Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

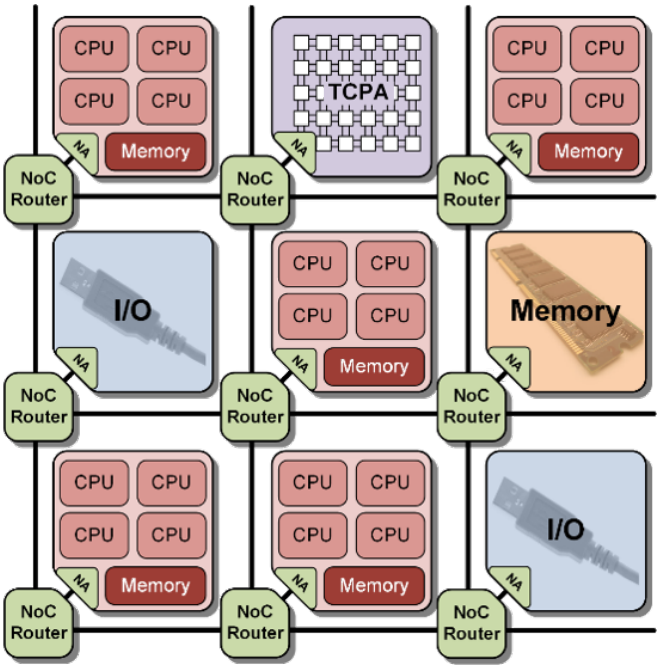
Lehrstuhl Prof. Dr.-Ing. Snelting

Invasives Verteiltes Job Queue Framework

Masterarbeit von

Norman Christopher Böwing

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. Jörg Henkel
Betreuende Mitarbeiter:	Andreas Zwinkau

Bearbeitungszeit: 30. Juni 2015 – 02. Dezember 2015

Zusammenfassung

Um die Ausführungszeit von Anwendungen zu minimieren, ist es nötig, die vorhandenen Prozessoren bestmöglich zu nutzen. Um dies zu erreichen, wird das Problem in Teilprobleme geteilt die dann parallel bearbeitet werden können. Selbst wenn die Teilprobleme die gleiche erwartete Ausführungszeit haben, kann deren tatsächliche Ausführungszeit jedoch stark variieren. Mithilfe dynamischer Lastverteilung werden diese Teilprobleme dann zur Laufzeit effektiv auf die vorhandenen Prozessoren verteilt. Mit *Invasive Computing* soll zukünftig in Zeiten steigender Prozessoranzahl pro Chip ressourcenbewusstes Programmieren möglich sein. Durch die Trennung von Reservierung der Ressourcen, Ausführung und Freigabe der Ressourcen im Invasive Computing ist es einem Programmierer feingranular möglich, die Parallelität seiner Anwendung zu steuern. Nach der Reservierung stehen Prozessoren einer Anwendung exklusiv zur Verfügung, was die Ausführungszeit der Anwendung berechenbarer macht. Diese Arbeit beschäftigt sich mit dem Entwurf und der Implementierung eines Frameworks zur dynamischen Lastverteilung für Invasive Computing. Betrachtet wird dabei der im Invasive Computing erreichbare Speedup gegenüber einer statischen Lastverteilung sowie verschiedenen zentralen Datenstrukturen. Weiter wird die Fähigkeit des entwickelten Frameworks betrachtet, zur Laufzeit eine optimale Anzahl an Prozessoren zu reservieren.

To minimize the execution time of applications in multi- and many-core architectures, parallel computing must be used. This can be done by dividing the job in smaller and more manageable jobs, which can be executed simultaneously. Even if those problems have the same expected execution time, their actual execution time may vary greatly. Dynamic load balancing techniques are used to effectively balance the load between all processors in those applications. The main idea of invasive computing is to introduce resource-aware programming in times of rising numbers of processors on a chip. The division of reservation of resources, execution and freeing of resources in invasive computing offers a programmer a fine-grained specification of the degree of parallelization. After reservation an application can use a processor exclusively. This leads to a more predictable execution time. This thesis proposes a design and implementation of a dynamic load balancing framework for invasive computing. The focus lies on the attained speedup over static load balancing and several central datastructures. Additionally, we focus on how the developed framework can adapt to the optimal number of reserved processors at runtime.

Inhaltsverzeichnis

1. Einführung

Heute existieren in jedem Desktop-PC, in jedem Laptop und in jedem Smartphone Multi- und Manycore-Prozessoren. Um Probleme auf diesen Prozessoren schnell lösen zu können, ist eine parallele Programmierung nötig. Dazu müssen Daten auf die verschiedenen Prozessoren verteilt, die Prozessoren koordiniert und die pro Prozessor erreichten Teilergebnisse am Ende der Berechnung zusammengeführt werden. Das Ziel besteht darin die Arbeit so zu verteilen, dass alle Prozessoren gleichzeitig mit der Bearbeitung fertig sind, um das Problem in möglichst niedriger Zeit zu lösen. Um dies zu schaffen, wird das Problem in Teilprobleme geteilt. Diese Teilprobleme werden dann wiederum auf die einzelnen Prozessoren verteilt. Da aber auf manchen Prozessoren zeitgleich noch etwaige andere Anwendungen laufen können, ist es denkbar, dass einzelne Prozessoren sehr gut ausgelastet sind, während andere fast gar nicht ausgelastet sind. Effizienter wäre eine gleichmäßige Verteilung der Last auf alle verfügbaren Ressourcen. Eine Möglichkeit zur Verteilung der Last ist eine statische Lastverteilung. Bei einer statischen Lastverteilung wird ein Problem vor der Berechnung in kleinere Teilprobleme geteilt, welche dann auf die unterschiedlichen Prozessoren verteilt werden. In einem homogenen System mit gleichmäßiger Auslastung aller verfügbaren Prozessoren und vorhersagbar großen Teilproblemen kann mit dieser Form der Lastverteilung eine gleichmäßige Verteilung der Last erreicht werden. Je heterogener ein System jedoch ist und je schlechter ein Problem balanciert ist, desto schwieriger wird eine gleichmäßige Verteilung der Last mit statischer Lastverteilung. Für Teilprobleme, deren Schwierigkeit a-priori nicht vorhersagbar ist, kann eine statische Lastverteilung keine gleichmäßige Verteilung der Last mehr erreichen. Da in diesem Fall die Schwierigkeit eines Teilproblems erst nach dessen Bearbeitung bekannt ist, kann eine effektive Lastverteilung insofern erst zur Laufzeit erfolgen, indem Last von Prozessoren mit schwierigen Teilproblemen zu Prozessoren mit leichten Teilproblemen verlagert wird. Dies nennt man dynamische Lastverteilung.

In naher Zukunft kann man mit bis zu 1000 Prozessoren auf einem Chip rechnen [1]. Diese Prozessoren werden dann sehr einfach gebaut sein, zudem wird ein Chip eine Vielzahl verschiedener Prozessortypen beinhalten können, die alle verschiedene Aufgaben erfüllen sollen. Um diesen Anforderungen gerecht zu werden, wird es neue Betriebssysteme, neue Hardware und auch neue Programmierkonzepte geben müssen. Gepaart mit der hohen Anzahl an Prozessoren auf einem Chip führt dies zu ganz neuen Herausforderungen und Möglichkeiten bezüglich der Lastverteilung. Während bisher die einer Anwendung zur

[1] http://www.itrs.net/

Verfügung stehenden Ressourcen fest waren, wird dies in Zukunft variabel sein und von weiteren laufenden Anwendungen abhängen. Zu unterschiedlichen Zeitpunkten ist es also denkbar, dass einer Anwendung unterschiedliche Ressourcen zur Verfügung stehen. Des Weiteren war es bisher möglich, dass die vorhandenen Ressourcen von mehreren Anwendungen geteilt wurden, was die tatsächliche Laufzeit von Anwendungen nur schwer vorhersagbar machte. Durch die hohe Anzahl an verfügbaren Ressourcen soll dies jedoch vermieden werden. Die einer Anwendung zugeteilten Ressourcen sollen exklusiv nur von dieser Anwendung nutzbar sein. Dies macht zum einen die Laufzeit von Anwendungen vorhersagbar, da keine anderen Anwendungen interferieren können. Zum anderen entfällt dadurch das Threadmanagment, das dafür zuständig ist, allen laufenden Threads Zugriff auf den Prozessor zu gewährleisten. Diese Exklusivität der Ressourcen führt dann auch dazu, dass Threads nicht mehr unterbrochen werden müssen, um andere Threads ausführen zu können. Dieses neue Programmierkonzept nennt sich *Invasive Computing* [2].

Diese Arbeit setzt an dieser Stelle an, um bisherige Lastverteilungsverfahren auf ihre Einsetzbarkeit mit Invasive Computing hin zu überprüfen. Dabei wird auf die speziellen Anforderungen der neuen Umgebung - wie die Nichtunterbrechbarkeit einer Anwendung sowie die Exklusivität der Ressourcen - Rücksicht genommen. Zum einen soll dadurch ein Problem in seiner Ausführungszeit beschleunigt werden können, in dem dynamische Lastverteilung eingesetzt wird. Zum anderen soll ressourcenbewusstes Programmieren ermöglicht werden, da die Anzahl der reservierten Ressourcen einer Anwendung während der Laufzeit veränderbar ist. In Phasen mit großer Parallelität innerhalb einer Anwendung sollen so mehr Ressourcen angefordert werden, während in Phasen mit niedriger Parallelität diese Ressourcen wieder freigegeben werden. Ziel ist die Implementierung eines Job Queue Frameworks zur verteilten Bearbeitung von unabhängigen Teilaufgaben. Das Framework soll dem Ansatz der ressourcenbewussten Programmierung folgen, sodass einerseits Ressourcen abgegeben werden, wenn die reservierten Ressourcen nicht genügend ausgelastet werden können, und andererseits sollen neue Ressourcen reserviert werden, wenn alle reservierten Ressourcen ausgelastet sind, sodass mehr Ressourcen zu einer verkürzten Bearbeitungszeit führen. Ein weiteres Ziel ist das Framework so zu entwerfen, dass die Veränderung der Ressourcen nicht nur zu fest definierten Zeitpunkten erfolgen soll, sondern zu jeder Zeit möglich ist.

[2] http://invasic.informatik.uni-erlangen.de/en/index.php

2. Grundlagen und Verwandte Arbeiten

Dieses Kapitel enthält Grundlagen, die für das weitere Verständnis der folgenden Kapitel nötig sind. Dies umfasst Grundlagen des Invasive Computing sowie Grundlagen der von invasiven Anwendungen verwendeten Programmiersprache X10. Darüber hinaus wird die verwendete Architektur betrachtet und Merkmale des benutzten Betriebssystems genannt. Des Weiteren werden in diesem Kapitel verschiedene Lastverteilungsstrategien erklärt und gegenübergestellt. Abschließend werden verwandte Arbeiten kurz vorgestellt.

