UNIKASSEL
VERSITÄT

Fachbereich 16 Elektrotechnik/Informatik
Studiengang Informatik

Ein Framework für globale Lastbalancierung

Masterarbeit

Marco Postigo Perez
Matrikelnummer: 272 025 93

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

Kassel, den 20.07.2015

Selbständigkeitserklärung

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

Kassel, den 20.07.2015

Marco Postigo Perez

Inhaltsverzeichnis

1 Einleitung

Die parallele Programmierung spielt eine immer größere Rolle in der Softwareentwicklung. Dabei wird das zu bearbeitende Problem in kleinere Teilprobleme zerlegt und diese zur gleichen Zeit auf mehreren Prozessoren bzw. Rechenkernen ausgeführt. Auf diese Weise kann die Laufzeit eines Programms deutlich reduziert werden. Für besonders rechenaufwändige Probleme können Berechnungen auch auf mehreren miteinander verbundenen Rechnern (Computer-Cluster) ausgeführt werden. Jeder Computer eines Clusters ist in der Regel ein symmetrisches Multiprozessorsystem (SMP). Ein SMP besitzt mindestens zwei Prozessoren, wobei sich alle Prozessoren einen globalen Speicher teilen. Diese Speicherarchitektur ist auch als *Uniform Memory Access* (UMA) bekannt. Demgegenüber stehen Multiprozessorsysteme mit *Non-Uniform Memory Access* (NUMA). Hier hat jeder Prozessor seinen eigenen lokalen Speicher, der allerdings von anderen Prozessoren über einen gemeinsamen Adressraum angesprochen werden kann.

Leider bringt parallele Programmierung nicht nur Vorteile mit sich. Beispielsweise muss aufgrund von Datenabhängigkeiten und dem Zugriff auf gemeinsam verwendete Variablen der Programmablauf synchronisiert werden. Dadurch erhöht sich die Komplexität der Programme, was zu einer höheren Fehleranfälligkeit sowie einer erschwerten Wartbarkeit führt.

Beim Einsatz paralleler Programmiersprachen muss die Zielplatform beachtet werden. Auf einem Einprozessorsystem oder SMP können Daten über den gemeinsamen Speicher ausgetauscht werden. Bei der Entwicklung von Programmen für diese Systeme hat sich im Laufe der Zeit OpenMP etabliert. Implementierungen von OpenMP existieren in C, C++ und Fortran. Im Gegensatz dazu kann bei einem System mit verteiltem Speicher (Computer-Cluster) nicht auf lokale Daten von anderen Rechenknoten zugegriffen werden. Um auf einem solchen System Daten auszutauschen, müssen diese erst übertragen werden. Ein weit verbreiteter Ansatz dafür ist das Message Passing Interface (MPI), eine Schnittstellendefinition für den Nachrichtenaustausch zwischen den Rechenknoten. Dieser

Austausch von Nachrichten erschwert die Entwicklung erheblich. Mit dem Partitioned Global Address Space (PGAS) wurde ein Programmiermodell eingeführt, das dem entgegen wirken soll. Auf diesem Modell aufbauende Programmiersprachen (PGAS-Sprachen) verbergen die Komplexität der Kommunikation vor dem Entwickler. Infolgedessen kann ein Programm für verteilten Speicher genau so formuliert werden, wie für ein System mit gemeinsamen Speicher. Bekannte PGAS-Sprachen sind z.B. Unified Parallel C (UPC [8]), Chapel [1] und X10 [5].

Um die Laufzeit von parallelen Programmen zu verbessern, ist eine gleichmäßige Verteilung der Arbeitslast auf die zur Verfügung stehenden Prozessoren wichtig. Im Rahmen dieser Arbeit wurde dafür das von Saraswat et al. [18] beschriebene Programmiermuster einer globalen Lastbalancierung mit Chapel implementiert. In X10 existiert bereits eine Implementierung dieses Modells, das als Vergleichsbasis herangezogen wurde. Dabei wurden Performance, Usability sowie Stärken und Schwächen beider Programmiersprachen bewertet. Die X10-Implementierung [23] (im Folgenden auch als GLB_X10 bezeichnet) überzeugt durch gute Programmlaufzeiten aber auch durch die einfache Verwendung. Im Gegensatz dazu kann die Chapel-Implementierung (im Folgenden auch als GLB_CHPL bezeichnet) bezüglich Performance und Usability nicht mithalten. Beides leidet sehr unter noch nicht implementierten Funktionalitäten im Bereich der Objektorientierung. Das liegt vor allem daran, dass Chapel sich noch immer in der Entwicklungsphase befindet und objektorientierte Features bisher keine hohe Priorität hatten [4].

In Kapitel 2 werden zunächst einige Grundlagen näher erläutert. Diese umfassen unter anderem das Programmiermodell des Partitioned Global Address Space (PGAS) und die Programmiersprachen Chapel und X10. Im Anschluss daran wird in Kapitel 3 das Konzept der globalen Lastbalancierung aus [18] genauer beschrieben. Kapitel 4 beschäftigt sich mit der Umsetzung dieses Konzepts in Chapel. Hier wird zuerst die Struktur des entwickelten Frameworks anhand eines Klassendiagramms erörtert und darauf aufbauend ausgewählte Aspekte der Programmierung vorgestellt. Das fünfte Kapitel gibt einen Überblick über die verwendeten Benchmarks, die zum Testen der GLB-Implementierung verwendet wurden. Abschließend werden die Ergebnisse präsentiert und ausgewertet. Kapitel 6 enthält eine Zusammenfassung und einen Ausblick.

2 Grundlagen zu parallelen Programmiersprachen

2.1 Partitioned Global Address Space

Partitioned Global Address Space (PGAS) ist ein Programmiermodell, das für die parallele Programmierung entwickelt wurde. Dabei wird der gemeinsame Speicher in Abschnitte aufgeteilt und jeder dieser Abschnitte einer Gruppe von Prozessoren zugeordnet. Jeder Prozessor kann auf die Speicherbereiche anderer Prozessoren zugreifen, wobei der Zugriff auf lokalen Speicher schneller erfolgt als auf entfernten. Dadurch vereinfacht sich die Entwicklung paralleler Software erheblich. Im Programmcode müssen keine expliziten Nachrichten an andere Prozessoren gesendet werden, wie bspw. mit MPI. Stattdessen können Variablen auf entfernten Prozessoren direkt verwendet werden. Viele experimentelle parallele Programmiersprachen wie Chapel, UPC oder X10 basieren auf PGAS.

2.2 Einführung in Chapel und X10

Chapel und X10 wurden beide unter staatlicher Förderung des High Productivity Computing Systems (HPCS) Projekts der Defense Advanced Research Projects Agency (DARPA) in den USA entwickelt. An Chapel arbeitet Cray Inc. bereits seit 2003, wohingegen IBM 2004 mit dem Entwurf von X10 begonnen hat. Beide Sprachen sind Open-Source-Projekte. Chapel unterliegt der Apache v2.0 Lizenz[1] und X10 der Eclipse Public License 1.0[2]. Dadurch können weltweit Softwareentwickler bei der Weiterentwicklung der Sprachen helfen. Dies

1 http://www.apache.org/licenses/LICENSE-2.0.html
2 http://opensource.org/licenses/EPL-1.0

sind derzeit vorwiegend Universitäten und Hochschulen, aber auch Vertreter aus der Industrie.

Um die Produktivität bei der Entwicklung paralleler Software zu steigern, nutzen beide Sprachen unter anderem den Ansatz der Objektorientierung. Damit ist es möglich Probleme modular zu lösen und der geschriebene Programmcode kann einfacher wiederverwendet werden. Zugleich wird, wie zuvor beschrieben, von Hardware- und Kommunikationsdetails abstrahiert. Der Programmierer muss sich keine Gedanken um das verwendete Betriebssystem, die zur Verfügung stehenden Rechenknoten bzw. Prozessoren oder wie diese miteinander kommunizieren (Infiniband, Myrinet, UDP/TCP, etc.) machen. Ein weiterer wichtiger Punkt bei der Entwicklung der Sprachen war die Skalierbarkeit. Als Zielplatform sollten einfache Desktop-Computer und Laptops, aber auch Computer-Cluster sowie Hochleistungsrechner dienen. Außerdem sollte Chapel- und X10-Programmcode mit nur wenigen Änderungen auf Grafikprozessoren ausgeführt werden können (vgl. [21] und [22]). Seit 2013 wurde diese Unterstützung in Chapel vorerst aus dem offiziellen Release entfernt [3]. Des Weiteren war ein vorrangiges Ziel beider Sprachen, in Bezug auf Performance mit den derzeitigen parallelen Mainstream-Sprachen wie Cilk, MPI und OpenMP mitzuhalten.

X10 bietet darüber hinaus einen Garbage Collector (GC). Der GC überprüft ob Objekte im Programm noch (global) referenziert werden. Sollte das nicht der Fall sein, so werden diese Objekte automatisch gelöscht (vgl. [19], Kap. 4.4.5 und [16]). Chapel bietet derzeit keinen GC, weshalb der Entwickler sich selbst darum kümmern muss nicht mehr benötigte Objekte zu löschen (vgl. [11], Kap. 2.3.4). Daraus ergibt sich zum einen der Nachteil einer zusätzlichen Fehlerquelle. Beispielsweise kann ein Speicherleck entstehen, weil Objekte nicht gelöscht wurden. Zum anderen ist der positive Aspekt, dass ohne Garbage Collector kein extra Rechenaufwand nötig ist um Objekte auf noch bestehende Referenzen zu überprüfen und ggf. zu löschen.

Um verschiedene Aufgaben parallel zu bearbeiten existieren in X10 sog. `Activities` (Aktivitäten). Dabei stellt eine *Aktivität* einen leichtgewichtigen Thread dar, der jederzeit auf einem beliebigen Rechenknoten gestartet werden kann (vgl. [5], Kap. 2.3 + Kap. 14). In X10 werden Rechenknoten als Place bezeichnet, in Chapel als Locale. X10-Aktivitäten entsprechen in Chapel sog. Chapel-Tasks, die in Abschnitt 2.3 näher erläutert werden.

2.3 Wichtige Sprachkonstrukte in Chapel

Zum besseren Verständnis der GLB_CHPL-Implementierung in Kapitel 4 werden in diesem Abschnitt wichtige Sprachkonstrukte in Chapel behandelt. Aufgrund des begrenzten Rahmens der Arbeit konnte nicht auf alle Funktionalitäten bis ins Detail eingegangen werden. Daher sei für weitere Informationen auf die Spezifikation von Chapel [1] verwiesen. Auf eine detaillierte Einführung in X10 wird an dieser Stelle verzichtet, da kein Quelltext aus der X10-Implementierung vorgestellt wird.

Module

Module dienen zur Einteilung der Namensräume in Chapel. Damit ist es möglich Programme logisch zu strukturieren und damit die Lesbarkeit und Wartbarkeit zu verbessern. Außerdem können gleichnamige Methoden oder Klassen unterschieden werden, indem vor der Verwendung der Modulname mit angegeben wird (siehe Quelltext 2.1 und 2.2).

```
1 module A {
2   class C {}
3 }
4
5 module B {
6   class C {}
7 }
```

Quelltext 2.1: Beispiel: Module

```
1 use A;
2 use B;
3
4 var a : A.C = new A.C();
5 var b : B.C = new B.C();
```

Quelltext 2.2: Beispiel: Module #2

Klassen vs. Records

Für die objektorientierte Programmierung bietet Chapel Klassen und Records an. Von beiden kann man Instanzen bilden, die jeweils auf dem Locale liegen, auf dem die Instanziierung erfolgt. Klassen und Records unterscheiden sich hauptsächlich in der Art, wie sie gespeichert werden. Eine Variable, die auf eine Klasseninstanz verweist, enthält lediglich einen Zeiger auf den Speicherbereich, in dem das Objekt liegt. Eine Zuweisung auf eine neue Variable bewirkt nur, dass die Referenz gespeichert wird. Darüber hinaus wird für Funktionen, die ein Objekt als Parameter oder als Rückgabetyp besitzen, auch nur die Referenz übergeben bzw. zurückgegeben. Das hat vor allem Auswirkungen, wenn ein Objekt von einem entfernten Locale verwendet werden soll, da jeder Zugriff

auf Methoden oder Attribute des Objekts mit Kommunikationsaufwand verbunden ist. Demgegenüber enthalten Record-Variablen keinen Verweis auf ein Objekt, sondern direkt die zugehörigen Daten. Die Verwendung eines Records als Parameter oder Rückgabetyp bewirkt standardmäßig das Erstellen einer Kopie. Mit Hilfe von speziellen Schlüsselwörtern (`ref` und `const ref`) kann allerdings auch für Records eine Referenz statt einer Kopie übergeben werden (siehe [1], Kap. 13.5). In Quelltext 2.3 und 2.4 sind als Beispiel eine Klasse und ein Record aufgeführt. Beide enthalten die Definition eines Attributs `x` vom Typ *int* (Zeile 2) und einer Methode `doSomething` (Zeile 4). Die `doSomething`-Methode zeigt die Deklaration eines Rückgabewerts und Parameters. Der Parameter p ist vom Typ `object`, welcher in Chapel eine besondere Rolle spielt. Alle Klassen sind entweder explizit oder implizit von `object` abgeleitet.

```
1 class A {
2   var x : int = 5;
3
4   proc doSomething(p : object) : int {
5     return x;
6   }
7 }
```

Quelltext 2.3: Beispiel: Klasse

```
1 record A {
2   var x : int = 5;
3
4   proc doSomething(p : object) : int {
5     return x;
6   }
7 }
```

Quelltext 2.4: Beispiel: Record

Domain Maps

Domain Maps sind ein mächtiges Werkzeug in Chapel. Mit ihnen lässt sich bestimmen wie die Daten eines Arrays auf die Locales verteilt und wie sie im Speicher abgelegt werden. Im Folgenden werden ausschließlich die Möglichkeiten zur Verteilung der Daten auf die Locales betrachtet. Standardmäßig bietet Chapel drei Arten von Domain Maps: *Cyclic*, *Block* und *Block Cyclic*. Die Elemente werden dabei zyklisch, blockweise oder blockweise zyklisch verteilt. Die dazugehörigen Beispiele in den Abbildungen 2.1, 2.2 und 2.3 wurden aus einem Vortrag von Aroon Shama et al. [20] entnommen.

