fehlertoleranten Algorithmus und den beiden entwickelten fehlertoleranten Algorithmen durchgeführt.

Dazu wurde ein Account auf dem Lichtenberg-Hochleistungsrechner der TU Darmstadt [3] beantragt und verwendet. Der Lichtenberg-Hochleistungsrechner steht für wissenschaftliche Arbeiten kostenfrei zur Verfügung und bietet momentan über 800 Rechenknoten. Diese bestehen aus verschiedener Hardware. Der MPI-Sektor macht mit 706 Knoten den mit Abstand größten Teil aus. Die Knoten dieses Sektors besitzen 2 Intel Xeon Prozessoren vom Typ E5-2670 mit jeweils 8 Rechenkernen, wobei Hyperthreading deaktiviert ist. Die gewünschten Programme können über ein Warteschlangensystem, das sogenannte Batch-System, ausgeführt werden. Daher ist der Cluster für eine performante Ausführung paralleler Programme, und somit auch für den Performancevergleich, prädestiniert. Zum Kompilieren von X10-Programmen wird der GNU C Compiler (GCC) benötigt, dieser liegt auf dem Cluster in Version 4.8 vor. Die Ausführung der Programme erfolgte nicht exklusiv, sodass auch noch anderen Programme auf den Rechenknoten laufen konnten.

Das Programm wurde bei allen Messungen mit den gleichen Übergabeparameter wie folgt gestartet:

```
./Uts t1 a0 d13 b4 r29 p3
```

Da die Ausführungszeit bei mehreren identischen Ausführungen leicht schwankt, wurde jeder Lauf fünf mal ausgeführt und aus diesen Ergebnissen ein Mittelwert gebildet. Die Zeitmessung erfolgte mit X10-Mitteln auf Place 0.

Es wurden Programmläufe mit drei, sechs und neun Places. Die Worker-Anzahl pro Place wurde konstant bei drei belassen. Der Rhythmus der jeweiligen

Backups wurde auf den empirisch ermittelten Standardwert 6000000 gesetzt. Um die Performance der beiden entwickelten Algorithmen zu vergleichen, wurden mehrere Situationen untersucht. Dazu wurde in den jeweiligen Algorithmen an gewissen Stellen mittels `System.killHere()` ein gewollter Place-Absturz herbeigeführt. Die verschiedenen Situationen werden in den folgenden Unterkapiteln beschrieben. Als primäres Maß wurde die verbrauchte Zeit verwendet. Am Ende fasst eine Gesamtübersicht die Ergebnisse zusammen.

Kein ausfallender Place

Zuerst wurde eine klassische Situation, bei der kein Place abstürzt, betrachtet. Dieser Fall tritt je nach Anzahl der Places in der Realität sehr oft auf und ist daher wichtig. Diese Untersuchung soll zeigen, wieviel Mehraufwand das zusätzliche Feature der Fehlertoleranz benötigt. Die Ergebnisse der Messungen sind in Tabelle 7.1. zu sehen. Daraus ist zu erkennen, dass

- der ursprüngliche Algorithmus bei drei Places eine vergleichsweise hohe Laufzeit hat. Diese könnte darauf zurückzuführen sein, dass in der Konstellation des betrachteten Baums die Funktion des entfernten Stehlens verhältnismäßig oft aufgerufen wird.
- der Algorithmus des verteilten Backups bei steigender Place-Anzahl ähnlich gut wie der nicht fehlertolerante Algorithmus skaliert. Beim zentralen Backup dagegen steigt die Ausführungszeit bei neun Places.
- alle Algorithmen bei sechs Places die kürzeste Laufzeit haben.
- die Fehlertoleranz bei sechs Places circa 18% (verteiltes Backup) und 63% (zentrales Backup) Mehraufwand benötigt.

Anzahl Places	Ursprünglicher Algo.	Verteiltes B.	Zentrales B.
3	19,57 sec.	14,10 sec.	20,10 sec.
6	8,71 sec.	10,29 sec.	14,26 sec.
9	9,49 sec.	12,88 sec.	21,34 sec.

Tabelle 7.1: Kein ausfallender Place

Ein ausfallender Place

Da bei getesteter Placeanzahl selten mehr als ein Place ausfällt, wurde nur diese Situation nachgestellt. Da ein Place-Ausfall während der gesamten Ausführung auftreten kann, wurden für den Performancevergleich zwei Situationen nachgestellt, die im folgenden als Average Case und Worst Case bezeichnet werden. Beide Situation wurden statisch im Quelltext mit einem `System.killHere()` realisiert, so dass die Situationen bei beiden Algorithmen vergleichbar sind.