2.1. Invasive Computing

Die Hauptidee von *Invasive Computing* ist es, ressourcenbewusstes Programmieren zu ermöglichen. Da die Ressourcenanforderungen von Anwendungen während der Laufzeit variieren, soll ermöglicht werden, dass Anwendungen den aktuellen Bedürfnissen entsprechend während ihrer Laufzeit Ressourcen reservieren und anschließend wieder freigeben können. Vor rechenintensiven Phasen einer Anwendung wäre es demnach möglich, weitere Ressourcen für den rechenintensiven Abschnitt zu reservieren und zu nutzen. Im Anschluss daran könnten die Ressourcen wieder freigegeben und anderen Anwendungen zur Verfügung gestellt werden. So soll im gesamten System ein möglichst hoher *Speedup* und eine möglichst hohe Auslastung erreicht werden.

2.1.1. Architektur

Die bei Invasive Computing gewählte Architektur ist eine aus einzelnen Kacheln bestehende Architektur. Bei einer aus Kacheln bestehenden Architektur befindet sich eine kleine Anzahl an Prozessoren auf einer Kachel. Innerhalb einer Kachel herrscht zwischen den Prozessoren auf einer Kachel Cache-Kohärenz. Dadurch verhalten sich die Prozessoren auf einer Kachel wie ein Multicore-Prozessor mit gemeinsamem Speicher. Für die Kommunikation zwischen Prozessoren auf unterschiedlichen Kacheln existiert

ein *Network on Chip (NoC)*. Die unterschiedlichen Kacheln können auch heterogen sein. Auch ist es möglich, dass Kacheln keine Prozessoren sondern beispielsweise Speicher enthalten.

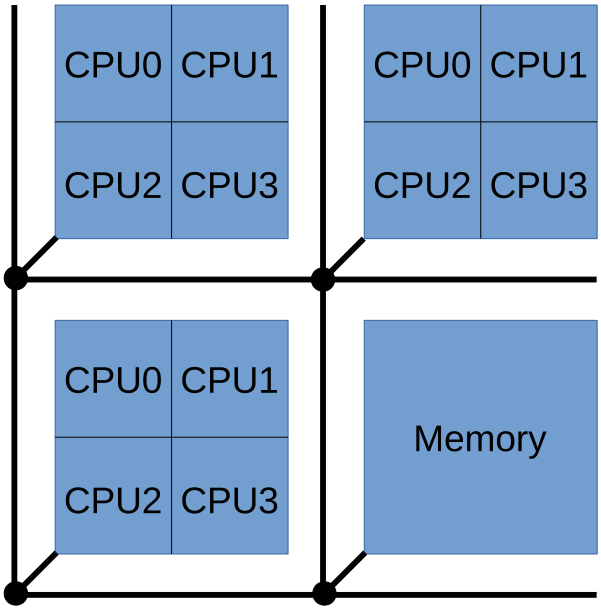


Abbildung 2.1.: Die aktuell verwendete Architektur, die auch in Kapitel 4 für alle Anwendungen verwendet wird. Diese besteht aus aus 4 Kacheln, wovon 1 Kachel nur Speicher enthält und 3 Kacheln jeweils 4 homogene Prozessoren enthalten, die dem Nutzer zur Verfügung stehen. Jede Kachel hat einen weiteren Prozessore der exklusiv für das Betriebssystem reserviert ist.

2.1.2. X10 Grundlagen

Da viele der für Invasive Computing nötigen Konzepte bereits in der Sprache X10 [1] vorhanden sind, wird für die Programmierung invasiver Anwendungen die Sprache X10 verwendet. Dies bezieht sich insbesondere auf die Konzepte *Place*, *APGAS*, async und at. Ein Place bezeichnet einen Bereich des globalen Adressraums. Pro Place läuft eine bestimmte vom Programmierer vorgegebene Anzahl an Threads, die alle ausschließlich auf den Speicher zugreifen können, der dem Place zugeordnet ist. Im Zusammenhang mit der gewählten Architektur wird im Allgemeinen eine 1:1 Abbildung zwischen Kachel und Place verwendet. APGAS steht für *Asynchronous Partitioned Global Address Space* und ist eine Erweiterung des *Partitioned Global Address Space*-Modells (PGAS). Das PGAS-Modell beschreibt einen globalen Adressraum, der in mehrere dedizierte Adressbereiche unterteilt ist. Die Erweiterung kommt durch die Konzepte Place und async zustande,

[1] http://x10-lang.org/

```
val otherPlace = Place.places(1); // get place with id '1'

finish {
    /* create a new activity on 'otherPlace' */
    at (otherPlace) async {
        Console.OUT.println(here); // print the place the
            ↪ activity is running on
    }
} // wait for all spawned activities to finish

Console.OUT.println(here); // print the place the activity is
   ↪ running on
```

Codeausschnitt 2.1: Benutzung der X10 Konzepte Place, finish , at, async und here

```
Place(1)
Place(0)
```

Codeausschnitt 2.2: Ausgabe der Anwendung aus Codeausschnitt 2.1

die das PGAS-Modell um die Möglichkeit erweitern dynamisch mehrere Aktivitäten zu erzeugen [18]. Dabei erzeugt das Schlüsselwort async eine neue Aktivität. Eine Aktivität bezeichnet einen leichtgewichtigen Thread ohne dedizierten Speicherbereich. Alle auf einem Place laufenden Aktivitäten werden auf die unterschiedlichen verfügbaren schwergewichtigen Threads des Betriebssystems verteilt, deren Anzahl vom Programmierer vorgegeben wurde. Um mit der aktuell laufenden Aktivität eine Operation auf einem anderen Place auszuführen, wird das Schlüsselwort at verwendet. Das abstrakte at nutzt die von der X10RT (X10 Runtime) vorgegebene Implementierung für die Kommunikation zwischen Places. Dies kann beispielsweise über MPI, Sockets oder auch DMA geschehen. Soll auf einem Place p eine neue Aktivität gestartet werden kann dies mit at(p) async {..} erreicht werden. Das finish Schlüsselwort erzeugt für die aufrufende Aktivität eine Barriere, die erst passiert werden kann, wenn alle innerhalb des finish {..} gestarteten Aktivitäten beendet worden sind. Die Benutzung dieser Konzepte ist in Codeausschnitt 2.1 zu sehen.

Das Schlüsselwort here gibt immer den Place zurück, auf dem die aufrufende Aktivität aktuell läuft. Das finish auf Place(0) ist erst beendet, wenn alle innerhalb des finish {..} erzeugten Aktivitäten beendet sind. Dies gilt insbesondere auch für die auf Place(1) erzeugte Aktivität, die die Konsolenausgabe auf Place(1) veranlasst.

```
val constraints = new PEQuantity(1, 8);
val claim = Claim.invade(constraints);

claim.infect((id:IncarnationID) => {
    Console.OUT.println(id);
});

claim.retreat();
```

Codeausschnitt 2.3: Invasive Anwendung die den `Constraint` PEQuantity nutzt, der angibt, dass diese Anwendung mindestens 1 und maximal 8 PEs benötigt.

2.1.3. Invasive Anwendungen

Jeder Ablauf einer invasiven Anwendung lässt sich in drei Phasen einteilen. Diese sind der Reihe nach *Invade*, *Infect* und *Retreat*. Ein Invade reserviert die gewünschten Ressourcen. Ein Infect verbreitet den Programmcode auf den vorher reservierten Ressourcen und führt diesen aus. Ein Retreat gibt die reservierten Ressourcen wieder frei. Diese drei Operationen werden auf einem *Claim* ausgeführt. Ein Claim ist ein Objekt, welches der Anwendung den Zugriff auf seine reservierten Ressourcen ermöglicht. Dies können Prozessoren, Speicher oder auch Ein-/Ausgabe sein. Jede Anwendung läuft immer innerhalb eines Claims. Bei Programmstart wird dazu jeder Anwendung ein initialer Claim bereitgestellt, auf den die Anwendung Zugriff hat. Dieser enthält immer 1 `ProcessingElement` (PE). Ein Claim ist nur auf der Kachel gültig, auf der er erstellt wurde, und kann nicht über ein at kopiert werden. Auch die Anforderung eines neuen Claims innerhalb eines anderen Claims ist problemlos möglich. Allerdings kann innerhalb eines Claims nur auf Ressourcen des jeweiligen Claims zugegriffen werden.

Die Anwendung aus Codeausschnitt 2.3 hat den `Constraint` mindestens 1 und maximal 8 PEs zu brauchen. Dieser Constraint wird dem Invade bei der Ressourcenanforderung übergeben. Da die Anzahl an Ressourcen mitunter von der Auslastung des gesamten Systems zur Laufzeit abhängt, ist vorher nicht klar, wie viele Ressourcen reserviert werden können. Die Anzahl der tatsächlich reservierten Ressourcen sind nach erfolgreichem Invade über den Claim abrufbar. Anschließend wird mit dem Infect eine Funktion an alle reservierten PEs übergeben. Das Infect startet dann so viele Inkarnationen wie zuvor PEs reserviert wurden. Jede Inkarnation ist durch eine IncarnationID gekennzeichnet. Diese werden beginnend bei 0 durchnummeriert und enthalten zusätzlich noch das PE, auf dem sie ausgeführt werden. Diese führen daraufhin alle eine Konsolenausgabe aus. Die Ausgabe der Anwendung hängt also nicht nur vom entsprechenden Timing der PEs ab, sondern auch von der Anzahl an reservierten PEs. Bei einer Reservierung von nur einem PE wird dementsprechend nur „0“ ausgegeben. Sollten acht PEs reserviert worden sein, werden die Zahlen 0-7 in beliebiger Reihenfolge ausgegeben. Im Anschluss wer-

den dann die angeforderten Ressourcen wieder freigegeben und die Anwendung läuft im aufrufenden Claim weiter.