Taskparallelität

In Chapel existieren drei Anweisungen um Berechnungen parallel auszuführen: `begin`, `cobegin` und `coforall`. Jeder dieser Befehle erstellt eine bestimmte Anzahl Chapel-Tasks (CT), die zur Laufzeit den Threads des Betriebssystems zugeordnet werden. Dabei können

```
1 use CyclicDist;
2 var domain = {1..15};
3 var distribution = domain dmapped Cyclic(startIdx=domain.low);
4 var A: [distribution] int;
```

Quelltext 2.5: Beispiel: Zyklische Verteilung

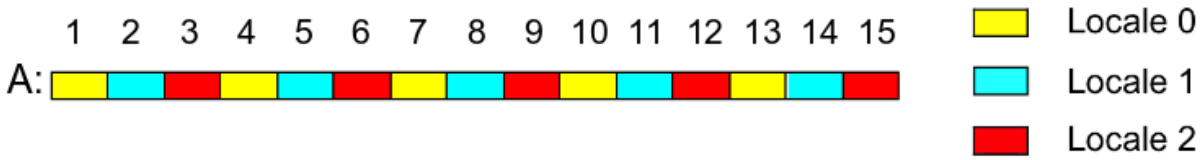

Abbildung 2.1: Beispiel: Zyklische Verteilung, entnommen aus [20]

```
1 use BlockDist;
2 var domain = {1..15};
3 var distribution = domain dmapped Block(boundingBox=domain);
4 var A: [distribution] int;
```

Quelltext 2.6: Beispiel: Blockweise Verteilung

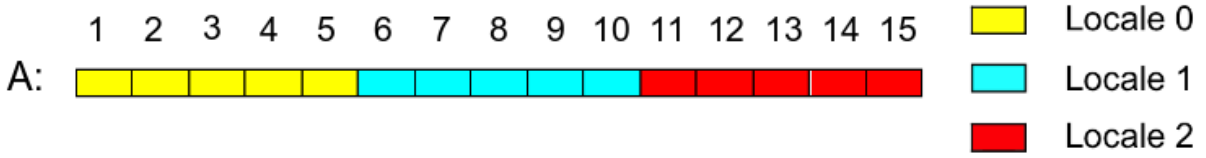

Abbildung 2.2: Beispiel: Blockweise Verteilung, entnommen aus [20]

```
1 use BlockCycDist;
2 var domain = {1..15};
3 var distribution = domain dmapped BlockCyclic(blocksize=3);
4 var A: [distribution] int;
```

Quelltext 2.7: Beispiel: Zyklische Blockweise Verteilung

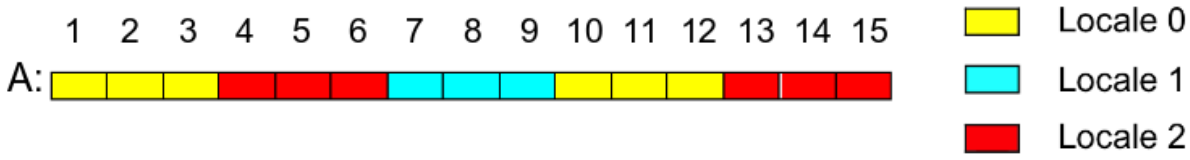

Abbildung 2.3: Beispiel: Zyklische Blockweise Verteilung, entnommen aus [20]

mehr CTs als Threads existieren. Die Zuweisung einer CT an einen Thread wird vom sog. *Tasking Layer* übernommen. Von diesem Layer existieren unterschiedliche Implementierungen, wobei der Standard Layer von der Zielplatform abhängt. Auf Windows ist das bspw.

`FIFO`[3]. Hier wird jedem Thread pro Zeitpunkt genau eine CT zugeordnet. Bevor diesem Thread nun eine neue CT zugeordnet werden kann, muss die derzeitige erst vollständig abgearbeitet worden sein. Demgegenüber steht der `qthreads`-Layer, welcher als Standard Layer für die meisten anderen Zielplatformen gewählt wird. Hier kann die Abarbeitung einer CT auch unterbrochen werden, um eine neue bzw. andere CT zu bearbeiten.

In Quelltext 2.8 ist ein Beispiel der `begin`-Anweisung zu sehen. Diese Anweisung erstellt immer genau eine CT, die alle in ihr enthaltenen Befehle ausführt. In diesem Fall wird `doSomething` und `doSomethingElse` in der gleichen CT ausgeführt. Der Befehl `cobegin` erstellt für jede enthaltene Anweisung eine CT. Im Codebeispiel 2.9 würden damit `doSomething` und `doSomethingElse` zwei unterschiedliche CTs darstellen. Quelltext 2.10 zeigt ein Beispiel der Verwendung von `coforall`. Die Variablen `a` und `b` sind dabei Arrays desselben Typs der Länge n. Die `coforall`-Anweisung erstellt nun genau n CTs, also für jede Iteration eine.

```
1 begin {
2   doSomething();
3   doSomethingElse();
4 }
```

Quelltext 2.8: Beispiel: begin

```
1 cobegin {
2   doSomething();
3   doSomethingElse();
4 }
```

Quelltext 2.9: Beispiel: cobegin

```
1 coforall i in 1..n do
2   a(i) = b(i);
```

Quelltext 2.10: Beispiel: coforall

Datenparallelität

In der Softwareentwicklung kommt es häufig vor, dass auf unterschiedlichen Daten dieselbe bzw. eine sehr ähnliche Operation ausgeführt werden muss. Die Ausführung lässt sich in vielen Fällen parallelisieren, was auch als Datenparallelität bezeichnet wird. Chapel bietet zu diesem Zweck die `forall`-Anweisung an.

3 Abkürzung für First In - First Out. Beschreibt ein allgeimeines Verfahren um Daten zu organisieren. Dabei werden die Daten, die zuerst gespeichert wurden, auch als erstes wieder entnommen.

```
1 forall i in 1..n do
2   a(i) = b(i);
```

Quelltext 2.11: Beispiel: forall

Quelltext 2.11 wurde aus Kapitel 25 der Spezifikation [1] entnommen. Wie in Quelltext 2.10 sind `a` und `b` Arrays der Länge n desselben Typs. Beim Kompilieren des Programms wird festgelegt wieviele CTs zur Ausführung der Schleife erstellt werden. Dabei wird vorausgesetzt, dass eine `forall`-Schleife auch sequentiell ausgeführt werden kann, also mit nur einer CT. Die Anzahl der Tasks hängt von dem Objekt ab, über das iteriert wird. In diesem Fall ist das Objekt vom Typ `range` (1..n), das alle Indizes im Bereich 1 bis n beinhaltet (inklusive). Eine verkürzte Schreibweise für den Quelltext in 2.11 ist eine einfache Zuweisung des Arrays `b` an `a` ($a = b$). Interessant wird es, wenn über eine Domain Map iteriert wird. Hier können die Elemente auf unterschiedlichen Locales verteilt sein. Die `forall` Schleife erstellt in diesem Fall für jeden Locale eine CT. Jeder dieser CTs kann wiederum weitere CTs für die Abarbeitung der lokal vorliegenden Elemente erzeugen [9].

Synchronisation

Die Synchronisation paralleler Tasks wird in Chapel über sog. Synchronisationsvariablen erreicht. Für die Deklaration bzw. Definition dieser Variablen existiert in Chapel die Konvention, dass der Name mit einem $-Zeichen endet. Synchronisationsvariablen können den Zustand *leer* oder *voll* annehmen. Ein lesender Zugriff auf eine volle Variable liefert den Wert an den aufrufenden Thread bzw. Prozess zurück und setzt den Zustand auf leer. Weitere Versuche die Variable zu lesen blockieren nun solange, bis ihr ein neuer Wert zugewiesen wird. Mit einer neuen Zuweisung wechselt der Zustand wieder auf voll. Weitere Versuche in eine bereits gefüllte Variable zu schreiben werden ebenfalls blockiert.

In Chapel werden zwei Arten von Synchronisationsvariablen zur Verfügung gestellt: `single` und `sync`. Im Gegensatz zur `sync`-Variable darf der `single`-Variable nur einmal ein Wert zugewiesen werden. Weitere Lesezugriffe ändern den Zustand einer solchen Variablen nicht mehr. Um mehr Kontrolle über die Synchronisierung zu erhalten, bieten die Variablen zusätzlich Methoden an. An dieser Stelle sollen nur diejenigen vorgestellt werden, die in der GLB_CHPL-Implementierung verwendet wurden:

- `readFF()`

  Blockiert bis die Variable voll ist und liest ihren Wert. Die Variable verbleibt nach dem Lesevorgang im Zustand voll.

- `readXX()`

  Nicht-blockierender Lesevorgang, bei dem die Variable im selben Zustand verbleibt. Sollte die Variable leer sein, so wird der letzte Wert zurückgegeben, den sie enthalten hat. Wenn der Variablen noch kein Wert zugewiesen wurde, wird der Standardwert zurückgegeben, mit dem jede Variable initialisiert wird (vgl. [1], Kap. 8.1.1).

- `writeFF(v: t)`

  Blockiert bis die Variable voll ist und weist ihr dann einen neuen Wert zu. Die Variable verbleibt nach dem Schreibvorgang im Zustand voll.

- `writeXF(v: t)`

  Nicht-blockierender Schreibvorgang, der einer Variablen einen neuen Wert zuweist, auch wenn sie bereits gefüllt ist.

- `isFull`

  Überprüft ob die Variable voll ist.

3 Globale Lastbalancierung

In der parallelen Programmierung werden Probleme meist in kleinere und möglichst unabhängige Teilprobleme bzw. Teilaufgaben (im Folgenden auch als Tasks bezeichnet) zerlegt. Diese Tasks werden dann auf mehreren Rechenknoten gleichzeitig bearbeitet, wobei die Abarbeitung einer Task ggf. noch weitere erzeugen kann.

Eine bewährte Strategie um Tasks gleichmäßig zu verteilen ist das sog. *Work Stealing* [7]. Bei diesem Verfahren werden die Tasks von sog. *Workern* bearbeitet. Jeder Worker hat einen Pool von Aufgaben. Ist dieser leer, so versucht der betroffene Worker Tasks von einem anderen Worker zu stehlen. Allerdings erweist sich dieser Ansatz als problematisch, falls das Programm auf verteiltem Speicher ausgeführt wird. Versuche zu stehlen verursachen einen hohen Kommunikationsaufwand, wodurch die Performance des Programms negativ beeinflusst wird. Dies gilt insbesondere wenn nur wenige Tasks vorliegen und dadurch unnötige Anfragen versendet werden. Darüber hinaus ist es meist nicht trivial festzustellen, ob alle Worker die ihnen zugewiesenen Tasks abgearbeitet haben und das Programm terminieren kann. Der Programmentwickler muss sich explizit darum kümmern und zusätzlichen Kommunikationsaufwand in Kauf nehmen.

In Saraswat et al. [18] wird der Ansatz des Work Stealings so erweitert, dass eine hohe Effizienz auf verteiltem Speicher erreicht werden kann. Zudem lässt sich das dort beschriebene Schema einfach in eine Programmbibliothek auslagern und kann damit wiederverwendet werden. Das beschriebene Konzept setzt voraus, dass während der gesamten Laufzeit des Programms maximal ein Thread pro Rechenknoten aktiv ist, welcher einen Worker realisiert. Dadurch können das Stehlen und die Verteilung von Tasks ohne zusätzliche Synchronisation stattfinden.

Lifeline-Graph

Ein zentraler Bestandteil des Konzepts [18] ist die Anordnung der Rechenknoten in einem sog. *Lifeline-Graphen*. Dabei handelt es sich um einen zusammenhängenden und gerichteten Graphen, bei dem jeder Knoten einen Rechenknoten bzw. Worker repräsentiert. Zudem stellen alle Kanten sog. Lifelines (Lebenslinien) dar. Wenn ein Worker keine Tasks mehr hat und auch keine weiteren gestohlen werden konnten, dann werden diese Lebenslinien aktiviert und der Worker wechselt in den Ruhemodus. Die Aktivierung der Lebenslinien findet nacheinander statt, gemeinsam mit dem Versenden des Steal-Requests an den zur Lebenslinie zugehörigen Knoten. Nach jeder gesendeten Stehlanfrage wird erst auf eine Antwort gewartet. Aus Sicht des anfragenden Knotens, werden die mit ihm verbundenen Knoten auch als Lifeline-Buddies bezeichnet. Falls das Stehlen von einem Lifeline-Buddy (LB) erfolgreich war, so wird die zugehörige Lebenslinie wieder deaktiviert und die Arbeit fortgesetzt. Ansonsten bleibt die Lebenslinie aktiv. Der Worker wechselt in den Ruhemodus, wenn keine Tasks von den LBs gestohlen werden konnten. Sollte nun einer der LBs wieder Tasks erhalten, so werden diese direkt über die aktivierten Lebenslinien weitergeleitet und der entsprechende Worker verlässt den Ruhemodus, um die Arbeit wieder fortzusetzen. Dabei wird die Lebenslinie wieder deaktiviert. Das Programm terminiert, wenn alle Lebenslinien im Lifeline-Graph aktiviert sind und sich folglich alle Worker im Ruhemodus befinden.

Die Hauptziele bei der Gestaltung des Lifeline-Graphen waren möglichst wenige ausgehende Kanten je Knoten sowie kurze Kommunikationswege. Damit sollten Tasks schnell auf die zur Verfügung stehenden Rechenknoten verteilt und das Terminieren des Programms früh erkannt werden. Darüber hinaus muss der Graph zusammenhängend sein, d.h. jeder Knoten muss von einem beliebigen anderen Knoten über eine oder mehrere Kanten erreicht werden können. Um die gewünschten Eigenschaften zu gewährleisten, wird jeder Knoten als eine Zahl zur Basis h mit z Stellen dargestellt. Für ein beliebiges h und z hat ein Knoten v beispielsweise die Form $v = (a_1, \ldots, a_z)$, mit $a_i \in \{0, \ldots, h-1\}$. Durch diese Schreibweise lassen sich die Knoten des Graphen in einem z-dimensionalen Koordinatensystem abbilden. Ein Knoten v hat nun genau dann eine ausgehende Kante zu $\hat{v}$, wenn die Manhatten Distanz (in Modulo h Arithmetik) zu $\hat{v}$ eins beträgt. Sei zum Beispiel $v = (a_1, \ldots, a_z)$ und $\hat{v} = (\hat{a}_1, \ldots, \hat{a}_z)$. Dann hat der Knoten v eine ausgehende Kante zu $\hat{v}$ genau dann, wenn ein $i \in \{1, \ldots, z\}$ existiert, so dass $v = (\hat{a}_1, \ldots, (\hat{a}_i + 1)\%h, \ldots, \hat{a}_z)$.