Average Case

Der Average Case simuliert einen Place-Ausfall inmitten der Ausführung des Algorithmus und soll somit einen durchschnittlichen Fall darstellen. Tabelle 7.2. zeigt die Messergebnisse:

- Das verteilte Backup ist in allen Fällen etwas schneller als das zentrale. Dies dürfte an der Tatsache liegen, dass der Rechenaufwand beim verteilten Backup auf alle Places verteilt ist und sich nicht auf einen konzentriert.
- Beide Algorithmen brauchen ungefähr die gleiche Zeit zur Ausführung wie in der Situation, in der kein Place abstürzt.

Anzahl Places	Verteiltes B.	Zentrales B.
3	19,15 sec.	21,19 sec.
6	10,76 sec.	14,57 sec.
9	11,99 sec.	20,22 sec.

Tabelle 7.2: Average Case

Worst Case

Der Worst Case repräsentiert den Fall, dass ein Place nach allen Berechnungen der Tasks, aber vor der abschließenden Addition der Teilergebnisse, abstürzt. Die Werte sind in Tabelle 7.3. dargestellt:

- Die Skalierbarkeit ähnelt dem Average Case.
- Im Vergleich zum Average Case sind die Ausführungszeiten leicht gestiegen.

Anzahl Places	Verteiltes B.	Zentrales B.
3	19,02 sec.	22,78 sec.
6	12,57 sec.	16,91 sec.
9	12,52 sec.	22,68 sec.

Tabelle 7.3: Worst Case

Gesamtübersicht

Eine Gesamtübersicht der Tests ist in Abbildung 7.1. grafisch dargestellt. Dort sind die oben beschriebenen Eigenschaften nochmals deutlich zu erkennen:

- Bei sechs Places ist die Performance insgesamt am besten.

- Ohne Place-Absturz sind die fehlertoleranten Varianten langsamer als der ursprüngliche Algorithmus, allerdings in einem vertretbaren Ausmaß (verteiltes Backup: ca. 18%, zentrales Backup ca.63%).

- Der Algorithmus des verteilten Backups ist, wie zu erwarten war, schneller als der des zentralen Backups.

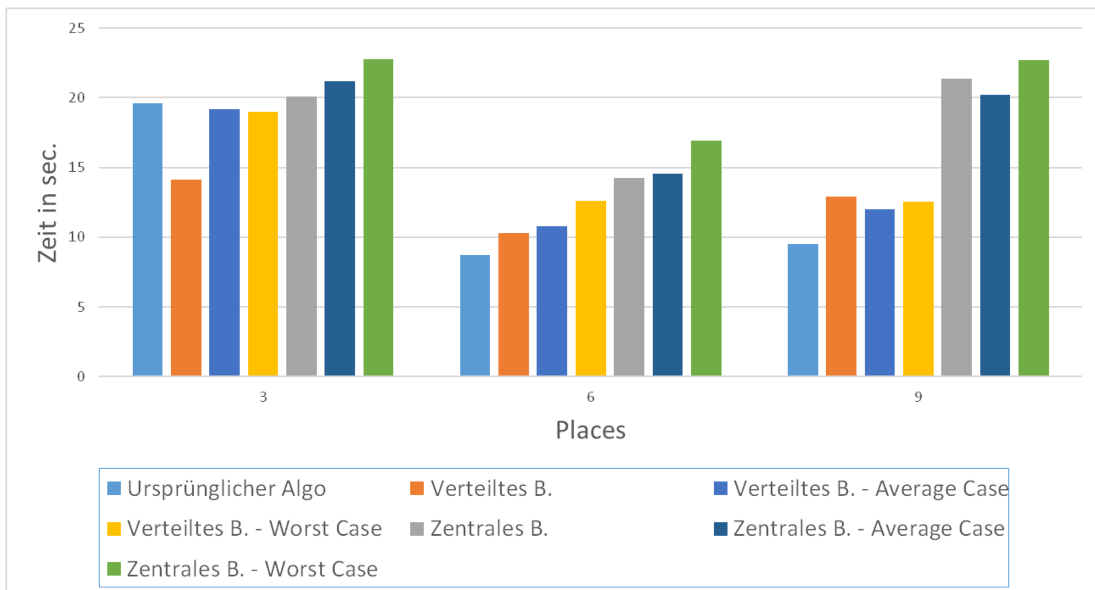

Abbildung 7.1: Balkendiagramm von allen Tests

8 Zusammenfassung und Ausblick

Gegenstand dieser Bachelorarbeit war die Erweiterung eines UTS-Algorithmus, der in X10 vorlag, um Fehlertoleranz gegenüber ausfallenden Knoten.

Das Hauptaugenmerk bei der Entwicklung lag auf der Inter-Place-Kommunikation. Es wurden zwei Algorithmen entwickelt, von denen einer zentrale Backups und der andere verteilte Backups verwendet. Die Intra-Place-Kommunikation musste aufgrund der knappen Zeit ausgeklammert werden. Der Performancevergleich ist aus diesem Grund nur von vorläufiger Natur, gibt allerdings einen ersten Eindruck über die Laufzeit der beiden Algorithmen: Der Algorithmus mit verteilten Backups hat lediglich 18% Overhead bei einer fehlerfreien Ausführung, der mit zentralem Backup dagegen 63%.