Constraints

Ein Constraint kann für eine invasive Anwendung angeben, welche Randbedingungen erfüllt sein müssen, damit diese Anwendung laufen kann. Des Weiteren können Constraints genutzt werden, um die Eigenschaften einer Anwendung genauer zu beschreiben und damit dem Betriebssystem eine bessere Ressourcenzuweisung zu ermöglichen. Es kann zwischen 5 verschiedenen Klassen an Constraints unterschieden werden. Die 4 Klassen **PredicateConstraint**, **SetConstraint**, **PartitionConstraint** und **OrderConstraint** geben Randbedingungen einer Anwendung an. Ein **PredicateConstraint** gibt Randbedingungen für einzelne reservierte PEs an. Ein **SetConstraint** gibt Randbedingungen für eine Menge an PEs an. Ein **PartitionConstraint** gibt Randbedingungen für das Layout der reservierten PEs an und ein **OrderConstraint** ordnet die Liste an verfügbaren PEs nach einem bestimmten Kriterium und wählt zuerst die PEs für eine Reservierung aus, die das Kriterium am besten erfüllen. Die Klasse **Hint** klassifiziert alle nicht notwendigerweise zu erfüllenden Constraints. Eine Übersicht der in dieser Arbeit erwähnten Constraints:

- **Hint**
  - DowneyScalabilityHint
    Gibt mit Hilfe einer Speedup-Kurve an wie skalierbar eine Anwendung ist.
  - TileSharing
    Gibt an, ob eine Kachel mit Claims derselben Anwendung oder Claims anderer Anwendungen geteilt werden darf.
- **PredicateConstraint**
  - ThisPlace
    Es dürfen nur PEs reserviert werden, die auf demselben Place liegen wie der Claim.
- **SetConstraint**
  - PEQuantity
    Gibt die minimale und maximale Anzahl an zu reservierenden PEs an.
- **PartitionConstraint**

```
val claim = Claim.invade(new PEQuantity(1,8)); // reserve 1–8
  ↪ PEs
for(var i:Int = 0; i < COUNT; i++) {
    claim.infect((id:Incarnation) => {
        /* do something in parallel */
    });
    claim.reinvade(new PEQuantity(1,8)); // retreat + invade
}
claim.retreat(); // free resources
```

Codeausschnitt 2.4: Reinvade eines Claims nachdem ein Infect erfolgreich ausgeführt wurde

– PlaceCoherent
  Es dürfen nur PEs auf einem Place reserviert werden.

Reinvade

Reinvade ist ein weiteres Schlüsselwort im Invasive Computing, welches die Phasen Retreat und Invade direkt nacheinander durchführt. Ein Reinvade kann also dazu benutzt werden dem Betriebssystem die Möglichkeit zu geben, die an eine Anwendung zugewiesenen Ressourcen in Anbetracht der aktuellen Auslastung des Gesamtsystems zu überprüfen. Wird das Reinvade mit Constraints aufgerufen, so kann die Anwendung die reservierten Ressourcen in einem Schritt verändern. Dies ermöglicht, dass von der Anwendung genutzter Speicher auf einem Place nicht freigegeben wird. Würde eine Anwendung erst ein Retreat und anschließend ein Invade auf einem Claim ausführen, so würden alle reservierten Ressourcen abgegeben und damit auch der dazugehörige Speicher freigegeben werden. Sobald auf einem Place kein PE mehr reserviert ist, hat die Anwendung keinen Zugriff mehr auf diesen Place und damit auch nicht auf den dort reservierten Speicher. Damit ein Reinvade also keinen Speicher freigibt, der eventuell noch gültige Daten enthält, werden niemals alle reservierten PEs von einem Place bei einem Reinvade abgegeben, da nach einem Reinvade die Anwendung bereits keinen Zugriff mehr auf einen entfernten Place hätte. Ein Reinvade wird daher an Stellen in einer Anwendung genutzt, an denen sich die Parallelität ändert. Im gängigsten Fall wird dabei das Reinvade außerhalb eines laufenden Infect aufgerufen, wie in Codeausschnitt 2.4 zu sehen ist. Durch den wiederholten Aufruf von Reinvade innerhalb der Schleife hat das Betriebssystem die Möglichkeit, vor jedem Infect die Ressourcen für den Claim `claim` optimal zu wählen.

Eine weitere Möglichkeit der Nutzung ist es, ein Reinvade innerhalb eines Infect auszuführen und die Ressourcen zu verändern. Dies ist in Codeausschnitt 2.5 dargestellt. Der

```
val claim = Claim.invade(new PEQuantity(1) && new ThisPlace());
claim.infect((id:Incarnation) => {
        /* only 1 PE acquired here */
        /* do something in serial */

        claim.reinvade(new PEQuantity(2,8) && new PlaceCoherent
           ↪ ());

        /* 2-8 PEs acquired here */
        /* do something in parallel */
    });
claim.retreat();
```

Codeausschnitt 2.5: Während eines laufenden Infects wird auf dem Claim Objekt `claim` ein Reinvade ausgeführt. Die Bearbeitung des ersten Abschnitts erfolgt seriell. Nachdem Reinvade stehen dann mehrere PEs zur Verfügung, sodass mit Hilfe von async neue Aktivitäten auf die PEs verteilt werden können.

`PlaceCoherent` Constraint gibt an, dass alle PEs auf demselben Place liegen müssen, damit im folgenden parallelen Abschnitt async genutzt werden kann, um die erhaltenen PEs zu nutzen. Zu beachten ist, dass die Operationen Infect, Invade, Reinvade und Retreat immer nur von der Kachel aus benutzt werden können, auf der das zugehörige Claim Objekt erstellt wurde. Dies wird hier durch den `ThisPlace` Constraint sichergestellt.

Malleable invasive Anwendungen

Asynchron *malleable* invasive Anwendungen können jederzeit während ihrer Laufzeit durch das Betriebssystem unterbrochen werden[3]. Invasive Anwendungen hingegen ermöglichen Ressourcenveränderungen nur beim Aufruf von Invade, Reinvade und Retreat. Dazu ist es nötig einen Malleable Constraint für die Anwendung anzulegen, der einen *Resize Handler* übergeben bekommt, der das Verhalten bei einer Ressourcenänderungen definiert.

Im Codeausschnitt 2.6 ist die Erstellung einer einfachen Malleable Anwendung gezeigt. Denkbar ist ein Einsatz des Malleable Contraints in einem Queue Framework mit einer Queue pro PE, sodass bei Entfernen eines PEs alle noch in der Queue verbliebenen Elemente zu einem anderen PE transferiert werden. Bei Hinzufügen eines PEs hingegen könnten Elemente aus Queues von anderen PEs zum hinzugefügten PE transferiert werden. Im Gegensatz zu einem Reinvade können durch den Malleable Constraint auch alle PEs auf einem Place entfernt werden, da der Aufruf des spezifizierten Resize Handlers

```
val resizeHandler = (added:List[ProcessingElement,removed:List[
  ↪ ProcessingElement]) =>
    { for(pe in added) {
        // do something for each added pe
    }
    for(pe in removed) {
        // do something for each removed pe
    }
};
val constraints = new PEQuantity(1, 8) && new Malleable(
  ↪ resizeHandler);

val claim = Claim.invade(constraints);
claim.infect((id:IncarnationID) => {
    // do something
});
claim.retreat();
```

Codeausschnitt 2.6: Für die Erstellung eines Claims mit Malleable Constraint ist eine Funktion nötig, die das Verhalten beim Hinzufügen und Entfernen von PEs angibt. Diese wird an den Malleable Constraint übergeben.

eine Reaktion auf eine Ressourcenänderung erlaubt.

2.1.4. OctoPOS

OctoPOS ist ein speziell für Invasive Computing entwickeltes paralleles Betriebssystem [15]. Dabei soll eine heterogene Umgebung über jetzige Standard Multi- und Manycore-Architekturen hinaus unterstützt werden. Zum einen wird eine Abstraktion angeboten, die es dem Programmierer ermöglicht, eine bestimmte Anzahl an PEs anzufordern. Darüber hinaus kann der Programmierer jedoch auch spezifische Hardwareanforderungen stellen, die dann von OctoPOS eingehalten werden müssen. So ist demnach denkbar, dass auf einem *MPSoC* eine bestimmte Anzahl an Prozessoren keine Gleitkommaeinheiten besitzen, der Programmierer jedoch für seine Berechnung zwingend Gleitkommaeinheiten benötigt. Auf jeder Kachel läuft eine OctoPOS Instanz, die ein dediziertes PE zur Verfügung hat, welches der Nutzer nicht reservieren kann.

Jeder in OctoPOS ausgeführte Code läuft in einem *i-let*. Ein i-let ist ein leichtgewichtiger Thread, der auf einem PE ausgeführt wird. Ein i-let kann vom Programmierer durch die Erstellung eines X10 async erzeugt werden. Ein async wird dann direkt auf ein i-let

abgebildet. Dabei bestehen keine Garantien, auf welcher Ressource und zu welcher Zeit das i-let ausgeführt wird.

Ausführungsmodell

Standardmäßig werden alle in einem Claim reservierten Ressourcen für ein Programm exklusiv reserviert. Keine andere Anwendung hat auf diese Ressourcen Zugriff. Zusätzlich haben alle i-lets eine *run-to-completion*-Semantik. Das bedeutet i-lets können nicht während der Laufzeit unterbrochen werden. Dies bewahrt den Programmierer vor Problemen wie *Starvation* und ermöglicht OctoPOS ein einfacheres Scheduling, da nicht mehrere i-lets um ein PE konkurrieren. Starvation tritt auf, wenn ein laufendes i-let nicht mehr ausgeführt wird, da genug andere i-lets zur Ausführung bereit stehen und immer vorher ausgeführt werden. Zudem macht die run-to-completion-Semantik die Ausführungszeit eines i-lets berechenbar.

Scheduling

Aktuell hat jedes PE eine lokale Queue für i-lets. Nachdem ein i-let auf einem Place gestartet wurde, wird es per *Round-Robin* an ein PE zugeteilt und der dazugehörigen Queue angehängt. Ist ein i-let einmal einem PE zugewiesen bleibt es auf diesem PE. Jede Queue hat Platz für 4 i-lets, denen über Hardware i-lets zugewiesen werden können. Jedes weitere i-lets muss über das Betriebssystem in einer Software-Queue gespeichert werden.

2.2. Lastverteilung

Lastverteilung bezeichnet die Verteilung von Last auf die zur Verfügung stehenden Ressourcen. Im invasiven Fall ist dies eine Verteilung der Last auf die reservierten PEs. Diese können sich während der Laufzeit eines Programms ändern. Angestrebt wird eine Verteilung der Last auf alle PEs, sodass die Auslastung der einzelnen PEs gleichmäßig ist und eine möglichst kurze Ausführungszeit erreicht wird.

2.2.1. Statische vs. dynamische Lastverteilung

Grundsätzlich unterscheidet man bei der Lastverteilung zwei grundlegende Strategien. Eine statische Lastverteilung entscheidet vor der Ausführung einer Anwendung, wie die

einzelnen Teilaufgaben innerhalb der Anwendung am besten verteilt werden. Dies hat zur Folge, dass für eine Anwendung, für deren Teilaufgaben keine Aussage getroffen werden kann, bezüglich ihrer Ausführungszeit eine statische Lastverteilung keine zufriedenstellende Verteilung der Last erreichen kann. Ein Vorteil der statischen Lastverteilung ist jedoch, dass während der Laufzeit der Anwendung keine Synchronisation stattfinden muss.