Struktur und Ablauf

Als Erstes muss auf jedem Rechenknoten ein Worker erstellt und die vorhandenen Tasks gleichmäßig auf diese verteilt werden. Allerdings müssen nicht alle Worker Tasks zugewiesen bekommen. Zum Beispiel könnte zunächst nur eine Task vorliegen, was dazu führt, dass zu Beginn auch nur ein Worker aktiv ist.

Ein aktiver Worker beginnt zunächst mit der Abarbeitung von Tasks, wobei immer höchstens n Tasks bearbeitet werden. Der Parameter n entspricht einer beim Programmstart festgelegten Konstante. Sollte ein Worker jedoch weniger als n Tasks haben, so werden auch nur diese abgearbeitet. Im Anschluss daran wird überprüft, ob andere Rechenknoten Anfragen zum Stehlen (im Folgenden auch Steal-Request) gestellt haben und ggf. Tasks an diese verteilt. Mit dem Konfigurieren des Wertes von n lässt sich also einstellen, wie oft eingehende Anfragen bearbeitet werden. Ist n sehr klein, so wird die Abarbeitung der Tasks häufig unterbrochen um ggf. wartende "Diebe" mit Tasks zu versorgen. Ist n sehr groß, werden Anfragen erst viel später bearbeitet.

Die Abarbeitung der Tasks, die Überprüfung auf eingehende Anfragen, sowie die Verteilung der Tasks werden solange wiederholt bis keine Tasks mehr vorliegen. In diesem Fall wird zuerst versucht von w zufällig ausgewählten Workern zu stehlen. Dabei wird der Worker (der stehlen möchte) jeweils solange blockiert, bis die Anfrage beantwortet wurde. Der Parameter w lässt sich ebenfalls beim Start des Programms festlegen.

Wenn das Stehlen der Tasks erfolgreich war, kann die Taskbearbeitung fortgesetzt werden. Ansonsten wird nun versucht von den LBs zu stehlen. Auch hier muss solange gewartet werden, bis die Anfrage beantwortet wurde. Wie bereits erwähnt, wird während des Stehlens von LBs die zugehörige Lebenslinie aktiviert. Falls alle Lifeline-Steal-Requests erfolglos waren, wird der Worker in den sog. *Ruhemodus* versetzt. Ansonsten wird die Taskbearbeitung fortgesetzt. Ein Worker wird wieder aktiv bzw. verlässt den Ruhemodus, wenn er von einem LB neue Tasks erhält. Das Programm terminiert erst, wenn alle Worker im Ruhemodus sind und damit auch keine Tasks mehr vorliegen.

In Abbildung 3.1 wird das beschriebene Konzept noch einmal anhand eines vereinfachten Flussdiagramms dargestellt. Die ausgehenden Kanten des Ruhemodus wurden zusätzlich gekennzeichnet (*1 und *2), um zu verdeutlichen, dass dieser Zustand nur durch Aktionen von außen verlassen wird. Bei *1 wird die Aktivierung beispielsweise von einem LB

angestoßen, wohingegen *2 von einer übergeordneten Kontrollinstanz durchgeführt wird (siehe Kapitel 4).

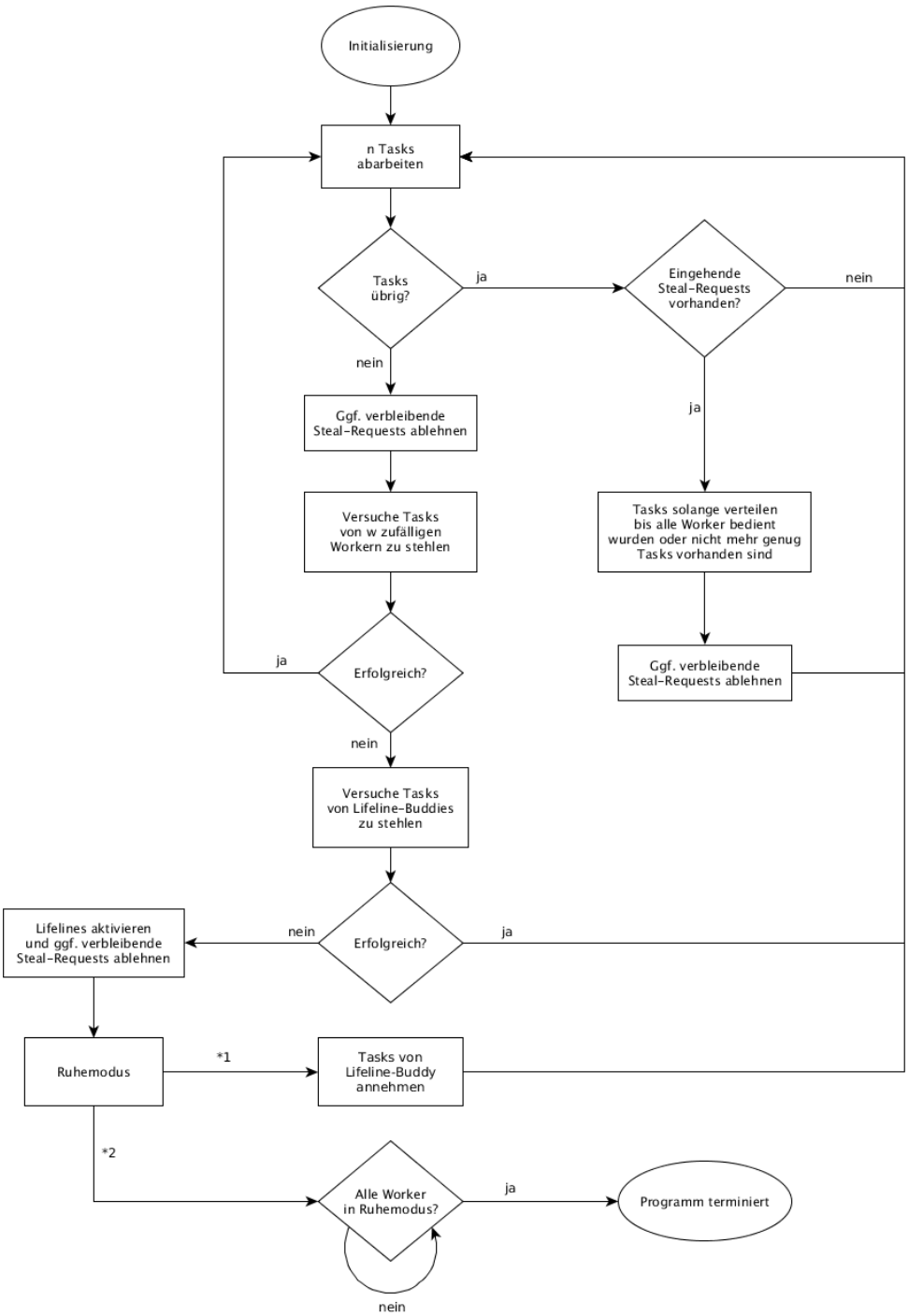

Abbildung 3.1: Flussdiagramm - GLB

4 Implementierung

Bei der Entwicklung der globalen Lastbalancierung für Chapel, wurde darauf geachtet möglichst nah an der GLB_X10-Implementierung [23] zu bleiben. Damit konnte ein besserer Vergleich in Bezug auf die Performance erreicht werden. Leider war eine exakte Nachbildung nicht möglich, da nicht alle benötigten Funktionalitäten in Chapel zur Verfügung standen. Von Chapel wurde die derzeit aktuellste Version (1.11.0) verwendet.

4.1 Struktur

Das Klassendiagramm in Abbildung 4.1 gibt einen Überblick über die interne Struktur der GLB_CHPL-Implementierung. Für eine bessere Wiederverwendbarkeit wurde der generische Typparameter T verwendet. Leider wird die Usability der GLB_CHPL-Implementierung aufgrund fehlender Funktionalitäten in Chapel stark eingeschränkt. Zum einen ist es derzeit noch nicht möglich abstrakte Klassen oder Interfaces zu definieren und zum anderen kann nicht von generischen Klassen geerbt werden. Auf konkrete Beispiele wird an dieser Stelle bewusst verzichtet, da zunächst einige benötigte Grundlagen zur Implementierung erläutert werden sollen. Weitere Informationen bezüglich der Usability folgen in Abschnitt 4.3.

Die Klasse `GLB` stellt den Einstiegspunkt dar. Beim Konstruktoraufruf des `GLB`-Objekts können unterschiedliche Parameter zur Konfiguration gesetzt werden. Dazu gehören unter anderem die maximale Anzahl abzuarbeitender Tasks (n), die Anzahl der zu sendenden zufälligen Stehlanfragen (w) und die Basis h des Lifeline-Graphen (vgl. Kapitel 3). Die Methode `run` startet die Berechnungen und gibt das Ergebnis (aus Konsistenzgründen) in Form eines Arrays zurück. In der Regel enthält dieses Array nur einen einzelnen Wert. Allerdings gibt es auch Fälle, bei denen mehr als eine Lösung existiert (z.B. Betweenness-Centrality in Kapitel 5). Der Einfachheit halber wird in den nachfolgenden Erläuterungen

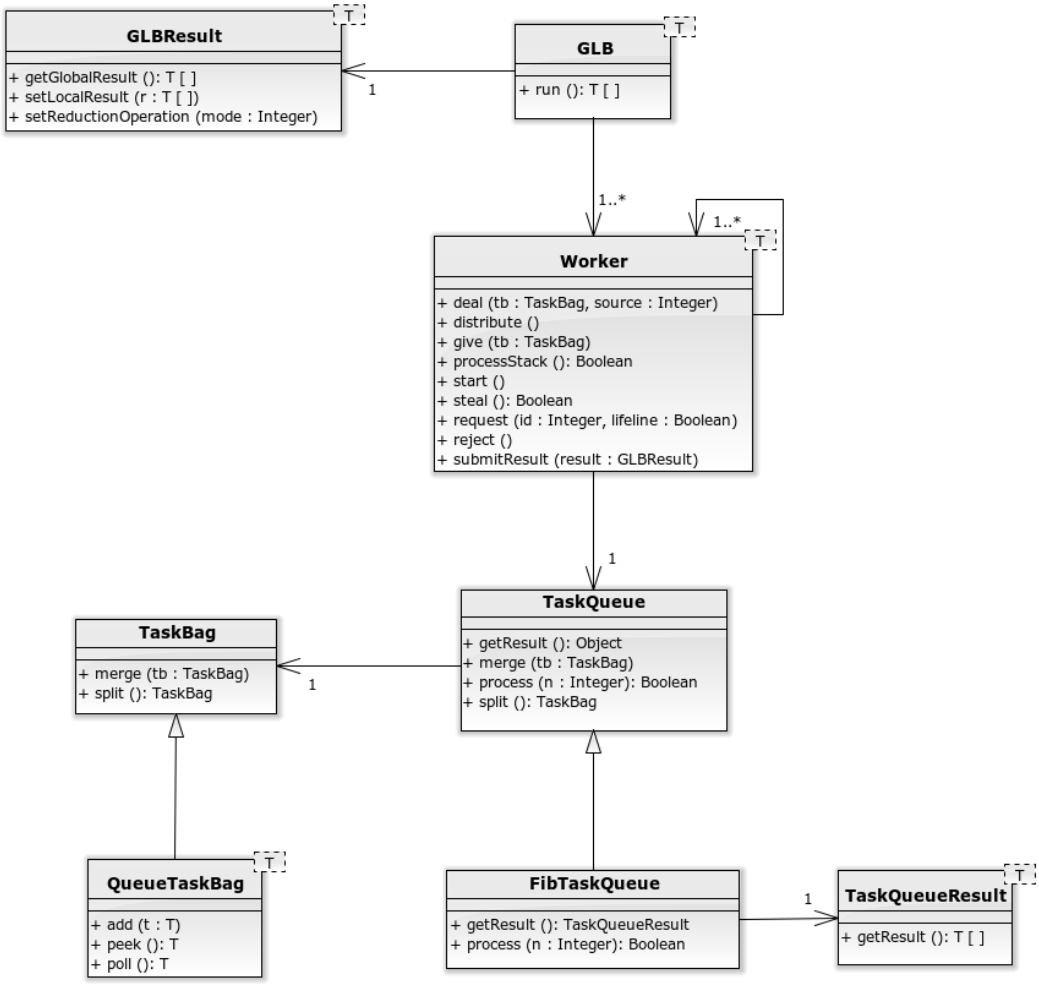

Abbildung 4.1: Klassendiagramm - GLB

die Formulierung auf Ergebnisse mit einem Wert beschränkt, wobei die Aussagen natürlich auch für mehrere Ergebnisse gelten.

Zur Bestimmung des Ergebnisses wird intern die Klasse `GLBResult` verwendet. Dabei wird das lokale Endergebnis jedes Workers in einem Objekt vom Typ `GLBResult` gespeichert (`setLocalResult`). Sobald die Ergebnisse aller Worker vorliegen, werden diese mit Hilfe von Reduktion zusammengefasst und zurückgegeben (`getGlobalResult`). Der Reduktionsoperator kann ebenfalls beim Aufruf des Konstruktors des `GLB`-Objekts festgelegt werden. In der Methode `run` wird das zusammengefasste Ergebnis dann lediglich weitergeleitet.

Die Klasse `Worker` entspricht dem in Kapitel 3 beschriebenen Worker. Von dieser Klasse wird genau ein Objekt je Locale instanziiert. Jeder `Worker` hat eine Referenz auf alle anderen

`Worker`. Damit ist gewährleistet, dass Stehlanfragen an beliebige Worker gesendet werden können. Um Wiederholungen zu vermeiden werden die aufgelisteten Methoden der Klasse `Worker` erst in Abschnitt 4.2 erläutert.