Des Weiteren konnte in der Bachelorarbeit durch die Entwicklung der Algorithmen das relativ neue Feature der Fehlertoleranz aus Sicht eines Programmierers bewertet werden. Das verwendetet Prinzip der Ausnahmebehandlung wird als intuitiv, leicht benutzbar und gut umgesetzt eingeschätzt. Da sich X10 noch in der Entwicklung befindet, konnte keine komplett fehlerfreie Version erwartet werden. Der Support der X10-Entwickler erwies sich aber als zeitnah und hilfreich.

Für spätere wissenschaftliche Arbeiten wäre die Weiterentwicklung der Algorithmen, speziell im Bereich Intra-Place-Kommunikation, und weitere Optimierungen mit Sicherheit interessant.

Literaturverzeichnis

[1] *Apache Hadoop Website.* http://hadoop.apache.org, Zugriff: Februar 2014.

[2] *Cray Chapel Website.* http://chapel.cray.com/, Zugriff: Februar 2014.

[3] *Der Lichtenberg-Hochleistungsrechner an der TU Darmstadt.* http://www.hhlr.tu-darmstadt.de/hhlr/lichtenberg/index.de.jsp, Zugriff: Februar 2014.

[4] *Offizielles SVN-Repository von X10.* https://svn.code.sourceforge.net/p/x10/code/trunk, Zugriff: Februar 2014.

[5] *Resilient X10: failure to reliably raise DPE when System.killHere is called.* http://jira.codehaus.org/browse/XTENLANG-3368, Zugriff: Februar 2014.

[6] *The Unbalanced Tree Search Benchmark.* http://sourceforge.net/p/uts-benchmark/wiki/Home, Zugriff: Februar 2014.

[7] *X10: Performance and Productivity at Scale.* http://x10-lang.org, Zugriff: Februar 2014.

[8] Claudia Fohry, Jens Breitbart: *Experiences with Implementing Task Pools in Chapel and X10.* 2013.

[9] David Cunningham, David Grove, Benjamin Herta Arun Iyengar Kiyokuni Kawachiya Hiroki Murata Vijay Saraswat Mikio Takeuchi Olivier Tardieu: *Resilient X10 - Efficient failure-aware programming.* ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), 2013.

[10] James Dinan, Arjun Singri, P. Sadayappan Sriram Krishnamoorthy: *Selective Recovery From Failures In A Task Parallel Programming Model.* CCGRID '10 Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Seiten 709–714, 2011.

[11] Jeffrey Dean, Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters.* Communications of the ACM, Seiten 107–113, 2004.

[12] Nawab Ali, Sriram Krishnamoorthy, Mahantesh Halappanavar Jeff Daily: *Multi-Fault Tolerance for Cartesian Data Distributions.* International Journal of Parallel Programming, Seiten 469–493, 2011.

[13] Pavan Balaji, Darius Buntinas, Dries Kimpe: *Fault Tolerance Techniques for Scalable Computing.* Scalable Computing and Communications: Theory and Practice., 2012.

[14] Peng Du, Aurelien Bouteiller, George Bosilca Thomas Herault Jack Dongarra: *Algorithm-based Fault Tolerance for Dense Matrix Factorizations.* PPoPP '12 Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, Seiten 225–234, 2012.

[15] Stephen Olivier, Jun Huan, Jinze Liu Jan Prins James Dinan P. Sadayappan Chau-Wen Tseng: *UTS: An Unbalanced Tree Search Benchmark.* LCPC'06 Proceedings of the 19th international conference on Languages and compilers for parallel computing, Seiten 235–250, 2006.

[16] Vijay Saraswat, George Almasi, Ganesh Bikshandi Calin Cascaval David Cunningham David Grove Sreedhar Kodali

Igor Peshansky Olivier Tardieu: *The Asynchronous Partitioned Global Address Space Mod.* 2012.

[17] Vijay Saraswat, Bard Bloom, Igor Peshansky Olivier Tardieu David Grove: *X10 Language Specification Version 2.4.* 2014.

[18] Wesley Bland, Peng Du, Aurelien Bouteiller Thomas Herault George Bosilca Jack J. Dongarra: *A Checkpoint-on-Failure Protocol for Algorithm-Based Recovery in Standard MPI.* Euro-Par'12 Proceedings of the 18th international conference on Parallel Processing, Seiten 477–488, 2012.

Anhang: Quellcode auf CD

Das ursprüngliche Programm, auf den die entwickelten Algorithmen basieren, wurde von Claudia Fohry und Jens Breibart entwickelt und unterliegt deren Copyright-Rechten: `http://www.uni-kassel.de/eecs/fachgebiete/plm/team/prof-dr-claudia-fohry.html`