Eine dynamische Lastverteilung hingegen entscheidet während der Laufzeit einer Anwendung, wie die Last am besten verteilt werden soll. Eine dynamische Lastverteilung arbeitet also mit mehr Informationen als eine statische, da auch bei der Ausführung gesammelte Informationen für eine bessere Lastverteilung genutzt werden können. Der Preis dafür ist, dass Synchronisation zwischen den einzelnen Threads nötig wird, um die Last zu verteilen. Dies führt dazu, dass eine statische Lastverteilung bei solchen Problemen effizienter ist, bei denen die Ausführungszeit einzelner Teilaufgaben gut abgeschätzt werden kann, und eine dynamische Lastverteilung hingegen bei solchen Problemen sinnvoller ist, bei denen dies nicht der Fall ist.

2.2.2. Work-Stealing

Work-Stealing ist eine Strategie zur dynamischen Lastverteilung [4]. Bei der Lastverteilung mithilfe von Work-Stealing wird ein Prozessor ohne vorhandene Arbeitspakete versuchen, von einem anderen Prozessor Arbeitspakete zu stehlen. Das Verteilen von Last mit Work-Stealing wird also vom Empfänger initiiert. Die zusätzliche Arbeit des Stehlens wird bei dieser Strategie also nur von Prozessoren geleistet, die selbst keine Arbeitspakete mehr haben. Der Prozessor, von dem etwas gestohlen werden soll, wird als *Opfer* bezeichnet. Work-Stealing Implementierungen unterscheiden sich in der Auswahl eines Opfers und in der Menge an gestohlenen Arbeitspaketen. Die Auswahl des Opfers kann beispielsweise zufällig oder einer bestimmten Hierarchie nach erfolgen. Die Menge an gestohlenen Arbeitspaketen kann fest vorgegeben sein oder auch abhängig von der Anzahl an verfügbaren Arbeitspaketen des Opfers sein.

2.2.3. Work-Sharing

Work-Sharing ist eine weitere Strategie zur dynamischen Lastverteilung. Im Gegensatz zu Work-Stealing wird bei Work-Sharing jedoch die Entscheidung zur Lastverteilung bei Prozessoren mit zu hoher Last getroffen. Das Verteilen von Last mit Work-Sharing wird also vom Sender initiiert.

2.2.4. Work-Stealing vs. Work-Sharing

Bei geringer Systemlast ist eine vom Sender initiierte Strategie einer vom Empfänger initiierten Strategie überlegen. Bei hoher Systemlast ist dies genau umgekehrt [10]. Das liegt daran, dass es bei geringer Systemlast wenige ausgelastete Prozessoren gibt, sodass die vielen stehlenden Prozessoren länger Opfer suchen müssen, von denen Arbeitspakete gestohlen werden können. Ist die Systemlast hoch, gibt es viele ausgelastete Prozessoren und wenige Prozessoren mit niedriger Auslastung. Ein stehlender Prozessor findet so sehr schnell ein Opfer, von dem Arbeitspakete gestohlen werden können. Für Work-Sharing ist es in beiden Fällen genau umgekehrt, da die hoch ausgelasteten Prozessoren die Verteilung der Last initiieren.

Darüber hinaus wurde von Dinan et al. gezeigt, dass eine Work-Sharing-Strategie unter optimal gewählten Parametern eine bessere Performance als eine Work-Stealing-Strategie erreichen kann [9]. Parameter bezeichnen in diesem Fall die Anzahl gestohlener Arbeitspakete und die Auslastung des Systems. Eine Work-Stealing-Strategie jedoch erreicht über eine größere Bandbreite der Parameter eine bessere Performance als eine Work-Sharing-Strategie.

2.2.5. Terminationsdetektion

Ein Teil jeder dynamischen Lastverteilung ist die Terminationsdetektion. Da zusätzlich zu der Bearbeitung von Arbeitspaketen immer wieder Last zwischen einzelnen Prozessoren verteilt werden muss, ist es nötig zu erkennen, wann es im System keine Arbeitspakete mehr gibt, damit die Anwendung beendet werden kann. Denn kein Prozessor kann lokal entscheiden, ob die Anwendung beendet werden kann. Bei einer Work-Stealing-Strategie würden alle Prozessoren nach einer bestimmten Zeit versuchen, Arbeitspakete von anderen Prozessoren zu stehlen. Bei einer Work-Sharing-Strategie würden ebenfalls nach einer bestimmten Zeit versuchen alle Prozessoren darauf warten, von anderen Prozessoren neue Arbeitspakete geschickt zu bekommen. Während beim Work-Stealing eine explizite Terminationsdetektion durchgeführt werden muss, kann beim Work-Sharing das X10 finish verwendet werden, um die globale Termination festzustellen.

2.3. Verwandte Arbeiten

Eine ganze Reihe von Arbeiten befasst sich mit dynamischer Lastverteilung für nicht-invasive parallele Anwendungen auf Rechnern mit gemeinsamen oder verteiltem Speicher. Dinan et al. [9] vergleichen Work-Stealing und Work-Sharing miteinander für eine Architektur mit verteiltem Speicher. Dabei wird für die Kommunikation zwischen den

Knoten MPI eingesetzt. Ravichandran et al. [17] haben Work-Stealing für eine Architektur mit gemeinsamen Speicher und verteiltem Speicher eingesetzt. Auf einem Knoten befinden sich mehrere Prozessoren mit gemeinsamen Speicher, die untereinander auf einem Knoten Work-Stealing betreiben. Sollte ein ganzer Knoten keine verfügbare Arbeit mehr haben, gibt es einen dedizierten Thread pro Knoten, der dann Work-Stealing von einem anderen Knoten versucht. Dabei werden für das Stehlen auf gemeinsamen Speicher und verteiltem Speicher jeweils andere Strategien benutzt. Auch wird die Terminationsdetektion für jede Form des Work-Stealings hier speziell geregelt. Saraswat et al. [19] schlagen eine Lifeline-basierte Lastverteilung vor. Das heißt jeder Knoten ist mit einer bestimmten Anzahl anderer Knoten über eine Lifeline verbunden. Sollte ein Knoten lokal keine Arbeit mehr haben, versucht dieser nur maximal k-mal von anderen Knoten zu stehlen. Danach versucht der Knoten Arbeit von den Knoten zu bekommen, über die er mit seiner Lifeline verbunden ist. Sollte ein Knoten neue Arbeit erhalten kann er diese über seine Lifeline an andere Knoten verteilen, wenn diese vorher bereits nach Arbeit gesucht haben. Vorteil dieses in X10 implementierten Schemas ist der Verzicht auf eine komplizierte Terminationsdetektion, da diese durch das finish Konstrukt in X10 überflüssig ist. Allerdings kann auf jedem Knoten nur maximal 1 Aktivität laufen, damit die Terminationsdetektion funktioniert.

3. Entwurf und Implementierung

Dieses Kapitel beschäftigt sich mit dem Entwurf und der Implementierung des Job Queue Frameworks. Zu Beginn werden daher die Besonderheiten von Invasive Computing im Zusammenhang mit dynamischer Lastverteilung im Allgemeinen und Work-Stealing im Speziellen genannt. Dies umfasst die dadurch entstehenden Herausforderungen sowie die sich daraus ergebenden Möglichkeiten. Darauf folgt eine detaillierte Beschreibung des Entwurfs des Job Queue Frameworks. Zusätzlich werden einzelne Details der Implementierung genauer beleuchtet, sowie die Benutzung des Frameworks anhand kleiner Codeausschnitte gezeigt.

3.1. Invasive dynamische Lastverteilung

Nicht-invasive dynamische Lastverteilung (siehe Unterabschnitt 2.2.1) beschäftigt sich mit dynamischer Lastverteilung in Systemen mit einer festen Anzahl an Ressourcen, auf die eine Anwendung Zugriff hat. Im Invasive Computing allerdings ist es möglich, dass sich die für eine Anwendung zur Verfügung stehenden Ressourcen während der Laufzeit ändern. Es ist also möglich, während der Ausführung mehr Ressourcen zugeteilt zu bekommen, die eine Anwendung auch nutzen können sollte. Zusätzlich besteht die Möglichkeit, dass der Anwendung zugeteilte Ressourcen entzogen werden. Eventuell müssen daraufhin Arbeitspakete neu verteilt werden. Ergänzend soll das Job Queue Framework die dynamische Ressourcenzuteilung bestmöglich nutzen, um eine optimale Anzahl an PEs zu reservieren. Aufgrund der in Unterabschnitt 2.2.1 genannten höheren Robustheit des Work-Stealings gegenüber Work-Sharing bei der Variation von Parametern sowie der besseren Performance von Work-Stealing nutzt das Job Queue Framework Work-Stealing. Ein weiterer Vorteil von Work-Stealing ist, dass Work-Stealing zu einer besseren Lastverteilung führt, wenn viele PEs eine hohe Last aufweisen. Work-Sharing hingegen ist leistungsfähiger, wenn viele PEs eine niedrige Last aufweisen. Durch die Möglichkeit der dynamischen Ressourcenanpassung ist es möglich, während der Laufzeit PEs abzugeben, sodass im Idealfall alle reservierten PEs eine hohe Last aufweisen. Dadurch ist eine Work-Stealing-Strategie einer Work-Sharing-Strategie vorzuziehen.

3.1.1. Schwierigkeiten in Verbindung mit Invasive Computing

Der Ansatz des Invasive Computing sieht vor, dass X10-Aktivitäten bzw. OctoPOS i-lets eine run-to-completion-Semantik haben. Das bedeutet, dass eine einmal gestartete Aktivität nicht unterbrochen werden kann. Dieses Verhalten führt zu Problemen bei der Implementierung von Work-Stealing und Work-Sharing.