Ein wichtiger Bestandteil sind die sog. `TaskQueues`, die vom Benutzer der GLB_CHPL-Implementierung bereitgestellt werden müssen. Jeder `Worker` hat genau eine solche `TaskQueue`. Die Methode `process` arbeitet die vorhandenen Tasks ab, wobei immer höchstens `n` Tasks bearbeitet werden (vgl. Kapitel 3). Mit der `getResult`-Methode kann das lokale Ergebnis der `TaskQueue` abgerufen werden. Der Typ des Ergebnisses müsste an dieser Stelle generisch sein, wobei `TaskQueue` deshalb im Idealfall auch eine generische abstrakte Klasse sein sollte. Allerdings ist dies, wie bereits erwähnt, leider noch nicht möglich. Um das Problem der Vererbung in Verbindung mit Generizität zu umgehen, wurde die Methode `getResult` so deklariert, dass sie den Rückgabetyp `object` hat. Beispielhaft wurde in Abbildung 4.1 die `FibTaskQueue` angegeben, welche für den Fibonacci-Benchmark aus Kapitel 5 verwendet wurde. Weil alle Klassen in Chapel implizit oder explizit von `object` erben, können von `TaskQueue` abgeleitete Klassen den Rückgabetyp mit der generischen Klasse `TaskQueueResult` überschreiben (siehe `FibTaskQueue` in Abbildung 4.1). Diese Klasse beinhaltet das eigentliche Ergebnis. Damit kann nun die Klasse `Worker` das lokale Endergebnis der `TaskQueue` abrufen und dieses mit Hilfe der Methode `submitResult` (der Klasse `Worker`) in dem zuvor erwähnten `GLBResult`-Objekt abspeichern.

Die Tasks selbst liegen in einem Objekt der Klasse `TaskBag`. Standardmäßig bietet die GLB_CHPL-Implementierung einen `QueueTaskBag` an. Dieser speichert alle Tasks in einer Warteschlange. Der Benutzer kann auch selbst einen `TaskBag` programmieren, wobei darauf geachtet werden muss die Methoden `merge` und `split` zu überschreiben. In `merge` werden eingehende Tasks (in Form eines `TaskBag`s) dem lokalen Taskpool hinzufügt. Im Gegensatz dazu werden bei `split` Tasks entnommen und zurückgegeben. Ein Aufruf der Methoden `merge` und `split` der Klasse `TaskQueue` wird für gewöhnlich einfach an die Klasse `TaskBag` weitergeleitet.

4.2 Ausgewählte Aspekte

Quelltext 4.1 zeigt die `run`-Methode der Klasse `GLB`. Wie bereits in Kapitel 3 erwähnt wurde, ist es nicht einfach festzustellen, ob ein auf Work Stealing basierendes Programm alle Tasks abgearbeitet hat. Mit Hilfe des von Saraswat et al. [18] vorgestellten Konzepts und

den von Chapel zur Verfügung gestellten Sprachkonstrukten wird für die GLB_CHPL-Implementierung nur eine zusätzliche Anweisung benötigt um dieses Problem zu lösen: `sync` (siehe Zeile 2). Diese Anweisung stellt sicher, dass alle im Block erzeugten CTs abgeschlossen wurden, bevor der Programmfluss in Zeile 10 fortgesetzt wird. Wenn keine CTs mehr ausgeführt werden, befinden sich alle Worker im Ruhemodus. Daraus folgt nach der Beschreibung in Kapitel 3, dass alle Tasks abgearbeitet wurden und das Programm terminieren kann.

In Zeile 3 von Quelltext 4.1 wird zunächst überprüft, ob die Arbeitslast dynamisch erzeugt wird oder statisch vorgegeben ist. Eine dynamisch erzeugte Last kennzeichnet sich dadurch aus, dass die Menge der Tasks im Vorfeld nicht bekannt ist. Durch die Abarbeitung einer Task können beliebig viele weitere Tasks entstehen. An dieser Stelle wird angenommen, dass nur eine oder sehr wenige Tasks beim Start des Programms vorliegen. Diese sollten nur einem Worker zugeordnet sein. Der Worker wird dann als einziger gestartet (Zeile 4), um unnötige Stehlanfragen (aufgrund zu weniger Tasks) zu vermeiden. Außerdem werden alle Lebenslinien (außer die des gestarteten Workers) aktiviert, damit die Tasks effizient verteilt werden.

Im Gegensatz dazu sind bei einer statisch vorgegebenen Last bereits alle Tasks vorab bekannt und werden auf die zur Verfügung stehenden `TaskQueues` verteilt. In diesem Fall können alle Worker gleichzeitig gestartet werden (Zeile 6). Bei der Variable `workers` handelt es sich um ein zyklisch verteiltes Array der Länge P, mit $P =$ "Anzahl der Locales". Damit existiert genau eine Instanz der Klasse `Worker` je Locale. Die Anweisung `workers.start()` (Zeile 6) startet parallel auf jedem Locale eine CT in dem die `start`-Methode des entsprechenden `Worker`-Objekts ausgeführt wird.

Das der Start der Worker anhand der Art der Last unterschieden wird kann verwirrend sein. Beispielsweise könnten bei einer dynamischen Last zu Beginn auch mehr Tasks als Worker vorliegen. In einem solchen Fall sollten natürlich die Tasks auf alle Worker verteilt und auch jeder Worker gestartet werden. Allerdings müsste dafür die widersprüchliche Angabe gemacht werden, dass die Last statisch ist.

Mit dem Starten der Worker beginnt der parallele Teil des Programms. Um die Konsistenz der Daten zu bewahren, müssen bestimmte Programmabschnitte synchronisiert werden. In diesen sog. *kritischen Abschnitten* ist zu einem bestimmten Zeitpunkt immer nur eine aktive CT pro Locale erlaubt. Zur Synchronisierung der CTs wurden in der Klasse `Worker` zwei Synchronisationsvariablen definiert: `restartLock` und `waiting`. Falls sich der Worker im

```
1  proc run() {
2    sync {
3      if(dynamic) {
4          workers(0).start();
5      } else {
6          workers.start();
7      }
8    }
9    // submitting worker's local result
10   workers.submitResult(resType, result);
11
12   // reducing and returning result
13   return result.getGlobalResult();
14 }
```

Quelltext 4.1: GLB - run()

Ruhemodus befindet, so führt keine CT die `start`-Methode aus. Damit der Worker die Taskbearbeitung fortsetzt muss eine neue CT erstellt werden, welche die `start`-Methode aufruft. Mit dem `restartLock` wird eine mehrfache Ausführung dieser Methode verhindert. Die Synchronisationsvariable `waiting` lässt den Worker nach einem Steal-Request solange warten, bis eine Antwort erhalten wurde.

Darüber hinaus existiert eine weitere Synchronisationsvariable (`empty`), die allerdings nicht zum Absichern kritischer Abschnitte genutzt wird, sondern garantiert, dass alle parallel laufenden CTs auch wirklich den aktuellen Wert aus dem Speicher lesen. Sollte beispielsweise eine CT der Variable `empty` einen neuen Wert zuweisen, während dieser kurz danach von einer anderen CT ausgelesen wird, so garantiert Chapel nur für Synchronisationsvariablen Speicherkonsistenz, d.h. nur für Synchronisationsvariablen ist sichergestellt, dass die lesende CT auch wirklich den kurz zuvor geschriebenen Wert erhält (vgl. [1], Kap. 30). Der Zweck von `emtpy` ist es zu signalisieren, ob der lokale Taskpool leer (`true`) oder voll (`false`) ist. Alle drei Synchronisationsvariablen sind vom Typ `Boolean`.

```
1  proc start() {
2    empty$.writeXF(!queue.hasWork());
3    active = true;
4
5    do {
6       restartLock$.writeXF(true);
7       processStack();
8
9       // prevent multiple restarts
10      restartLock$;
11   } while (!empty$.readXX());
12   active = false;
13   restartLock$.writeXF(true);
14 }
```

Quelltext 4.2: Worker - start()

```
1  proc processStack() {
2     do {
3        while(queue.process(n)) {
4           distribute();
5           reject();
6        }
7        reject();
8     } while(steal());
9     reject();
10 }
```

Quelltext 4.3: Worker - processStack()

```
1  proc deal(loot : TaskBag, source : int) {
2     const lifeline = (source >= 0);
3     if(lifeline) {
4        lifelinesActivated(source) = false;
5     }
6     queue.merge(loot);
7     empty$.writeXF(false);
8     waiting$.writeXF(false);
9     restartLock$;
10    if(!active) {
11      // deal(...) runs in own Chapel-Task
12      start();
13    } else {
14      restartLock$.writeXF(true);
15    }
16 }
```

Quelltext 4.4: Worker - deal()

Quelltext 4.2 zeigt die `start`-Methode der Klasse `Worker`. Hier werden zuerst die Variablen `empty` und `active` initialisiert (Zeile 2 + 3). Für `empty` wird die in Abschnitt 2.3 beschriebene Methode `writeXF` verwendet. Das ist notwendig, da diese Variable niemals leer ist. Eine normale Zuweisung ohne vorherigen Lesezugriff ($empty\$ = true$) würde den Thread blockieren. Alternativ könnte die Methode `writeFF` verwendet werden. Jedoch wird hier die Überprüfung, ob sich die Variable im Zustand voll befindet, nicht benötigt. Der Aufruf `queue.hasWork()` überprüft ob Tasks in der `TaskQueue` vorliegen und initialisiert `empty` dementsprechend.

Die Variable `active` beschreibt den Zustand des Workers. Enthält sie den Wert `true`, so ist der Worker aktiv, d.h., eine CT führt die `start`-Methode aus. Falls `active` den Wert `false` beinhaltet, befindet sich der Worker im Ruhemodus.

Ein wichtiger Bestandteil der `start`-Methode ist der Aufruf der Funktion `processStack` (Zeile 7). Der dazugehörige Quelltext 4.3 bildet das in Abbildung 3.1 beschriebene Verhalten ab. Die Methode `processStack` wird solange ausgeführt, bis keine Tasks mehr vorliegen (Zeile 11 in 4.2). In diesem Fall wird der Worker in den Ruhemodus versetzt (Zeile 12 in 4.2). Allerdings kann es währenddessen vorkommen, dass Tasks von einem oder mehreren LBs eintreffen. Der Worker, der die Tasks teilen möchte, ruft die `deal`-Methode (4.4) des anfragenden Workers auf. Diese wird parallel in einem eigenen CT ausgeführt und startet den Worker, falls sich dieser im Ruhemodus befindet (Zeile 12 in 4.4). Bevor der genaue Synchronisationsmechanismus erläutert wird, der das mehrfache Starten des Workers verhindert, sollen zunächst die `processStack`- und `deal`-Methode im Detail erläutert werden.

In `processStack` (4.3) werden zunächst n Tasks abgearbeitet (Zeile 3). Anschließend wird die Methode `distribute` aufgerufen (Zeile 4). Diese Methode ist für die Bearbeitung eingehender Steal-Requests verantwortlich. Dabei werden Tasks an anfragende Worker nur gesendet, wenn mindestens zwei Tasks (lokal) vorliegen. Sollten nach dem Aufruf von `distribute` noch Steal-Requests vorliegen, so werden diese abgelehnt (Zeile 5). An dieser Stelle werden Stehlanfragen von LBs separat abgespeichert und ggf. zu einem späteren Zeitpunkt bearbeitet (vgl. Kapitel 3). Die Abarbeitung der Tasks und Anfragen wird solange wiederholt, bis (lokal) keine Tasks mehr verfügbar sind. In diesem Fall werden zunächst wieder alle noch ausstehenden Anfragen abgelehnt (Zeile 7). Im Anschluss daran wird versucht Tasks von anderen Workern zu stehlen (Zeile 8). Sollte das Stehlen erfolgreich sein, so wird die Abarbeitung fortgesetzt. Ansonsten werden evtl. neu hinzugekommene Steal-Requests abgelehnt (Zeile 9) und das Programm in Zeile 10 von Quelltext 4.2 fortgesetzt.

Quelltext 4.4 zeigt die `deal`-Methode. Hier wird zu Beginn überprüft, ob die eingehenden Tasks von einem LB oder einem zufällig ausgewählten Worker gesendet wurden (Zeile 3). Falls die Tasks von einem LB kommen, so wird die während des Steal-Requests aktivierte Lebenslinie wieder deaktiviert (Zeile 4). Danach werden die eingehenden Tasks der lokalen `TaskQueue` hinzugefügt (Zeile 6) und die Synchronisationsvariablen `empty` und `waiting` jeweils auf `false` gesetzt (Zeile 7 + 8). Die Zuweisung erfolgt wie in Quelltext 4.2 mit Hilfe von `writeXF`. An dieser Stelle kann `waiting` leer oder voll sein, je nachdem, ob der Worker gerade einen (anderen) Steal-Request gestartet hat. Falls `deal` z.B. von einem LB aufgerufen wird, so kann es sein, dass der Aufruf zu einem beliebigen Zeitpunkt nach der Aktivierung der Lebenslinie erfolgt (vgl. Kapitel 3). Zu diesem Zeitpunkt könnte der Worker eine neue Stehlanfrage gesendet haben und auf die Antwort des Opfers warten. In diesem Fall hat `waiting` den Zustand leer. Durch die Zuweisung unterbricht der LB den Worker während des Wartens auf die Antwort, wodurch dieser seine Arbeit bereits früher fortsetzen kann. Allerdings muss nun das Opfer berücksichtigen, dass `waiting` bereits im Zustand voll ist und damit ein nicht-blockierender Schreibzugriff benötigt wird. Sollte der Aufruf der `deal`-Methode jedoch nicht während eines Steal-Requests stattfinden, so ist `waiting` immer voll. Aus diesem Grund wird bei der Zuweisung eines Wertes an `waiting` stets die `writeXF`-Methode verwendet.

Daraufhin wird in Zeile 9 der Wert des `restartLock`s gelesen. An dieser Stelle soll noch einmal genauer darauf eingegangen werden, wie ein mehrfacher Aufruf der `start`-Methode verhindert wird. Der `restartLock` wird ausschließlich in `start` (4.2) und `deal` (4.4) genutzt.