Um in einer X10-Anwendung auf einen Place p zugreifen zu können, ist es nötig das at(p) Konstrukt zu benutzen. Dies führt die aktuelle Aktivität auf Place p aus. Damit diese Aktivität allerdings auf Place p überhaupt ausgeführt werden kann, muss ein PE auf Place p gerade ungenutzt sein. Laufen auf jedem PE auf Place p bereits Aktivitäten, so wird die Aktivität an die Queue eines PEs angehängt. Es ist also nicht möglich, jedem dem Job Queue Framework zugewiesenen PE eine Aktivität zuzuweisen, die kontinuierlich Arbeitspakete bearbeitet, bis keine mehr vorhanden sind. Dies würde dazu führen, dass die durch das Work-Stealing gestarteten Aktivitäten zur Lastverteilung erst ausgeführt werden, wenn bereits alle Arbeitspakete abgearbeitet sind. Arbeitspakete könnten also nie gestohlen werden. Der Codeausschnitt 3.1 zeigt wie at(p) genutzt werden kann, um Work-Stealing durchzuführen.

```
def steal() { // steal a job from another place
    val victim = getVictimPlace();
    /* shift activity to place 'victim' */
    val job = at(victim) {
        return stealJobFromQueue(); // steal job from place '
            ↪ victim'
    };
    push(job); // push job to local queue
}
```

Codeausschnitt 3.1: Beispiel für eine Work-Stealing Aktivität

Um trotzdem Work-Stealing nutzen zu können, ist es daher nötig dafür zu sorgen, dass die durch das Work-Stealing gestarteten Aktivitäten während der Laufzeit der Anwendung schnellstmöglich ausgeführt werden. Dies könnte zum einen erreicht werden, indem auf jedem Place p ein dediziertes PE bereitgestellt wird das ausschließlich Work-Stealing Aktivitäten bearbeitet und ansonsten unbenutzt bleibt. Auf einem Place mit k reservierten PEs würden k-1 PEs Arbeitspakete abarbeiten und 1 PE würde das Stehlen auf diesem Place handhaben. Durch das in Abschnitt 2.1.4 beschriebene Round-Robin Scheduling jedoch würde nur jeder k-te Stehlversuch auf dem dedizierten PE bearbeitet werden. Alle anderen Stehlversuche wären auf ausgelasteten PEs und würden somit blockiert werden, bis die Aktivitäten auf diesen PEs beendet werden. Zudem würde dies bei einer optimalen Arbeitsverteilung, in der kein Work-Stealing nötig ist nur noch zu einem Speedup $S = p * k - p$ führen, wobei p die Anzahl von Places ist und k die Anzahl

an PEs pro Place. Im Gegensatz zu dieser dauerhaften Unbenutzbarkeit eines PEs pro Place sieht die in dieser Arbeit implementierte Lösung es vor, die Aktivitäten, die für das Abarbeiten der Arbeitspakete verantwortlich sind, nach einiger Zeit erneut zu starten. Dies ermöglicht das Ausführen der Work-Stealing Aktivitäten während der Laufzeit. Durch dieses Neustarten der Aktivitäten entsteht jedoch ein zusätzlicher Overhead. Der Overhead, der durch ein X10 async entsteht, wurde von Mohr et al. untersucht [14].

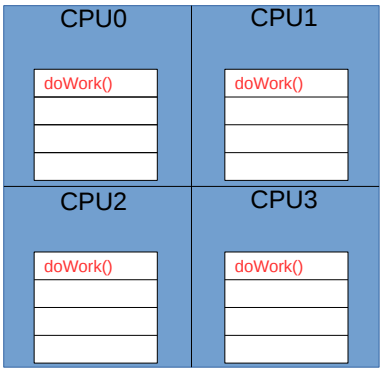
Da eine in Unterabschnitt 2.1.1 beschriebene Kachelarchitektur vorliegt und Work-Stealing auf einem Place auch über gemeinsamen Speicher funktioniert und damit ohne at(p) möglich ist, wird das Neustarten der Aktivitäten nur für das Work-Stealing über Place-Grenzen hinweg nötig. Im Gegensatz zu Work-Stealing innerhalb eines Places ist diese Operation seltener.

```
def doWork(work:(job:Job) -> void) {
    while(true) {
        val job = get();
        if(job == null) {
            // no more jobs in queue
            // ...
        }
        work(job); // work on 'job'
    }
}
```

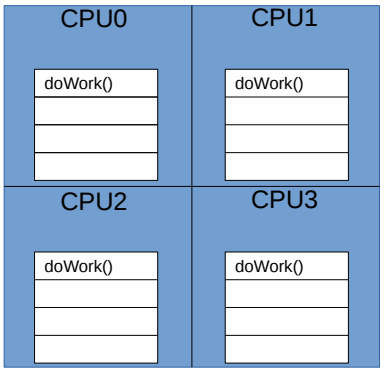
Codeausschnitt 3.2: Beispiel für eine doWork() Aktivität

Das Neustarten der Aktivitäten auf einem Place und die Bearbeitung eines verteilten Stehlvorgangs sind in Abbildung 3.1 illustriert. Beispielcode für die genannten Aktivitäten sind in Codeausschnitt 3.1 und in Codeausschnitt 3.2 abgebildet.

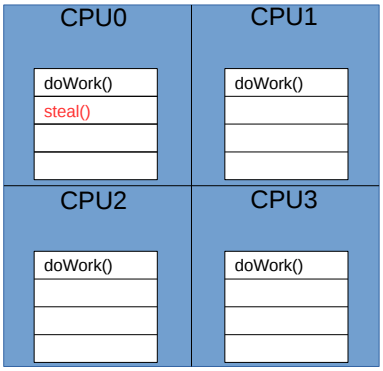
Um den Overhead, der durch das Neustarten der Aktivitäten entsteht, möglichst gering zu halten, wird eine Aktivität nicht nach jedem abgearbeiteten Arbeitspaket neugestartet, sondern erst nach einem Bündel von Arbeitspaketen. Dies reduziert den Overhead des Work-Stealings, erhöht aber gleichzeitig die Zeit, die ein PE für das Stehlen von Arbeitspaketen benötigt. Die optimale Größe des Bündels von Arbeitspaketen hängt zusätzlich von der Bearbeitungszeit eines Arbeitspaketes ab. Je höher die Bearbeitungszeit eines Arbeitspaketes ist, desto geringer ist der prozentuale Overhead, der durch das Neustarten der Aktivität entsteht und desto geringer wiederum sollte die Größe eines Bündels sein, um weiterhin effektiv und schnellstmöglich Arbeitspakete verteilen zu können. Das Job Queue Framework bietet die Möglichkeit, die Zeit zu wählen, nach der eine Aktivität neugestartet wird. Da das Job Queue Framework zusätzlich eine einfache statische Lastverteilung bei Start einer Berechnung durchführt, ist so auch eine effiziente Bearbeitung von balancierten Problemen möglich. Sollte keine Angabe über die Zeit gemacht werden, nach der eine Aktivität neugestartet wird, so misst das Job Queue



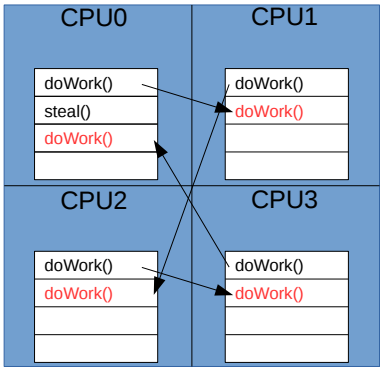
a Starten der *doWork()* Aktivitäten



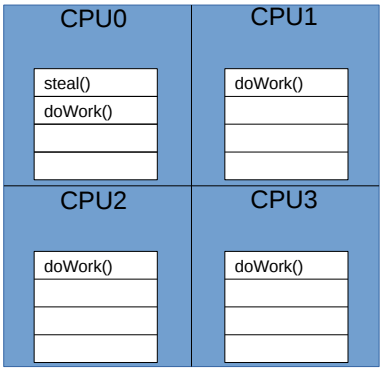
b Bearbeitung von Arbeitspaketen



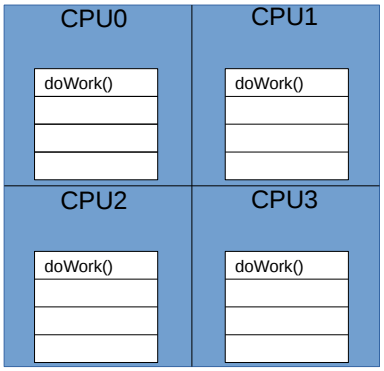
c Eingehende *steal()* Aktivität



d Neustarten der *doWork()* Aktivitäten und Scheduling mit Round-Robin



e Bearbeitung der *steal()* Aktivität



f Stehlvorgang beendet

Abbildung 3.1.: Ablauf eines verteilten Stehlvorgangs

Framework die Zeit, die ein at(p) mit der Serialisierung einer kleinen Liste benötigt. Diese Zeit wird mit der Bearbeitungszeit eines Arbeitspaketes in Verbindung gesetzt,

um eine geeignete Neustartzeit für Aktivitäten zu wählen. Dies soll verhindern, dass der Overhead durch das Neustarten zu hoch wird und gleichzeitig gewährleisten, dass Arbeitspakete effektiv gestohlen werden können, wenn dies notwendig ist.

Die für Work-Stealing beschriebenen Probleme wirken sich auch auf Work-Sharing aus, da bei Work-Sharing Last von Places mit hoher Auslastung auf Places mit geringer Auslastung verschoben werden soll. Angenommen ein System hat 2 Places p und k. Wenn nun eine Aktivität auf Place p entscheidet, dass die Last auf Place k verteilt werden muss, so wird auf Place k die Aktivität an die Queue eines PEs angehängt. Wird at(k) benutzt blockiert die Aktivität auf Place p nun solange, bis das Work-Sharing auf Place k bearbeitet wurde, obwohl auf Place p eigentlich genug Arbeitspakete zu bearbeiten sind, was ein Nachteil gegenüber Work-Stealing ist, da dort eine Aktivität blockiert die keine Arbeitspakete mehr bearbeitet. Eine andere Möglichkeit ist es, at(k) async {..} zu benutzen. Die Aktivität auf Place p bearbeitet in diesem Fall weiter Arbeitspakete, kann dadurch aber keinerlei Informationen über die Last auf Place k erhalten. Ohne diese Informationen kann jedoch keine optimale Entscheidung getroffen werden, auf welchen Place in Zukunft Last verschoben werden soll, da Place p in diesem Fall nur Informationen über die eigene Auslastung und die verschickten Arbeitspakete hat.

3.2. Entwurf

Die Struktur des implementierten Job Queue Frameworks gliedert sich in zwei Aspekte. Dies sind zum einen die auf einem Place verwendeten Datenstrukturen und Konzepte zur Ermöglichung von Work-Stealing innerhalb eines Places. Zum anderen sind es die verwendeten Datenstrukturen und Konzepte, um verteiltes Work-Stealing über Place-Grenzen hinweg zu ermöglichen.

In Abbildung 3.2 ist die verwendete Architektur mit der Lage der Queues zu sehen. Auf jedem Place, auf dem das Job Queue Framework mindestens ein PE reserviert hat, wird für jedes dort befindliche PE eine Queue angelegt. Dies geschieht, um während der Laufzeit durch einen Reinvade, wie in Unterabschnitt 3.3.4 beschrieben, keine neuen Queues anlegen zu müssen. Da X10 keinen lokalen Speicher für Aktivitäten besitzt, ist es nicht direkt möglich jede laufende Aktivität auf einer dedizierten Queue arbeiten zu lassen. Um dieses Problem zu lösen, wird auf jedem Place eine Queue pro vorhandenem PE angelegt. Bei jeder `Enqueue`- und `Dequeue`-Operation wird die Queue des PEs ausgewählt, auf dem die aktuelle Aktivität läuft. Dies ist möglich, da auf jedem PE nur maximal eine Aktivität gleichzeitig laufen kann und somit nie mehrere Aktivitäten gleichzeitig auf derselben Queue arbeiten. Das Job Queue Framework startet auf jedem Place genau 1 Aktivität pro reserviertem PE, um einen höchstmöglichen Speedup erreichen zu können, indem alle PEs ausgelastet werden. Das Starten von k Aktivitäten auf einem Place mit l reservierten PEs mit $k > l$ würde dazu führen, dass die i-let Queues der PEs mehr als 1

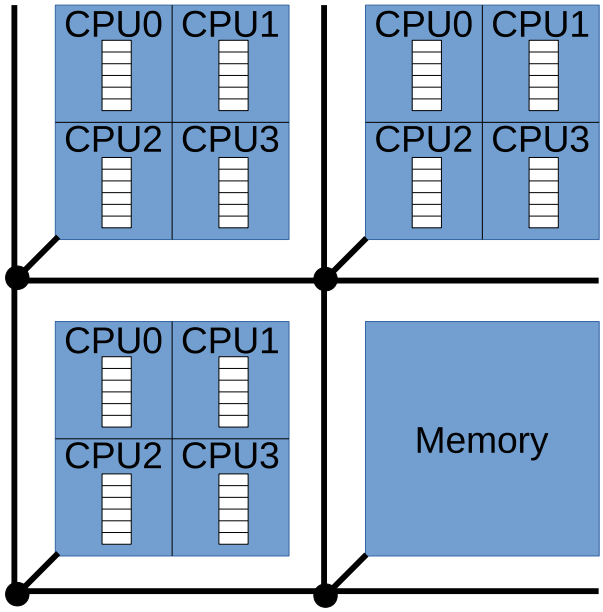


Abbildung 3.2.: Architektur mit den verwendeten Queues für Work-Stealing

i-let enthalten. Dies würde die Dauer eines Stehlvorgangs verlangsamen, da alle i-lets in der Queue abgearbeitet werden müssen, bis der Stehlvorgang bearbeitet werden kann.

3.2.1. Annahmen

Bei der Implementierung des Job Queue Frameworks wurden Annahmen getroffen, die erfüllt sein müssen, damit das Job Queue Framework korrekt arbeiten kann. Diese sind:

- Arbeitspakete müssen unabhängig voneinander sein.
- Während ein `Malleable` Resize Handler bearbeitet wird, werden nur PEs für das Scheduling verwendet, die nach der Abarbeitung des Resize Handlers noch dem Job Queue Framework zur Verfügung stehen. Zu entfernende Places sind davon ausgenommen, um die Daten von diesen Places zu sichern.

Abhängigkeiten können direkt modelliert werden, indem Arbeitspakete erst erzeugt werden, wenn alle nötigen Daten für die Bearbeitung des Arbeitspaketes vorliegen. Komplexere Abhänigigkeiten können zu diesem Zeitpunkt noch nicht modelliert werden. Während der Bearbeitung des Resize Handlers stehen dem Claim die Vereinigung an Ressourcen vor und nach dem Resize zur Verfügung, sodass Daten zwischen den Places ausgetauscht werden können. Das Scheduling auf die PEs zu beschränken, die nach dem abgeschlossenen Resize noch zur Verfügung stehen, ermöglicht es dem Job Queue Framework alle Aktivitäten zu stoppen und Arbeitspakete zwischen Places neu zu ver-

teilen. Das nachträgliche Starten der Aktivitäten allerdings darf keine Aktivitäten auf den zu entfernenden PEs starten, da diese dem Job Queue Framework nicht mehr zur Verfügung stehen, sobald der Resize Handler bearbeitet wurde.

3.2.2. Intra-Place Work-Stealing

Innerhalb eines Places ist das oberste Ziel ein möglichst geringer Overhead bei der Abarbeitung von Arbeitspaketen. Um einem Nutzer die Entscheidung über die Nutzung dieses Frameworks zu erleichtern, muss das Framework bei optimal verteilten Arbeitspaketen eine vergleichbare Leistung liefern wie eine statische Lastverteilung, die die Arbeitspakete optimal auf alle PEs verteilt. Sobald die Arbeitspakete nicht mehr statisch optimal verteilt werden können, sollte das Framework dann einen Speedup gegenüber der statischen Lastverteilung erreichen. Um dies zu erreichen, verteilt das Framework im Vorfeld alle zu Beginn verfügbaren Arbeitspakete gleichmäßig auf alle reservierten PEs. So kann anfangs auf Work-Stealing verzichtet werden.

Für jedes PE auf einem Place gibt es eine dedizierte Queue. Die gerade auf diesem PE arbeitende Aktivität benutzt diese Queue für das Hinzufügen und Entnehmen von Arbeitspaketen. Ist eine Queue leer, so wird die darauf zugreifende Aktivität versuchen Arbeitspakete aus anderen Queues auf diesem Place zu stehlen. Dabei wählt die Aktivität zufällig eine Zielqueue aus der Arbeitspakete gestohlen werden sollen. Dies wird mehrmals versucht. Sollte eine Aktivität bei keiner anderen Queue Arbeitspakete stehlen können, so registriert diese Aktivität sich an einer Barriere und wird geparkt. Dies geschieht solange, bis die letzte noch laufende Aktivität die Barriere erreicht. Zu diesem Zeitpunkt liegen auf diesem Place keine Arbeitspakete mehr, und das Work-Stealing kann lokal auf diesem Place gestoppt werden. Die letzte an der Barriere angekommene Aktivität kann dann zum Inter-Place Work-Stealing übergehen. Diese Aktivität stiehlt nun Arbeitspakete für den ganzen Place von anderen Places. So bleiben die anderen PEs auf diesem Place frei und können eingehende Stehlversuche und Tokens der Terminationsdetektion bearbeiten.

3.2.3. Inter-Place Work-Stealing

Sobald auf einem Place lokal keine Arbeit mehr vorhanden ist, wird eine Aktivität auf diesem Place damit beginnen, Inter-Place Work-Stealing durchzuführen. Diese Aktivität wählt zufällig einen Place aus von dem versucht wird, Arbeitspakete zu stehlen. Diese Work-Stealing Aktivität wird auf dem Ziel-Place eine zufällige Queue auswählen, von der versucht wird Arbeitspakete zu stehlen. Ist dies nicht erfolgreich, werden alle restlichen Queues dieses Places nach Arbeitspaketen durchsucht. Dies wird gemacht, da der durch

at entstandene Overhead sehr hoch ist und es effizienter ist, mehrere lokale Stehlversuche zu unternehmen, bevor versucht wird, von einem anderen Place Arbeitspakete zu stehlen. Sollten Arbeitspakete so gestohlen werden können, werden diese zurück auf den Place in die entsprechende Queue übertragen und das Intra-Place Work-Stealing wird auf diesem Place wieder aktiviert. Zusätzlich werden alle geparkten Aktivitäten wieder gestartet, um erneut Arbeitspakete zu bearbeiten. Falls jedoch keine Arbeitspakete gestohlen werden konnten, so wird zufällig ein neuer Ziel-Place gesucht und der Vorgang wiederholt. Bei k aktiven Places wird $k-1$-mal versucht Arbeitspakete zu stehlen. Sollten alle Stehlversuche fehlschlagen, wird die in Unterabschnitt 2.2.5 beschriebene verteilte Terminationsdetektion gestartet. Zur verteilten Terminationsdetektion wird der Algorithmus von Dijkstra et al. benutzt [6]. Dabei wird auf dem Place p, der die Detektion startet, ein Token erstellt, welches reihum zum nächsten Place geschickt wird, bis das Token einmal alle Places erreicht hat und wieder bei Place p ankommt. Damit ist die Laufzeit der Terminationsdetektion $T(n)$ beschränkt durch $T(n) \in O(n)$ Wenn auf einem Place noch Arbeitspakete vorhanden sind, so schlägt die Terminationsdetektion fehl und die Aktivität beginnt wieder mit dem Stehlen von Arbeitspaketen von anderen Places. Das bei der Terminationsdetektion herum geschickte Token enthält von allen Places die Anzahl an gestohlenen und geschickten Arbeitspaketen. Diese beiden Zahlen müssen nach einem kompletten Durchlauf des Tokens übereinstimmen, bevor das Job Queue Framework beendet werden darf. Sollte diese Anzahl nicht übereinstimmen, so wurden, während das Token versandt wurde, noch Arbeitspakete gestohlen. Diese müssen erst abgearbeitet werden, bevor das Job Queue Framework beendet werden darf.

3.2.4. Alternative Ansätze

Eine Alternative zur Nutzung von einer Queue pro PE ist die Nutzung einer synchronen Datenstruktur pro Place, auf die alle PEs auf diesem Place Zugriff haben. Durch die Verwendung einer synchronen Datenstruktur entfällt die zweistufige Work-Stealing Hierarchie. Work-Stealing wäre dann nur noch zwischen Places notwendig und die lokale Termination auf einem Place kann direkt erkannt werden sobald die Datenstruktur leer ist. Allerdings steigt der Overhead für `Dequeue`- und `Enqueue`-Operationen bei einer synchronen Datenstruktur an, wenn mehr PEs auf die Datenstruktur zugreifen. Diesen Nachteil gibt es bei der Verwendung von einer Queue pro PE nicht. Je kleiner die Arbeitspakete sind, desto größer wird der zusätzliche Overhead, den eine zentrale synchrone Datenstruktur hat, da wiederholt und in kurzen Abständen auf die Datenstruktur zugegriffen wird.

3.3. Implementierung

Das Job Queue Framework wurde in der Programmiersprache X10 implementiert mit dem Ziel, in die Bibliothek des InvasIC-Projektes aufgenommen zu werden. Die Implementierung baut auf bekannten Datenstrukturen und Algorithmen auf, die für die Verwendung mit Invasive Computing angepasst wurden. Dies betrifft die verwendeten Queues, die benutzten Work-Stealing-Strategien sowie die Terminationsdetektion. Zusätzlich gibt es die Möglichkeit die reservierten Ressourcen während der Laufzeit mit einem Reinvade anzupassen. Darüber hinaus soll das Job Queue Framework als asynchron Malleable invasive Anwendung konzipiert werden, sodass jederzeit durch das Betriebssystem Ressourcenänderungen erfolgen können. Auch werden die verfügbaren Constraints im Hinblick auf ihre Kompatibilität mit dem Job Queue Framework hin betrachtet.

3.3.1. Nutzung des Job Queue Frameworks

Im Codeausschnitt 3.3 sieht man die Benutzung des Job Queue Frameworks an einem Beispiel.