Um der Variablen einen Wert zuzuweisen, wird ebenfalls die Methode `writeXF` verwendet. Allerdings gilt für den `restartLock`, dass auch eine einfache Zuweisung genügen würde. Jedoch wird bei einfachen Zuweisungen immer überprüft, ob die Variable wirklich leer ist. Da diese Überprüfung aber nicht benötigt wird, erfolgt der Schreibzugriff über die `writeXF`-Methode. Zu Beginn der `start`-Methode wird die Variable aufgefüllt (Zeile 6 in Quelltext 4.2). Der zugewiesene Wert spielt dabei keine Rolle. Im Anschluss daran wird die bereits beschriebene `processStack`-Methode ausgeführt. In der Zwischenzeit können entfernte Worker (aufgrund von Steal-Requests) die Methode `deal` aufrufen. Für diese Aufrufe ist eine Synchronisation mit Hilfe des `restartLock`s nicht nötig, da der Worker während der Ausführung von `processStack` aktiv ist und die `start`-Methode demnach nie aufgerufen wird. Interessant wird es, wenn der Worker die `processStack`-Methode verlässt. In diesem Fall existiert eine Wettlaufsituation (Race Condition) zwischen der Ausführung von `start` und `deal`. Hier müssen zwei Fälle unterschieden werden: entweder liest der Worker den `restartLock` zuerst (Zeile 10 in 4.2), oder der CT, der die `deal`-Methode ausführt (Zeile 9 in 4.4).

1. Fall:
Als Erstes soll der Fall betrachtet werden, in dem der Worker den `restartLock` zuerst liest. Dabei muss wieder zwischen zwei Fällen unterschieden werden. Falls `empty` in `deal` noch nicht auf `false` gesetzt wurde (Zeile 7 in 4.4), so verlässt der Worker die Schleife in der `start`-Methode und wechselt anschließend in den Ruhemodus (Zeile 11 + 12 in 4.2). Vor dem Verlassen der `start`-Methode wird der `restartLock` wieder aufgefüllt (Zeile 13 in 4.2), so dass dieser in der `deal`-Methode gelesen werden kann (Zeile 9 in 4.4). Nachdem der Wert gelesen wurde, wird der Zustand des Workers überprüft (Zeile 10 in 4.4). Da sich der Worker im Ruhemodus befindet, wird nun die `start`-Methode ausgeführt (Zeile 12 in 4.4). In der Methode `start` wird zunächst signalisiert, dass der Worker wieder aktiv ist (Zeile 3 in 4.2). Anschließend wird dem `restartLock` ein neuer Wert zugewiesen und `processStack` aufgerufen (Zeile 6 + 7 in 4.2).

Sollte `empty` allerdings in `deal` bereits auf `false` gesetzt worden sein (Zeile 7 in 4.4), so setzt der Worker seine Arbeit fort (Zeile 11 in 4.2). Als Nächstes wird der `restartLock` wieder aufgefüllt (Zeile 6 in 4.2) und in der `deal`-Methode gelesen (Zeile 9 in 4.4). Da der Worker noch aktiv ist (Zeile 10 in 4.4), wird dem `restartLock` lediglich ein neuer Wert zugewiesen (Zeile 14 in 4.4) und die `deal`-Methode verlassen.

2. Fall:

Im zweiten Fall liest die `deal`-Methode den Wert zuerst. Hier wird zunächst überprüft ob der Worker noch aktiv ist (Zeile 10 in 4.4). An dieser Stelle muss der Worker noch aktiv sein, da er beim Versuch den `restartLock` zu lesen (Zeile 10 in 4.2) warten muss. Daraus folgt, dass der `restartLock` wieder einen Wert zugewiesen bekommt (Zeile 14 in 4.4) und der `deal`-Aufruf anschließend beendet wird. Nun kann der Wert in der `start`-Methode gelesen werden (Zeile 10 in 4.2). Im nächsten Schritt wird überprüft, ob neue Tasks vorliegen (Zeile 11 in 4.2). Da in der `deal`-Methode `empty` der Wert `false` zugewiesen wurde, setzt der Worker seine Arbeit fort.

Zudem sei erwähnt, dass die Synchronisation auch für die parallele Ausführung mehrerer `deal`-Methoden funktioniert. Aufgrund der ausführlichen Bescheibung des `restartLock`s soll darauf nicht im Detail eingegangen, da sich das Prinzip einfach auf diesen Fall übertragen lässt.

```
1  proc steal() : bool {
2    empty$.writeXF(!queue.hasWork());
3    if(numLocales == 1) {
4      return false;
5    }
6
7    for i in 0..w-1 {
8      if(!empty$.readXX()) {
9         return true;
10     }
11
12     const v = victims(randomInt(m));
13     const h = here.id;
14     waiting$;
15     on Locales(v) do begin {
16       workers(v).request(h, false);
17     }
18     // wait for feedback
19     waiting$.readFF();
20   }
```

Quelltext 4.5: Worker - steal()

```
21   // getting work from random victims failed
         - check lifeline buddies
22   for i in lifelines.domain {
23     if(!empty$.readXX()) {
24       return true;
25     }
26
27     const lifeline = lifelines(i);
28     if(!lifelinesActivated(lifeline)) {
29       // mark lifeline as active
30       lifelinesActivated(lifeline) = true;
31       const h = here.id;
32       waiting$;
33       on Locales(lifeline) do begin {
34         workers(lifeline).request(h, true);
35       }
36       // wait for feedback
37       waiting$.readFF();
38     }
39   }
40   return !empty$.readXX();
41 }
```

Quelltext 4.6: Worker - steal() #2

Die Quelltexte 4.5, 4.6 und 4.7 zeigen die `steal`- und `request`-Methode der Klasse `Worker`. Ein Worker versucht immer dann Tasks zu stehlen, wenn lokal keine mehr vorliegen (Zeile 8 in Quelltext 4.3). Zu Beginn wird überprüft ob die lokale `TaskQueue` noch Tasks beinhaltet und die Variable `empty` dementsprechend aktualisiert (Zeile 2 in Quelltext 4.5). Hier wird mit Absicht nicht direkt der Wert `true` in `empty` geschrieben, da in der Zwischenzeit neue

Tasks eingegangen sein könnten. Wie zuvor beschrieben, kann es sein, dass ein LB die `deal`-Methode zu einem beliebigen Zeitpunkt aufruft und infolgedessen `empty` auf `false` setzt (Zeile 7 in Quelltext 4.4). Um diesen Wert nicht versehentlich zu überschreiben, muss die `hasWork`-Methode der Klasse `TaskQueue` verwendet werden. Falls `empty` an dieser Stelle den Wert `false` erhält, so wird die `steal`-Methode in der ersten Iteration der ersten Schleife verlassen (Zeile 9 in 4.5).

Vor dem Senden von Steal-Requests wird zunächst überprüft, ob das Programm mit nur einem Locale gestartet wurde (Zeile 3 in 4.5). In diesem Fall müssen keine Stehlanfragen gesendet werden, da keine anderen Worker existieren, von denen Tasks gestohlen werden könnten (Zeile 4 in 4.5). Ansonsten werden im nächsten Schritt Stehlanfragen an w zufällige Opfer gesendet (Zeile 7-20 in 4.5). Sollten alle Versuche ohne Erfolg sein, so wird anschließend eine Anfrage an jeden LB gesendet (Zeile 22-28 in Quelltext 4.6). Nach jeder Anfrage wird auf eine Antwort des entfernten Workers gewartet. Zu diesem Zweck wird die Synchronisationsvariable `waiting` verwendet. Vor dem Versenden der Anfrage wird diese Variable zuerst geleert (Zeile 14 und 32 in 4.5 und 4.6). Im Anschluss daran wird eine CT auf dem entfernten Locale des Opfers gestartet (Zeile 15 und 33). Danach wartet der Worker (der versucht zu stehlen) solange, bis er eine Antwort erhalten hat. Dafür wird die Variable `waiting` mit Hilfe der Methode `readFF` abgefragt. Diese Methode blockiert solange, bis `waiting` wieder den Zustand voll hat. Nach dem Funktionsaufruf verbleibt die Variable in diesem Zustand. Dadurch kann `waiting` in der nächsten Schleifeniteration wieder gelesen werden.

Auf dem Locale, der die Stehlanfrage erhält, wird die `request`-Methode des lokalen Workers aufgerufen. Hier wird zuerst überprüft, ob der Taskpool voll oder leer ist (Zeile 2 in 4.7). Falls keine Tasks vorhanden sind, wird die Stehlanfrage abgelehnt (Zeile 9 in 4.7). Dafür wird die `waiting`-Variable des anfragenden Workers wieder mit einem Wert befüllt. Bevor die Anfrage abgelehnt wird, muss noch einmal unterschieden werden, ob es sich bei dem anfragenden Worker um einen LB handelt. Sollte dies der Fall sein, so merkt sich der Worker diesen (Zeile 6 in 4.7) und leitet Tasks weiter, sobald neue zur Verfügung stehen (vgl. Kapitel 3).

Sollte der Taskpool noch Aufgaben enthalten, so wird der Request gespeichert (Zeile 14 und 16 in 4.7) und nach dem Verlassen der `process`-Methode (Zeile 4 in Quelltext 4.3) bedient. Die IDs von gewöhnlichen "Dieben", die keine LBs sind, werden dabei negiert. Diese Unterscheidung wird für Statistikzwecke (Lifeline-Steals vs. gewöhnliche Steals) und während des Ablehnens ausstehender Stehlanfragen verwendet.

```
1  proc request(id : int, lifeline : bool) {
2        if(empty$.readXX()) {
3           // no work left to share
4           if(lifeline) {
5              // only remember lifeline thieves and distribute work later
6              lifelineThieves.push(id);
7           }
8
9           workers(id).waiting$.writeXF(false);
10       } else {
11          // we've got work left -> remember thief.
12          // Mark non-lifeline thiefs with negative id and substract 1
13          if(lifeline) {
14             thieves.push(id);
15          } else {
16             thieves.push(-id-1);
17          }
18       }
19    }
```

Quelltext 4.7: Worker - request()

4.3 Vergleich mit X10-Implementierung

In diesem Abschnitt wird detaillierter auf die Unterschiede eingegangen, die aufgrund fehlender Funktionalitäten in Chapel entstanden sind.

Ein wichtiger Punkt ist die Einschränkung der Parallelität pro Rechenknoten in der X10-Implementierung. Momentan wird nur ein Thread pro Place unterstützt. Allerdings ist es geplant die Bibliothek so zu erweitern, dass auch mehrere Threads verwendet werden können [2]. Der aktuelle Stand der GLB_X10-Implemtierung erfordert nach jedem Durchlauf der `TaskQueue.process()`-Methode einen expliziten Aufruf von `Runtime.probe()`. Hier wird überprüft, ob auf dem aktuellen Place ausstehende Aktivitäten vorhanden sind. Sollte dies so sein, so werden die Aktivitäten nacheinander vom aktuellen Thread gestartet und abgearbeitet. Mit diesem Vorgehen wird keine Synchronisation innerhalb des Workers benötigt, da alle Anweisungen sequentiell ausgeführt werden.

Dieses Verhalten konnte in Chapel nicht nachgebildet werden. Eine mit `Runtime.probe()` vergleichbare Methode steht nicht zur Verfügung und auch die Verwendung des Tasking Layers `qthreads` hat nicht geholfen (vgl. Abschnitt 2.3). Alternativ hätten die Methoden `process`, `steal`, `deal` und `request` als kritische Abschnitte definiert werden können, womit auch in Chapel nur eine CT pro Locale aktiv gewesen wäre. Da allerdings in X10 die Einschränkung der Parallelität nur vorübergehend sein soll, wurde für die Chapel-Implementierung direkt der parallele Ansatz gewählt. Damit konnten die kritischen Abschnitte auf ein Minimum reduziert werden, was die Laufzeit verbessert.

In Abschnitt 4.1 wurde bereits darauf hingewiesen, dass die Usability der GLB_CHPL-Implementierung aufgrund von fehlenden Interfaces und abstrakten Klassen, sowie der Unmöglichkeit des Erbens von generischen Klassen stark beeinträchtigt wird. Demgegenüber ist dies alles in X10 möglich. Ein gutes Beispiel dafür ist die `TaskQueue`, die in der GLB_X10-Implementierung ein generisches Interface ist. Klassen, die dieses Interface implementieren, müssen die in `TaskQueue` deklarierten Methoden überschreiben. Im Gegensatz dazu musste die `TaskQueue` in Chapel als Klasse definiert werden. Somit muss der Benutzer genau darauf achten, welche Methoden für eine korrekte Funktion überschrieben werden sollten. Das erfordert im Vergleich zur X10-Implementierung einen deutlich höheren Einarbeitungsaufwand. Zusätzlich muss darauf geachtet werden, dass beim Überschreiben einer Methode eine identische Deklaration erfolgt. Dazu gehören auch die Namen der Parameter. Weicht ein Parametername auch nur geringfügig ab (ggf. wegen eines Tippfehlers), so wird die Methode nicht überschrieben und der Benutzer definiert eine neue Methode. Ein solcher Fehler fällt nicht immer sofort auf und ist meist mit unnötig langen Nachforschungen verbunden.

Abschließend soll noch erwähnt werden, dass in Chapel die Verwendung von Attributen oder Methoden noch nicht mit Zugriffsmodifikatoren eingeschränkt werden kann. Auf diese Weise kann der Benutzer zur Laufzeit des Programms Daten des Frameworks manipulieren, wodurch ein inkonsistenter Zustand entsteht. In X10 hingegen existieren solche Zugriffsmodifikatoren.

5 Experimente

Die folgenden Benchmarks wurden aus [18] übernommen. In X10 existierten bereits Beispielprogramme für diese Benchmarks, so dass die Implementierung nur noch für Chapel vorgenommen werden musste. Ausgeführt wurden die Benchmark-Tests auf dem Lichtenberg-Hochleistungsrechner der Technischen Universität Darmstadt. Insgesamt wurden bis zu acht Rechenknoten mit je 16 Prozessoren und acht Rechenkernen genutzt. Zum Zeitpunkt der Benchmarks war auf jedem dieser Rechenknoten SuSe Linux installiert. Als Prozessor wurde der Intel® Xeon® E5-2670 mit einer Taktfrequenz von 2,6 GHz verwendet. Für alle Benchmarks wurden 1, 2, 4, . . . , 256 Locales bzw. Places verwendet, wobei jedem Rechenknoten zyklisch ein Locale/Place zugeordnet wurde. Zum Beispiel waren bei acht Locales/Places auf jedem Rechenknoten genau ein Locale/Place, bei 16 jeweils zwei usw.

Dabei hat die Ausführung der X10-Programme mit 256 Places nicht funktioniert. Die Programme starteten zwar, lieferten jedoch auch nach mehreren Stunden keine Ergebnisse bzw. Fehlermeldungen. Deshalb wurden die Testläufe mit 256 Locales bzw. Places nicht in die Laufzeitmessungen mit einbezogen.