Das Job Queue Framework läuft immer in dem gerade aktiven Claim, der benutzt wird, um es zu erstellen. Dies wird gemacht, damit der Programmierer bereits Zugriff auf das Job Queue Framework hat, bevor es mit start (..) aufgerufen wird. Die Methode push(..) kann so vor der Ausführung mit start (..) genutzt werden, um dem Framework Arbeitspakete hinzuzufügen. Des Weiteren ist der gängigste Fall bei der Benutzung des Job Queue Frameworks das Anstoßen der Berechnung und das anschließende Warten auf das Ergebnis. Durch die Benutzung des aktuellen Claims steht dadurch ein zusätzliches PE für das Job Queue Framework zur Verfügung. Denn während der parallelen Berechnung durch ein Infect bleiben die PEs des aufrufenden Claims reserviert.

Sollte der Programmierer das Job Queue Framework in einem dedizierten Claim laufen lassen wollen, so muss er diesen Claim explizit erzeugen und das Job Queue Framework in diesem erstellen. Der Codeausschnitt 3.7 zeigt wie das Job Queue Framework in einem dedizierten Claim gestartet werden kann.

3.3.2. Verwendete Queues

Die dem Job Queue Framework zugrunde liegende Datenstruktur für die Speicherung von Arbeitspaketen ist eine lokal blockierungsfreie *Deque* von Dinan et al. [7, 8], die in Abbildung 3.3 vorgestellt wird. Deque ist die Abkürzung für *Double-Ended-Queue* und bezeichnet eine Queue mit zwei Enden. Diese verwendete *Split Task Queue* ermöglicht

```
private static class Job(left:Float, right:Float, depth:uint)
  ↪ {};
private var queueFramework:DistributedQueueFramework[Job,
  ↪ Double];

public def integrateRange(left:Float, right:Float) {
    val constraints = new PEQuantity(4, 4);

    /* create initial work */
    val initialWork = new ArrayList[Job]();
    /* add some jobs to initialWork */

    /* initialize job queue framework */
    val builder = new JobQueueFrameworkBuilder[Job, Double](
      ↪ constraints);
    queueFramework = builder.setWorkList(initialWork).build();

    /* start job queue framework */
    val total = queueFramework.start((job:Job) => {
        return integrateRange(job.left, job.right, job.depth);
    }, (i:Double, j:Double) => i + j);
    Console.OUT.println("Integral of sin(x) between 0 and 3: "+
      ↪ total+" (should be 1.98999)");
    return;
}

private def integrateRange(left:Float, right:Float, depth:uint)
  ↪ :Double {
    /* do work */

    /* add newly generated jobs to the framework */
    queueFramework.push(new Job(left,   center, depth+1));
    queueFramework.push(new Job(center, right,  depth+1));
}

public static def main(args: Array[String](1)) {
    val work = new Integrate((x:Float) => Math.sin(x), 0.00001f
      ↪ , 9);
    work.integrateRange(0f, 3f);
}
```

Codeausschnitt 3.3: Benutztung des Job Queue Frameworks, um das Integral von $\int_0^3 sin(x)\,\mathrm{d}x$ mit einem rekursiven Algorithmus zu berechnen.

```
/* initialize job queue framework */
val builder = new JobQueueFrameworkBuilder[Job, Double](
  ↪ constraints);
queueFramework = builder.setWorkList(initialWork).build();
```

Codeausschnitt 3.4: Initialisierung des Job Queue Frameworks durch Benutzung des `JobQueueFrameworkBuilder` mit der Übergabe der Constraints `constraints` und den initialen Arbeitspaketen `initialWork`.

```
/* start job queue framework */
val total = queueFramework.start((job:Job) => {
    return integrateRange(job.left, job.right, job.depth);
}, (i:Double, j:Double) => i + j);
```

Codeausschnitt 3.5: Starten des Job Queue Frameworks für Arbeitspakete des Typs `Job`. Für jeden `Job` wird die Funktion `integrateRange(..)` ausgeführt. Die daraus entstehenden Ergebnisse werden mit Hilfe der angegebenen Reduktionsfunktion `(i:Double, j:Double) => i+j` reduziert.

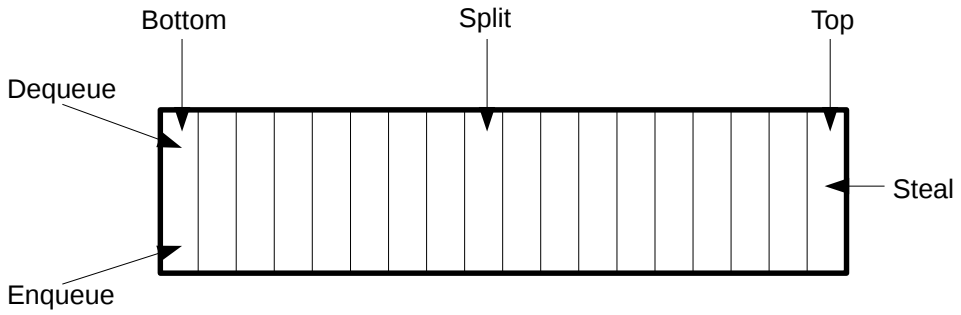


Abbildung 3.3.: Darstellung der Deque von Dinan et al. [7, 8]. Die Deque ist in einen privaten und einen öffentlichen Bereich geteilt. Am unteren Ende der Deque darf nur eine Aktivität arbeiten, die Dequeue und Enqueue Aktionen auf der Deque ausführt. Dies ist der private Bereicht gekennzeichnet durch `bottom - split`. Am oberen Ende der Deque darf eine beliebige Anzahl an Aktivitäten Steal Aktionen auf der Deque ausführen. Dies ist der öffentliche Bereich gekennzeichnet durch `split - top`. Die Größe der Deque ist durch `bottom - top` gegeben.

das Einfügen und Entfernen von Objekten am unteren Ende der Deque, sowie das Entfernen von Objekten am oberen Ende der Deque. Dabei darf nur eine Aktivität gleichzeitig auf das untere Ende der Deque zugreifen. Dies geschieht durch die Teilung der Deque in einen privaten und einen öffentlichen Abschnitt. Das Stehlen von Arbeitspaketen kann parallel ablaufen und wird über den Zugriff auf das obere Ende der Deque realisiert.

```
/* add newly generated jobs to the framework */
queueFramework.push(new Job(left,   center, depth+1));
queueFramework.push(new Job(center, right,  depth+1));
```

Codeausschnitt 3.6: Hinzufügen von neuen Arbeitspaketen vom Typ `Job` zum Job Queue Framework

```
/* create a new claim for the queue framework */
val qfClaim = Claim.invade(new PEQuantity(1));
qfClaim.infect((id:Incarnation) => {
    /* this claim 'qfClaim' will be reinvaded */
    /* the outer claim will remain unchanged */
    val builder = new JobQueueFrameworkBuilder[Job, Double](new
      ↪ PEQuantity(1,8));
    val queueFramework = builder.build();

    /* .. */

    /* run job queue framework in 'qfClaim' */
    val total = queueFramework.start((..) => {
        /* return result per job*/
    }, (i:Double, j:Double) => i + j);
});
qfClaim.retreat();
```

Codeausschnitt 3.7: Starten des Job Queue Frameworks in einem dedizierten Claim. Der äußere Claim bleibt unverändert und eine Anwendung kann so mehrere Job Queue Frameworks parallel benutzen.

Dies wird gemacht, da bei vielen Anwendungen die ersten Arbeitspakete oftmals signifikant mehr Arbeit enthalten als spätere Arbeitspakete, sodass die zuerst eingefügten Arbeitspakete zuerst gestohlen werden. Ein klassisches Beispiel ist eine Integralberechnung, die in jedem Arbeitspaket das zu berechnende Integral in 2 Teile teilt, wenn das Integral noch nicht genau genug ausgerechnet werden kann und für jedes der beiden neuen Integrale ein Arbeitspaket erzeugt. So soll mit möglichst wenigen gestohlenen Arbeitspaketen die Arbeit optimal verteilt werden. Während die Operationen im privaten Teil der Deque ohne Synchronisierungsmechanismen stattfinden, wird im öffentlichen Teil der Deque ein `Lock` eingesetzt, um mehrere parallele Stehlvorgänge synchronisieren zu können. Aufgabe der lokal arbeitenden Aktivität ist es zudem, Arbeitspakete je nach Erforderlichkeit vom privaten in den öffentlich Teil und umgekehrt zu verschieben. Auch dafür wird das `Lock` benötigt. Die Deque wurde zudem um die Möglichkeit ergänzt zwei verschiedene Strategien für das Stehlen zu ermöglichen. Dies bedeutet Intra-Place und Inter-Place kann eine unterschiedliche Anzahl an Arbeitspaketen gestohlen werden.

Alternative Datenstrukturen

Der Einsatz einer synchronen Datenstruktur auf einem Place hat sich als nicht optimal herausgestellt. Ein synchroner Stack und eine synchrone Queue nach Scherer et al. [20], sowie Flat-Combining nach Hendler et al. [12] wurden im Rahmen dieser Arbeit untersucht. Beide haben zwar gute Ergebnisse bei großen Arbeitspaketen gezeigt, konnten aber bei kleinen Arbeitspaketen keine zufriedenstellende Performance erreichen, da der Overhead für `Dequeue`- und `Enqueue`-Operationen zu hoch war. Des Weiteren wurde eine Deque nach Chase et al. [5] untersucht. Diese Deque ist synchron und nicht in 2 Abschnitte geteilt, was zu einem höheren Overhead bei rein lokaler Bearbeitung führt als bei der Split Task Queue von Dinan et al.. Außerdem wird ausschließlich das Stehlen von einem Arbeitspaket unterstützt.

3.3.3. Algorithmen Work-Stealing

Die verwendeten Intra-Place und Inter-Place Work-Stealing-Strategien unterscheiden sich voneinander. Beim Inter-Place Work-Stealing wird ein Ziel-Place zufällig ausgewählt und von dort wird versucht zu stehlen. Sollten dort keine Arbeitspakete zu stehlen sein, wird im Anschluss eine Terminationsdetektion durchgeführt. Das Opfer eines Intra-Place Stehlversuchs wird auch zufällig ausgewählt. Da aber durch einen Reinvade ein PE aus dem Claim entfernt werden kann, ist es nötig, dass Intra-Place jede Deque nach Arbeitspaketen durchsucht wird, bevor das Work-Stealing auf diesem Place beendet werden kann. Zuerst versucht jede Aktivität ohne Arbeitspakete zufällig in anderen Deques Arbeitspakete zu stehlen. Schlägt dies mehrmals fehl, so registriert sich die Aktivität an einer Barriere auf diesem Place. Bevor jedoch die letzte Aktivität das Work-Stealing auf diesem Place beendet, werden alle lokalen Deques nach Arbeitspaketen durchsucht. Dies ist möglich, da zu diesem Zeitpunkt nur noch eine Aktivität läuft. Falls in einer Deque noch Arbeitspakete gefunden werden, so werden alle geparkten Aktivitäten wieder gestartet. So kann garantiert werden, dass alle Arbeitspakete auf einem Place bearbeitet werden. Denn falls durch einen Reinvade ein PE entfernt wurde, muss garantiert werden, dass alle Arbeitspakete von der zu diesem PE gehörenden Deque gestohlen werden, bevor terminiert werden kann. Zusätzlich wird sichergestellt, dass in lokalen Deques nach Arbeitspaketen gesucht wird, bevor der Übergang ins Inter-Place Work-Stealing erfolgt.