Das Einrichten von Chapel auf dem Computer Cluster der TU Darmstadt erwies sich leider als sehr schwierig. Bereits die Standardinstallation von Chapel, die zur Kommunikation zwischen den Locales UDP verwendet, scheiterte. Beide Programmiersprachen unterstützen die Hochgeschwindigkeitsübertragungstechnik Infiniband, allerdings konnte diese nur mit X10 genutzt werden. Trotz umfangreicher Hilfestellungen des Entwicklerteams und des Supports von Cray Inc. war es nicht möglich Infiniband zu verwenden, weshalb schließlich Myrinet verwendet wurde. Myrinet verfügt über einen deutlich höheren Datendurchsatz als Ethernet, wurde allerdings in den vergangenen Jahren durch schnellere Techniken wie Infiniband und Gigabit-Ethernet abgelöst.

Die Konfigurationsparameter wurden aus den bereits bestehenden Programmen in X10 übernommen. Für die Berechnung der Fibonacci-Zahlen wurde $n = 100$ und $w = h = 4$

gewählt. Im Betweenness-Centrality und Unbalanced Tree Search Benchmark sind jeweils dieselben Parameter verwendet worden: $n = 511$, $w = 1$ und $h = 32$.

5.1 Benchmarks

Fibonacci

Ein für die globale Lastbalancierung gut geeigneter Benchmark ist die Berechnung der z-ten Fibonacci-Zahl. Die Fibonacci-Folge ist durch die folgende Rekursion definiert:

$$f_z = \begin{cases} 0 & \text{, für } z = 0 \\ 1 & \text{, für } z = 1 \\ f_{z-1} + f_{z-2} & \text{, sonst} \end{cases}$$

Für die Implementierung des Benchmarks wurde jeder Rekursionsschritt als Task abgebildet. Sollte z.B. f_3 berechnet werden, so wurde dies als Task der `TaskQueue` hinzugefügt. Bei der Abarbeitung dieser Task entstehen zwei neue (für die Berechnung von f_2 und f_1). Sobald $z \leq 2$ gilt, bricht die Erstellung neuer Tasks ab.

Betweenness-Centrality

Betweenness-Centrality (BC) wird in deutschen Veröffentlichungen auch oft als Intermediationszentralität oder Zwischenzentralität bezeichnet. Die Idee des BC stammt ursprünglich von Linton C. Freeman [14]. Bei diesem Verfahren wird für jeden Knoten v eines Graphen G ein sog. Betweenness-Centrality-Score (BC-Score) berechnet. Der BC-Score beschreibt dabei das Verhältnis der Anzahl der kürzesten Wege zwischen zwei beliebigen Knoten s, t (wobei $s \neq v \neq t$) und der Anzahl der kürzesten Wege in denen v enthalten ist. Sei σ_{st} die Anzahl der kürzesten Pfade zwischen s und t. Darüber hinaus sei $\sigma_{st}(v)$ die Anzahl der kürzesten Pfade die durch v gehen. Dann ist die BC eines Knotens v definiert als

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}. \tag{5.1.1}$$

Der für die GLB_X10-Implementierung verwendete Benchmark wurde von David Bader et al. [6] übernommen. Dort wird eine Applikation beschrieben, die unterschiedliche Techniken zur Analyse von gerichteten Graphen mit Kantengewichten nutzt. Diese Techniken werden auch als *Kernel* bezeichnet, wobei der BC-Benchmark dem Kernel 4 mit leichten Modifikationen entspricht (vgl. [6], Kap. 2.5). Beispielsweise werden nur ungewichtete Graphen betrachtet und Berechnungen für kürzeste Pfade mit Hilfe des Algorithmus von Dijkstra [10] gelöst. In [6] werden die Kantengewichte genutzt, um bestimmte Kanten herauszufiltern und sie damit in der Berechnung der BC nicht zu berücksichtigen.

In den Chapel- und X10-Programmen stellt die Berechnung eines einzelnen Summanden aus Gleichung 5.1.1 einen Task dar. Dabei werden die Knoten des Graphen gleichmäßig auf die zur Verfügung stehenden `Worker` verteilt. Diese ermitteln für jeden Knoten, der ihnen zugewiesen wurde, den BC-Score und tragen die Ergebnisse am Ende zusammen.

Die Erstellung des Graphen erfolgt mit Hilfe eines Zufallsgenerators. Leider verwenden Chapel und X10 unterschiedliche Algorithmen für die Erzeugung von Zufallszahlen, weshalb dafür in beiden Programmen auf nativen C-Code zurückgegriffen werden musste. Der C-Code stellt dafür die Methoden `randomLong()` und `randomDouble()` zur Verfügung, die beide die `rand()`-Methode aus der C Standard-Bibliothek (`stdlib.h`) verwenden. Dadurch kann für die Chapel- und X10-Implementierung sichergestellt werden, dass bei Verwendung des gleichen Startwertes (*seed*) s auch immer derselbe Graph erzeugt wird. Für die Ausführung des Programms wurde $s = 7$ gewählt.

Unbalanced Tree Search

Bei dem Unbalanced Tree Search (UTS) Benchmark wird die Zeit zum vollständigen Durchlaufen eines zufällig generierten Baumes gemessen. Das Konzept des Benchmarks wurde von Stephen Olivier et al. [17] in einer Veröffentlichung aus dem Jahr 2006 vorgestellt. Der verwendete Zufallsgenerator basiert auf dem *secure hash algorithm* (SHA1) und garantiert bei gleichen Parametern für jeden Programmlauf denselben Baum zu erzeugen (deterministisches Verhalten). Als Parameter können der Verzweigungsgrad (branching factor) b, ein Startwert s sowie die Tiefe des Baums (depth) d angegeben werden. Für den Benchmark wurden der Verzweigungsgrad und der Startwert fest auf $b = 2$ und $s = 7$ gesetzt. Die Tiefe variiert in den Testläufen von 1 bis 32 ($d \in \{1, 2, \ldots, 32\}$). Erwartungsgemäß

wäre die Größe des erzeugten Baumes b^d. Allerdings wird bei der Erstellung des Baumes die geometrische Verteilung genutzt, was dazu führt, dass die Anzahl der Kinder einiger Knoten deutlich über b liegt. Daraus folgt ein unbalancierter Baum (vgl. [18], Kap. 2.5.1).

Für die Umsetzung des Benchmarks mit der GLB_CHPL- und GLB_X10-Implementierung wird der gesamte Baum als `TaskBag` dargestellt. Jeder Knoten ist ein Tripel, mit einem Deskriptor, sowie einem unteren und oberen Index. Der Deskriptor beinhaltet den SHA1-Hashwert und der untere bzw. obere Index entspricht dem Start- bzw. Endindex der noch nicht besuchten Kindknoten. Wenn der `TaskBag` aufgeteilt werden soll, so wird für jeden bekannten Knoten $v(d, l, h)$ der Indexbereich der nicht besuchten Kinder geteilt. Also entstehen aus dem Knoten $v(d, l, h)$ die beiden Einträge $v_1(d, l, \hat{h})$ und $v_2(d, \hat{h}, h)$, mit $\hat{h} = \lfloor \frac{h-l}{2} \rfloor$. Der Eintrag v_1 wird weiterhin lokal berechnet, wohingegen v_2 an den anfragenden Rechenknoten weitergegeben wird. Falls kein Knoten mehr als einen nicht besuchten Kindknoten hat, so verbleiben die Berechnungen lokal, da das Senden unnötige Laufzeit kosten würde (vgl. [18], Kap. 2.5.2).

5.2 Ergebnisse

Aufgrund der besseren Übersicht wurden die Benchmark-Ergebnisse in jeweils zwei Diagramme aufgesplittet. Das Erste enthält die Ergebnisse der Benchmarks für einen, vier und 16 Locales/Places. Im Zweiten werden dann die Laufzeitmessungen für 32, 64 und 128 Locales/Places vorgestellt. Die Ergebnisse für zwei und acht Locales/Places wurden ausgelassen, um die Diagramme nicht zu überladen.

Fibonacci

In Abbildung 5.1 und 5.2 sind die Ergebnisse des Fibonacci-Benchmarks zu sehen. Hier fällt auf, dass sich die Laufzeiten mit steigender Anzahl der Rechenknoten bei kleinen Werten für z verschlechtern. Dieses Verhalten lässt sich anhand der wenigen Tasks, die für die Berechnung von kleinen Fibonacci-Zahlen anfallen, erklären. Die Verwendung mehrerer Rechenknoten bringt in solchen Fällen keinen Vorteil, da die Initialisierung und die zusätzliche Kommunikation mehr Laufzeit kostet, als einspart. Mit wachsendem z werden immer mehr Tasks erzeugt, wodurch die verursachten Kosten für die Initialisierung

und Kommunikation vernachlässigbar werden. Zum Beispiel ist die Ausführung des Programms mit $z = 24$ und vier Locales/Places bereits schneller als mit nur einem Locale/Place (siehe Abbildung 5.1).

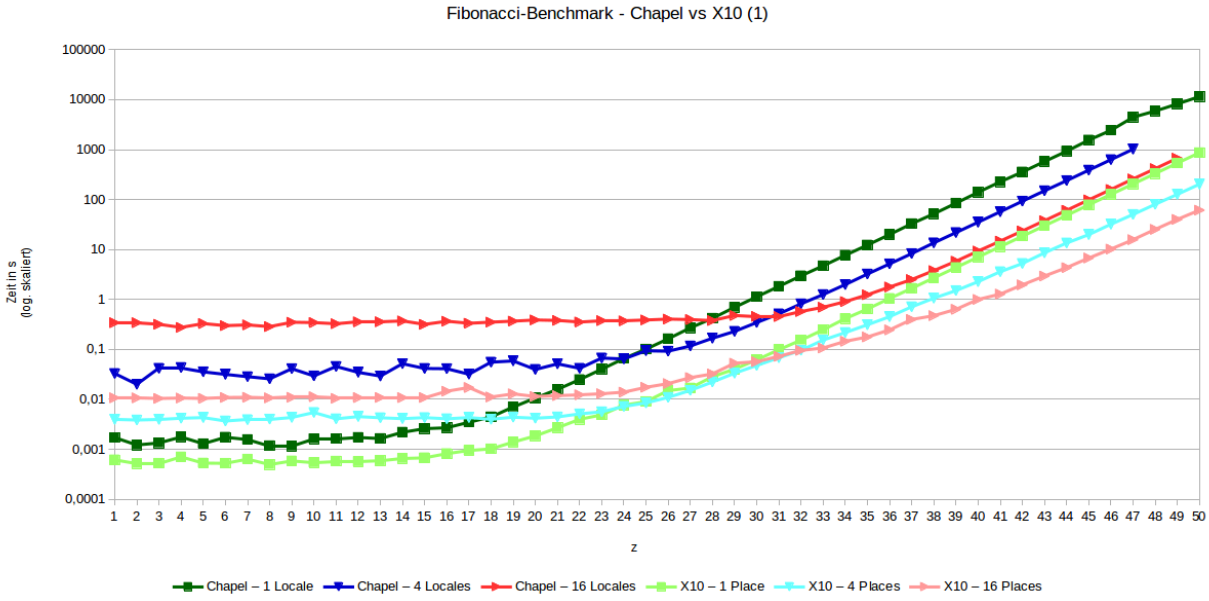

Abbildung 5.1: Fibonacci-Benchmark - Chapel vs. X10 (1)

Ferner ist die deutlich höhere Laufzeit der Chapel-Implementierung im Vergleich zur X10-Implementierung offensichtlich. Das gilt insbesondere für kleine z in Verbindung mit der Verwendung mehrerer Locales. Beispielsweise ist die Berechnung der ersten Fibonacci-Zahl mit 32 Locales ca. um das 40-fache langsamer als in X10 mit 32 Places (siehe Abbildung 5.2). Der Grund für die deutlich schlechteren Laufzeiten, ist die Art, wie Arrays kopiert werden. In Abschnitt 4.2 wurde das `workers`-Array vorgestellt, ein zyklisch verteiltes Array, dass auf jedem Locale genau eine Worker-Instanz enthält. Damit Stehlanfragen gesendet und beantwortet werden können, muss jeder Worker eine Referenz auf alle anderen Workern haben (vgl. Abschnitt 4.1). Ursprünglich wurde dafür eine Methode `setContext(w : [] Worker)` in der Klasse Worker definiert, die als Parameter das verteilte `workers`-Array erhalten hat. Bei dieser Variante waren die Laufzeiten für kleine Werte von z noch knapp um das dreifache langsamer (für $P \geq 2$, mit $P =$ "Anzahl der Rechenknoten"), als in der aktuellen Version. Nach längeren Nachforschungen musste festgestellt werden, dass es derzeit noch nicht möglich ist eine Referenz auf das Array zu übergeben. Auch die Verwendung der Schlüsselworte `ref` und `const ref` (vgl. Abschnitt 2.3) schaffen hier keine

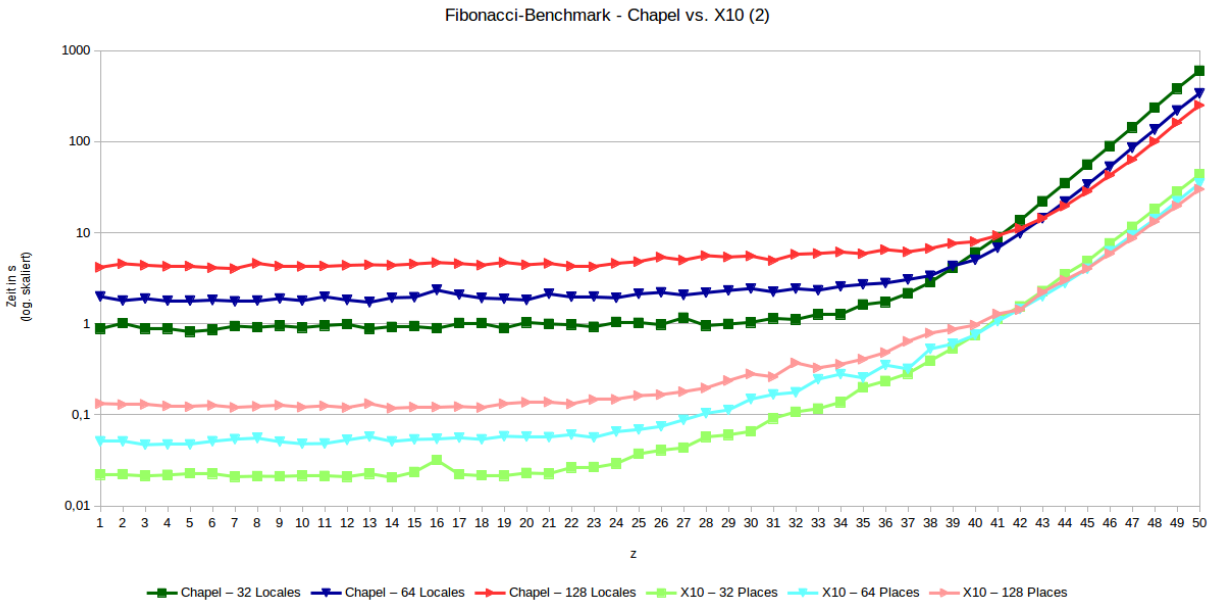

Abbildung 5.2: Fibonacci-Benchmark - Chapel vs. X10 (2)

Abhilfe, wobei leider auch keine Warnung oder Fehlermeldung von Chapel ausgegeben wurde. Stattdessen wird ein Array standardmäßig immer kopiert. Übergibt man das Array beispielsweise einer Methode, welche die Werte in dem Array manipuliert, so wird in der Methode zuerst nur auf einer Kopie gearbeitet. Beim Verlassen der Methode wird das Array wieder zurück kopiert und die neuen Werte werden in das ursprüngliche Array geschrieben. Mit dem Schlüsselwort `in` lässt sich das zurück kopieren zwar verhindern, allerdings war das Programm auch mit dieser Modifikation noch knapp zweimal langsamer als der aktuelle Stand. Um das Kopieren des `workers`-Arrays während des `setContext`-Aufrufs zu verhindern, wurde die Klasse `Context` definiert. Ein Objekt dieser Klasse enthält lediglich das Array und wird stattdessen der Methode `setContext` übergeben. Da bei Klasseninstanzen standardmäßig nur eine Referenz übergeben wird (vgl. Abschnitt 2.3), musste das Array nur noch einmal (während der Zuweisung an das entsprechende Attribut) in der Methode `setContext` kopiert werden. Warum das Kopieren des Arrays so viel Zeit in Anspruch nimmt konnte leider nicht festgestellt werden.

Im Anschluss an die Laufzeitmessungen wurde ein weiterer Versuch durchgeführt, um die Ausführungszeit für die Chapel-Programme ggf. noch zu beschleunigen. Dafür wurde nach dem Vorbild einer bestehenden UTS-Implementierung in Chapel [12] das `workers`-Array (vgl. Abschnitt 4.2) global auf Modulebene definiert. Dies ermöglichte allen Workern einen

direkten Zugriff auf das Array, so dass es nicht mehr kopiert werden musste. Für kleine Werte von z mit mehreren Locales war der Fibonacci-Benchmark ca. viermal schneller. Allerdings wurden dafür die Laufzeiten mit größer werdenden Werten von z immer schlechter. Ab $z \geq 35$ wurde die neue Variante sogar langsamer als die ohne globales `worker`-Array.

Abschließend soll noch einmal auf die fehlenden Messergebnisse der Chapel-Implementierung mit vier und 16 Locales eingegangen werden (siehe Abbildung 5.1). Das Programm ist an diesen Stellen abgestürzt, wobei die Fehlermeldung darauf hinwies, dass versucht wurde zu große Datenmengen zu senden. Der Stacktrace verwies dabei auf den Quelltext der Myrinet-Implementierung. An dieser Stelle wurde zunächst vermutet, dass eine zu große Menge an Tasks gestohlen werden sollte. Jedoch hat eine obere Grenze für die Anzahl der gestohlenen Tasks das Problem nicht behoben. Eine weitere Möglichkeit war, dass der in Abschnitt 4.1 beschriebene `QueueTaskBag` diesen Fehler auslöst. Hier wurde das verwendete Array zum Speichern der Elemente ebenfalls in seiner Größe limitiert. Allerdings hat auch diese Änderung keine Auswirkungen auf den Programmabsturz gehabt.

Betweenness-Centrality

Die Laufzeiten für den BC-Benchmark sind in Abbildung 5.3 und 5.4 aufgeführt. Der erzeugte Graph enthält immer genau 2^z Knoten. Wie zuvor erwähnt wurde, werden die Knoten immer gleichmäßig auf die Worker verteilt. Aufgrund dieser Tatsache fehlen die Messergebnisse an den Stellen, wo die Anzahl der Knoten kleiner ist als die Anzahl der Locales bzw. Places.

Bei den Messungen fällt wieder auf, dass die Laufzeiten für kleine Werte von z und mehrere Locales/Places höher sind als nur mit einem Locale/Place. Zudem ist die GLB_CHPL-Implementierung in diesem Fall wieder deutlich langsamer. Da dies bereits ausführlich für den Fibonacci-Benchmark erläutert wurde, wird darauf im Folgenden nicht weiter eingegangen.

Insgesamt ist die X10-Implementierung auch hier wesentlich schneller als die in Chapel. Abbildung 5.3 zeigt, dass die Laufzeit mit vier Locales (für $z \geq 12$) fast genauso lang ist, wie in X10 mit nur einem Place. Analog dazu, ist die Ausführungszeit bei 16 Locales fast identisch mit der von vier Places. Bei genauerem hinsehen stellt man fest, dass die GLB_X10-

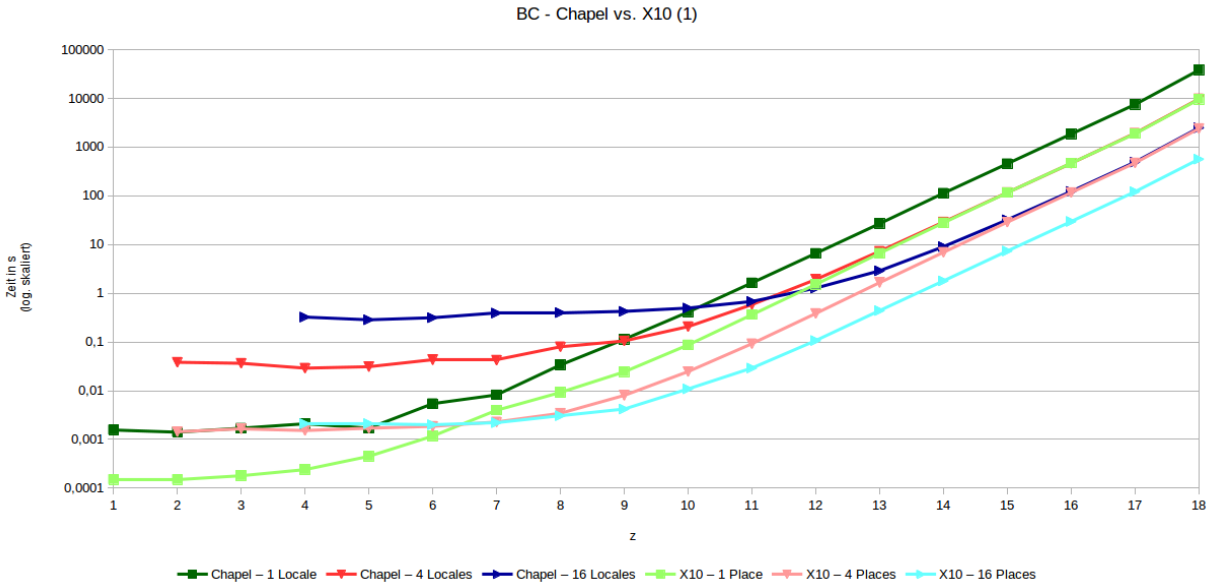

Abbildung 5.3: BC-Benchmark - Chapel vs. X10 (1)

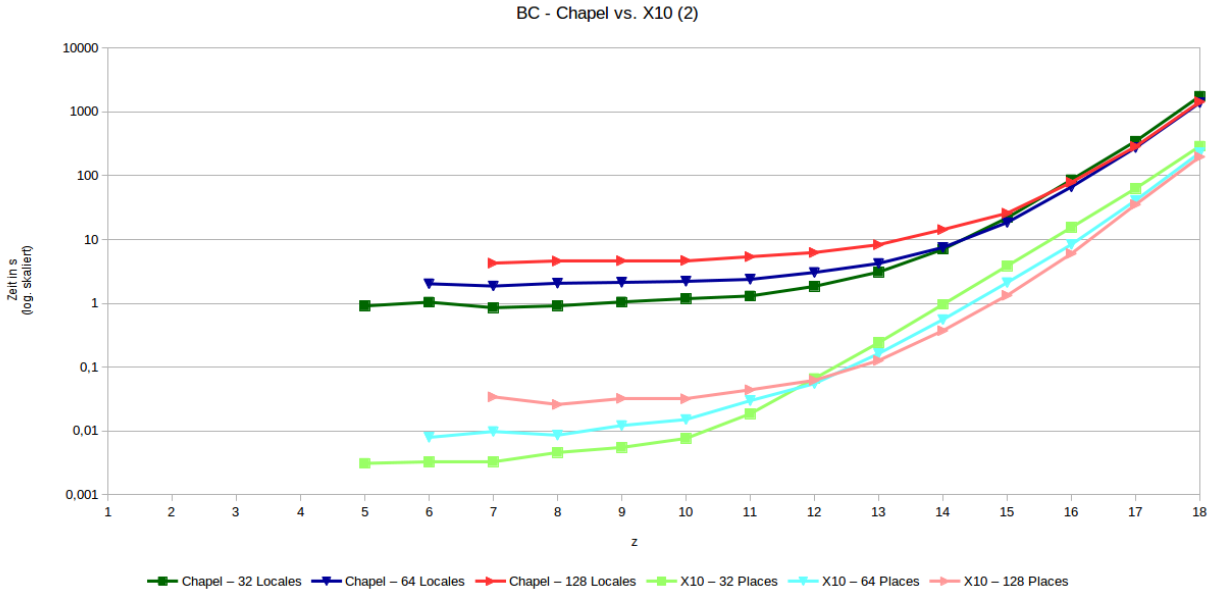

Abbildung 5.4: BC-Benchmark - Chapel vs. X10 (2)

Implementierung ungefähr viermal so schnell ist wie die GLB_CHPL-Implementierung. Allerdings gilt dies nicht mehr, wenn mehr Rechenknoten verwendet werden. In Chapel resultiert aus der Verwendung von mehr als 64 Locales (trotz ausreichender Anzahl von

Tasks) keine weitere Beschleunigung (siehe Abbildung 5.4). Im Gegensatz dazu wird dies in X10 erreicht. Die Laufzeit mit 128 Places ist bei $z \geq 17$ ca. achtmal so schnell wie die mit 128 Locales. Für Werte von z, die kleiner als 17 sind ist dieser Faktor sogar noch deutlich höher.

Unbalanced Tree Search

Die Abbildungen 5.5 und 5.6 zeigen die Ergebnisse des UTS-Benchmarks. Auf die erhöhten Laufzeiten mit mehreren Rechenknoten und kleinen Werten von d wird nicht weiter eingegangen, da dies bereits bei der Auswertung des Fibonacci-Benchmarks umfassend geschildert wurde.

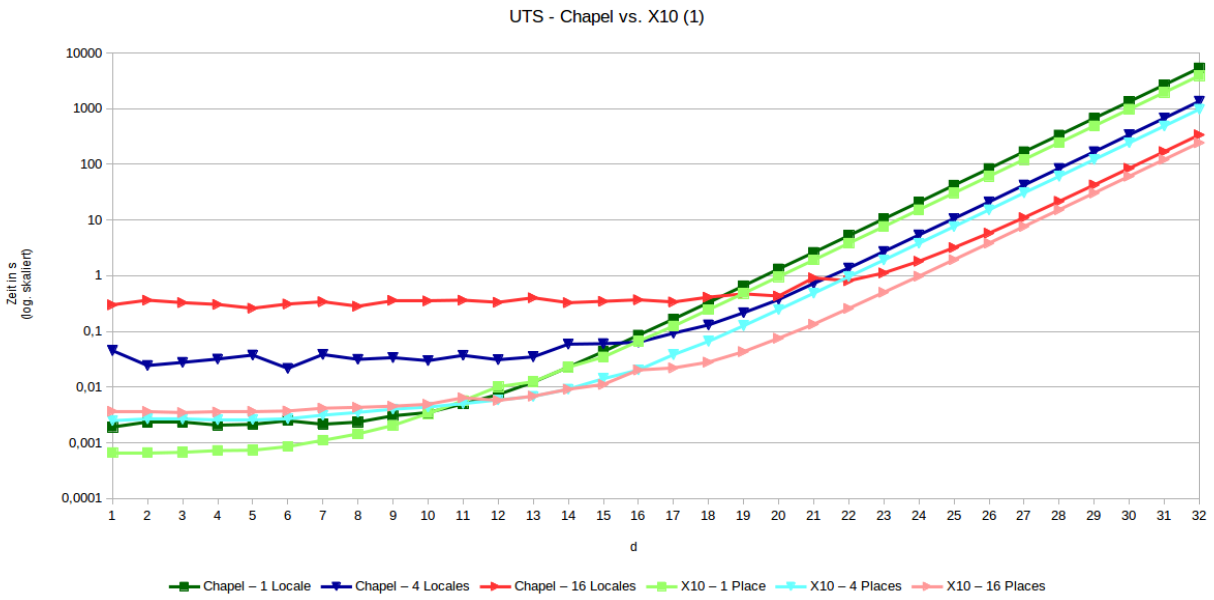

Abbildung 5.5: UTS-Benchmark - Chapel vs. X10 (1)

Von allen Benchmarks ist dies der Einzige, bei dem die Implementierung in Chapel mit der von X10 mithalten kann. Für bis zu 32 verwendete Locales/Places ist die X10 Variante ungefähr 1,5-mal so schnell wie die von Chapel (siehe Abbildung 5.5). Für einen Locale gilt dies bereits für $d \geq 8$. Bei mehreren Locales muss zunächst die Zeit, die während der Initialisierung aufgrund des Kopierens des verteilten Arrays (s.o.) benötigt wird, durch einen deutlich höheren Wert von d ausgeglichen werden. Mit der Verwendung von 64 und 128 Locales/Places erhöht sich dieser Faktor auf zwei bzw. drei (siehe Abbildung 5.6). Auch

in diesem Benchmark lässt die Beschleunigung in Chapel mit mehreren Locales signifikant nach.

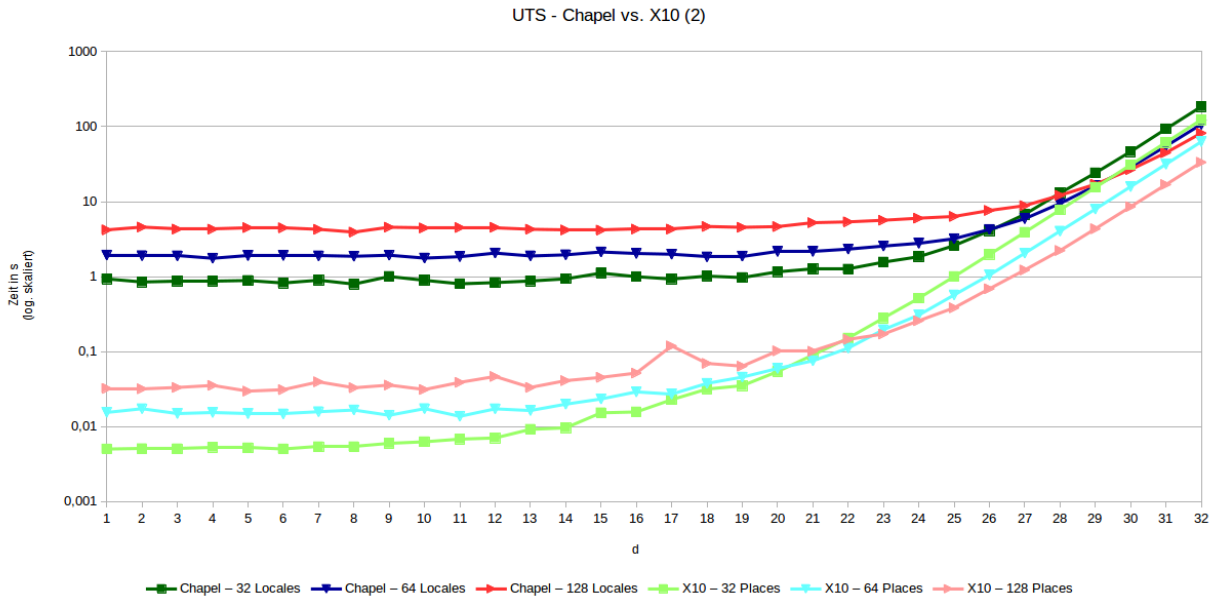

Abbildung 5.6: UTS-Benchmark - Chapel vs. X10 (2)

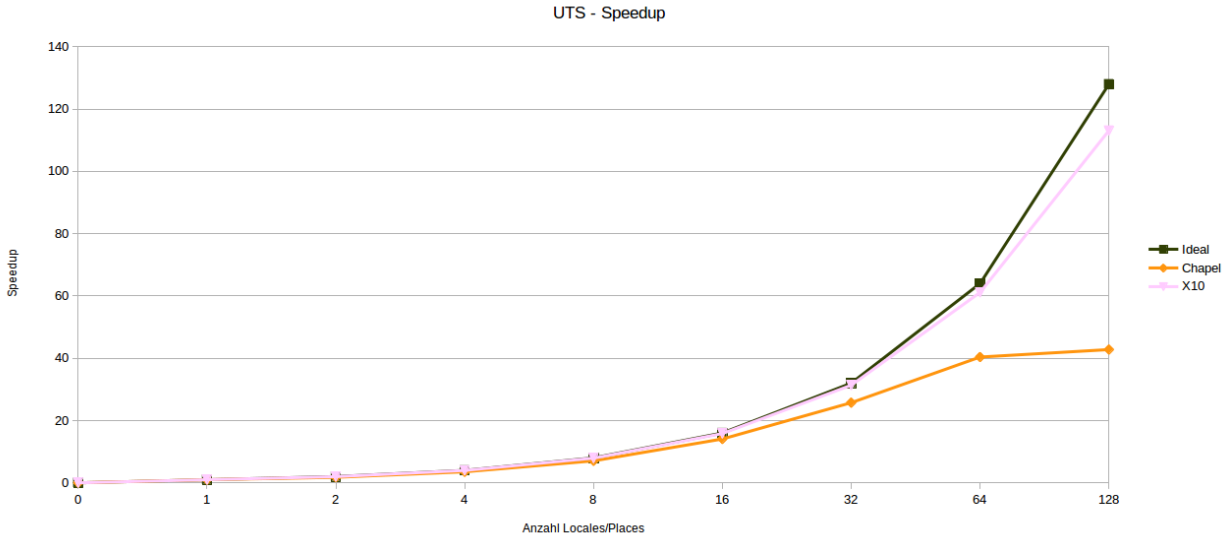

Abbildung 5.7: UTS - Speedup

In Abbildung 5.7 wird die erreichte Beschleunigung (*Speedup*) für den UTS-Benchmark noch einmal grafisch dargestellt. Der Speedup misst dabei das Verhältnis der sequentiellen zur parallelen Ausführung eines Programms. Im Idealfall ist die parallele Ausführung auf p Prozessoren genau p-mal schneller. Für den Speedup-Benchmark wurde $b = 2$ und $d = 30$ gewählt.

Für die X10-Implementierung ist die erreichte Beschleunigung für bis zu 64 Places nahezu Ideal. Erst bei 128 Places lässt der Speedup leicht nach. Die GLB_CHPL-Implementierung hingegen fällt bereits bei 32 Locales zurück. Bei 64 und 128 Locales ist das Programm nur noch knapp 40-mal schneller als die sequentielle Variante. Die Ergebnisse bestätigen was auch in den Benchmarks festgestellt wurde: für eine hohe Zahl von Locales verbessert sich die Laufzeit nur noch geringfügig oder gar nicht mehr.

6 Zusammenfassung und Ausblick

Als wichtigstes Ergebnis dieser Masterarbeit wurde ein Framework zur globalen Lastbalancierung in Chapel implementiert. Als Vorlage dienten das von Saraswat et al. beschriebene Konzept [18], sowie die bereits bestehende GLB-Bibliothek in X10 [23]. Das Konzept umfasst unter anderem die Definition von Lifeline-Graphen. Eine detailgetreue Umsetzung der globalen Lastbalancierung von Saraswat et al. [18] stellte sich allerdings als schwierig heraus, da Chapel zum aktuellen Zeitpunkt noch nicht über alle benötigten Funktionen verfügte. Aufgrund dessen mussten erhebliche Einschränkungen in Bezug auf Usability und Wartbarkeit in der GLB_CHPL-Implementierung im Vergleich zur X10 Variante in Kauf genommen werden. Darüber hinaus konnte Chapel auch im Hinblick auf Performance nicht überzeugen. Die Laufzeiten der X10-Benchmarks sind meist um das vier- bis fünffache schneller als in Chapel. Bereits bei der Installation von Chapel auf dem Hochleistungsrecher der TU Darmstadt traten trotz Hilfe des Supports zahlreiche Fehler auf. Selbst der Start des Programms bedurfte zahlreicher Parameter, die nicht dokumentiert waren und nur über beteiligte Softwareentwickler in Erfahrung gebracht werden konnten. Bei X10 hingegen verlief die Installation und Ausführung der Programme ohne Probleme.

Ein interessantes Thema für weiterführende wissenschaftliche Arbeiten wäre die Erweiterung der GLB_CHPL-Implementierung um Fehlertoleranz. Sollte beispielsweise ein Rechenknoten ausfallen, so gehen die dort berechneten Ergebnisse und noch ausstehenden Tasks verloren. Dafür könnten in bestimmten Zeitabständen Sicherungskopien von allen Rechenknoten zentral oder auch dezentral abgespeichert werden. Zudem müssten diese Kopien bei auftretenden Stehlanfragen aktualisiert werden. Wenn nun ein Rechenknoten ausfällt, könnte auf die Sicherungskopie zurückgegriffen und die Berechnungen fortgesetzt werden. Eine solche Implementierung existiert bereits in X10 [13].

Des Weiteren wäre ein Vergleich mit Charm++ [15] interessant. Charm++-Programme werden in einer besonderen Laufzeitumgebung, dem sog. *Charm++ Runtime System* (CRS),

gestartet. Das CRS sorgt zur Laufzeit dafür, dass die Arbeitslast gleichmäßig verteilt wird und darüber hinaus auch Sicherungspunkte angelegt werden. Damit bietet Charm++ bereits standardmäßig eine globale Lastbalancierung mit Fehlertoleranz. Außerdem analysiert das CRS welche Tasks die meiste Rechenzeit benötigen und verfügt über verschiedene Strategien zur Lastbalancierung. Ein Vergleich zwischen dem seit den 80er Jahren bestehenden Charm++ und den relativ neuen Sprachen Chapel und X10 wäre demnach sehr interessant.

Abbildungsverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] Cray Inc.: *Chapel Language Specification 0.97*. Webseite. <http://chapel.cray.com/spec/spec-0.97.pdf>, Abruf: 20.07.2015

[2] *Make x10.glb safe for multi-threaded places*. Webseite. <https://jira.codehaus.org/browse/XTENLANG-3391>, Abruf: 20.07.2015

[3] *Request for review: Remove GPU support from trunk*. Webseite. <http://sourceforge.net/p/chapel/mailman/message/30760842/>, Abruf: 20.07.2015

[4] *Shadowing a procedure with variable number of arguments*. Webseite. <http://sourceforge.net/p/chapel/mailman/message/33346784/>, Abruf: 20.07.2015

[5] International Business Machines Corporation (IBM): *X10 Language Specification 2.5*. Webseite. <http://x10.sourceforge.net/documentation/languagespec/x10-252.pdf>, Abruf: 20.07.2015

[6] Bader, David A.; Feo, John; Gilbert, John; Kepner, Jeremy; Koester, David; Loh, Eugene; Madduri, Kamesh; Mann, Bill ; Meuse, Theresa: *HPCS Scalable Synthetic Compact Applications #2: Graph Analysis*. <http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.0.pdf>. Version: 2006, Abruf: 20.07.2015

[7] Blumofe, Robert D.; Leiserson, Charles E.: Scheduling Multithreaded Computations by Work Stealing. In: *Journal of the ACM (JACM)* 46 (1999), Nr. 5, S. 720–748

[8] Bonachea, Dan; Funck, Gary: *UPC Language Specifications, Version 1.3*. Webseite. <http://upc.lbl.gov/publications/upc-spec-1.3.pdf>, Abruf: 20.07.2015

[9] Chamberlain, Brad: *Variable Block Distributions*. Webseite. <http://sourceforge.net/p/chapel/mailman/message/33361202/>, Abruf: 20.07.2015

[10] Dijkstra, Edsger W.: A Note on Two Problems in Connexion with Graphs. In: *Numerische Mathematik* 1 (1959), Nr. 1, S. 269–271

[11] Epperly, T.; Prantl, A. ; Chamberlain, B.: *Composite Parallelism: Creating Interoperability between PGAS Languages, HPCS Languages and Message Passing Libraries*. Webseite. <http://chapel.cray.com/papers/CompParallelismProgress.pdf>. Version: September 2011, Abruf: 20.07.2015. – LLNL Project Report

[12] Fohry, Claudia; Breitbart, Jens: *User Experiences with a Chapel Implementation of UTS*. Chapel Implementers and Users Workshop, 2014. Webseite. <http://chapel.cray.com/CHIUW2014.html>, Abruf: 20.07.2015

[13] Fohry, Claudia; Bungart, Marco ; Posner, Jonas: Towards an Efficient Fault-Tolerance Scheme for GLB. In: *Proceeding X10 2015 Proceedings of the ACM SIGPLAN Workshop on X10* (2015), S. 27–32

[14] Freeman, Linton C.: A Set of Measures of Centrality Based on Betweenness. In: *Sociometry*, 40 (1977), Nr. 1, S. 35–41

[15] Kalé, L.V.; Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: Paepcke, A. (Hrsg.): *Proceedings of OOPSLA'93*, ACM Press, September 1993, S. 91–108

[16] Kawachiya, Kiyokuni; Takeuchi, Mikio; Zakirov, Salikh ; Onodera, Tamiya: Distributed Garbage Collection for Managed X10. In: *Proceedings of the 2012 ACM SIGPLAN X10 Workshop* ACM, 2012, S. 5

[17] Olivier, Stephen; Huan, Jun; Liu, Jinze; Prins, Jan; Dinan, James; Sadayappan, P ; Tseng, Chau-Wen: UTS: An unbalanced tree search benchmark. In: *Languages and Compilers for Parallel Computing* (2007), S. 235–250

[18] 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

[19] Saraswat, Vijay A.; Tardieu, Olivier; Grove, David; Cunningham, David; Takeuchi, Mikio ; Herta, Benjamin: *A Brief Introduction To X10 (For the High Performance Programmer)*. Webseite. <http://x10.sourceforge.net/documentation/intro/intro-223.pdf>. Version: September 2012, Abruf: 20.07.2015

[20] Sharma, Aroon; Koehler, Joshua ; Barua, Rajeev: *Affine Loop Optimization using Modulo Unrolling in CHAPEL*. Webseite. <http://chapel.cray.com/CHIUW/2014/Sharma_talk.pdf>. Version: Mai 2014, Abruf: 20.07.2015

[21] Shirahata, Koichi; Doi, Jun ; Takeuchi, Mikio: *Performance Analysis of Lattice QCD in X10 CUDA*. Webseite. <http://x10.sourceforge.net/documentation/presentations/X10DayTokyo2015/x10daytokyo15-shirahata.pdf>. Version: Dezember 2015, Abruf: 20.07.2015

[22] Sidelnik, Albert: *Targeting GPUs and Other Hierarchical Architectures in Chapel*. Webseite. <http://chapel.cray.com/presentations/SC11/05-sidelnik-gpu.pdf>. Version: 2011, Abruf: 20.07.2015

[23] Zhang, Wei; Tardieu, Olivier; Grove, David; Herta, Benjamin; Kamada, Tomio; Saraswat, Vijay ; Takeuchi, Mikio: GLB: Lifeline-based global load balancing library in X10. In: *Proceedings of the first workshop on Parallel programming for analytics applications* ACM, 2014, S. 31–40