Ein weiterer Punkt des Work-Stealings ist die Anzahl gestohlener Arbeitspakete pro Stehlversuch. Diese Anzahl kann getrennt für Intra-Place und Inter-Place Work-Stealing gewählt werden. Standardmäßig wird bei Intra-Place Work-Stealing immer nur eine geringe Anzahl an Arbeitspaketen gestohlen, da die Kosten für einen lokalen Stehlversuch gering sind. Bei Inter-Place Work-Stealing hingegen wird standardmäßig immer die Hälfte der Deque gestohlen, da das verteilte Stehlen einen größeren Overhead hat. Zudem

ist auf jedem Place nur eine Aktivität aktiv, die versucht, Arbeitspakete von anderen Places zu stehlen. Daher wird an dieser Stelle eine größere Menge an Arbeitspaketen gestohlen. Der optimale Parameter für die Menge zu stehlender Arbeitspakete hängt jedoch von der Anwendung ab. Eine schlechte Wahl dieser Parameter kann zu einem hohen Verlust an Performance führen [13]. Beide Parameter können durch den Nutzer bei der Erstellung des Job Queue Frameworks konfiguriert werden.

3.3.4. Dynamische Ressourcenanpassung

Ein Reinvade stellt die Möglichkeit bereit, während der Laufzeit der Anwendung die Anzahl von verfügbaren Ressourcen zu verändern. So kann auf die aktuelle Auslastung des gesamten Systems Rücksicht genommen werden, was dem Ansatz der ressourcenbewussten Programmierung entspricht. Des Weiteren kann das Job Queue Framework sich so optimal an die noch verfügbare Arbeit anpassen, indem Ressourcen freigegeben werden können, wenn ein oder mehrere PEs nicht mehr mit genügend Arbeitspaketen versorgt werden können. Falls das Gesamtsystem noch weitere Ressourcen zu Verfügung hat, können diese dem Job Queue Framework zugeteilt werden. Durch die Unabhängigkeit der Arbeitspakete untereinander kann dann bei einer Erhöhung von k PEs auf l PEs mit $l > k$ im besten Fall ein Speedup von $S = \frac{l}{k}$ erreicht werden.

Sollten bei einem Reinvade dem Job Queue Framework neue PEs zugewiesen werden, so müssen neue Aktivitäten gestartet werden, die auf den neuen PEs arbeiten. Sollten dem Job Queue Framework neue Places zur Verfügung gestellt werden, so muss das Job Queue Framework dort initialisiert werden. Beim Wegfall von PEs und Places muss sichergestellt werden, dass dort vorhandene Arbeitspakete auf andere Places und PEs verteilt werden. Zurzeit ist es allerdings nicht möglich, durch ein Reinvade einen Place zu verlieren, da dann der Speicher auf diesem Place nach dem Reinvade für die Anwendung nicht mehr nutzbar ist. Es gäbe keine Möglichkeit in dem Speicherbereich abgelegte Daten noch zu retten. Dies kann erst durch die Implementierung des in Unterabschnitt 3.3.5 beschriebenen `Malleable` Constraints gelöst werden. Zudem darf die Terminationsdetektion durch den Wegfall von aktiven PEs nicht beeinträchtigt werden. Um dies zu erreichen, müssten beim Wegfall eines Places alle Arbeitspakete und alle Zähler für die Terminationsdetektion auf einen anderen Place verteilt werden. Dazu müssen diese Daten per at auf einen anderen Place verteilt werden, was bei zunehmender Arbeitspaketgröße auch mehr Zeit in Anspruch nimmt. Das Job Queue Framework unterstützt bereits diese Möglichkeit zur Wegnahme von Places, auch wenn diese aktuell noch nicht durch das Betriebssystem unterstützt wird.

Die Entscheidung, ob bei einem Reinvade mehr oder weniger Ressourcen angefordert werden, wird durch eine Metrik bestimmt. Dafür wird die erwartete Zeit berechnet, die eine Aktivität Arbeitspakete stehlen muss, um ein neues Arbeitspaket bearbeiten zu können. Auch die Zeit, die eine Aktivität geparkt ist, fließt in diese Berechnung der

erwarteten Wartezeit mit ein. Gespeichert wird diese erwartete Wartezeit pro PE. Für die Berechnung dieser erwarteten Wartezeit E_i wird ein gewichtetes arithmetisches Mittel mit exponentiell abnehmenden Gewichten genommen. So hat eine aktuell gemessene Verzögerung E_C durch längeres Stehlen mehr Einfluss als weiter zurückliegende eingetretene Verzögerungen. Die durch Stehlvorgänge erwartete Verzögerung zum letzten gemessenen Zeitpunkt $i-1$ wird mit E_{i-1} bezeichnet. Für jedes Arbeitspaket wird dann zum Zeitpunkt i die erwartete Wartezeit berechnet: $E_i = 0.8 * E_{i-1} + 0.2 * E_c$. Je höher diese Zeit E_i ist, desto mehr Zeit verbringt die Aktivität mit dem Stehlen von Arbeitspaketen, da lokal nicht genug Arbeitspakete vorhanden sind. Die Ressourcen können demnach verringert werden. Eine niedrige Zeit ist ein Indikator für eine hohe Auslastung des PEs, auf dem die Aktivität läuft, da das PE genug Arbeitspakete besitzt und selten neue Arbeitspakete stehlen muss. Bei optimal verteilter Arbeit ist diese Zeit 0. Ist die durchschnittliche Zeit für das Stehlen von Arbeitspaketen weitaus geringer als die durchschnittliche Bearbeitungszeit eines Arbeitspaketes, können die Ressourcen erhöht werden.

Probleme

Die aktuelle Implementierung des Reinvade im OctoPOS Betriebssystem führt dazu, dass alle laufenden Aktivitäten gestoppt werden müssen, bevor das Reinvade ausgeführt werden kann. Bevor also ein Reinvade durchgeführt werden kann, muss auf jedem Place ein Flag gesetzt werden, um alle Aktivitäten für die Dauer des Reinvades zu stoppen. Dies dauert eine gewisse Zeit, was zu häufige Reinvades kontraproduktiv macht. Dafür überprüft jede Aktivität periodisch, ob dieses Flag gesetzt ist und stoppt sich, falls das Flag gesetzt ist. Wie lange das Stoppen aller Aktvitäten dauert ist vom Scheduling der Aktivität abhängig, die auf jedem Place dieses Flag setzt. Um die Aktivitäten so selten und so kurz wie möglich zu stoppen, überprüft eine Aktivität auf dem Place, auf dem das Job Queue Framework erstellt wurde die aktuelle erwartete Wartezeit pro PE und berechnet daraus einen neuen Constraint für den nächsten Reinvade. Erst wenn dieser Constraint sich vom letzten verwendeten Constraint unterscheidet, werden alle Aktivitäten gestoppt und ein Reinvade versucht. Nachdem durch das Reinvade neue Aktivitäten gestartet wurden, werden alle gestoppten Aktivitäten wieder gestartet.

3.3.5. Besonderheiten von Invasive Computing

Für das Job Queue Framework können mehrere Constraints genutzt werden, die den Einsatz des Job Queue Frameworks flexibler machen. Dazu gehören:

- **TileSharing**
  Mit dem Constraint TileSharing ist es möglich, dass auf einer Kachel, auf der eine

Anwendung läuft, noch weitere Anwendungen PEs erhalten.

- **PotentiallyMorePEs**
  Eine Anwendung mit PotentiallyMorePEs kann PEs einer Anwendung mit dem Constraint PotentiallyLessPEs benutzen.

- **PotentiallyLessPEs**
  Eine Anwendung mit PotentiallyLessPEs kann PEs mit einer Anwendung mit dem Constraint PotentiallyMorePEs teilen.

- **Malleable**
  Einer in Abschnitt 2.1.3 beschriebenen Anwendung mit dem `Malleable` Constraint können jederzeit durch das Betriebssystem PEs weggenommen oder hinzugefügt werden.

Von diesen 4 Constraints wird der `TileSharing` Constraint jederzeit durch das Job Queue Framework benutzt. Dies stellt durch die dynamische Lastverteilung des Work-Stealings kein Problem dar. Im schlimmsten Fall benutzen auf einer Kachel andere Anwendungen den Bus mit und hindern das Job Queue Framework am Stehlen von Arbeitspaketen auf dieser Kachel.

Die Nutzung der Constraints `PotentiallyMorePEs` und `PotentiallyLessPEs` kann durch die in Abschnitt 2.1.4 beschriebene run-to-completion-Semantik zu Problemen führen. Bei der Benutzung von `PotentiallyLessPEs` könnte eine andere Anwendung eine lange laufende Aktivität auf den PEs des Job Queue Frameworks laufen lassen. Sollte eine durch das Job Queue Framework gestartete Aktivität auf demselben PE angestoßen werden, so kann das Framework nicht terminieren, solange diese Aktivität nicht bearbeitet wurde. Der Constraint wird daher nicht vom Job Queue Framework selbst benutzt, kann aber durch den Anwender gesetzt werden. Um die Bearbeitung zu beschleunigen, müssten bei der Nutzung von `PotentiallyMorePEs` zusätzliche Aktivitäten gestartet werden, die auf den zusätzlichen Ressourcen laufen können. Auf jede gestartete Aktivität muss allerdings bei der Terminierung gewartet werden. Diese zusätzlichen geteilten PEs können allerdings bei dem Stehlen von Arbeitspaketen helfen, wenn dort keine neuen Aktivitäten gestartet werden, da so eingehende Aktivitäten auf einem Place schneller bearbeitet werden. Eine Nutzung des `PotentiallyMorePEs` Constraints ergibt daher in jedem Fall Sinn, da die geteilten PEs nur jeweils für kurze Zeit durch das Job Queue Framework belegt werden und somit der Claim mit dem Constraint `PotentiallyLessPEs` kaum verlangsamt wird. Zudem wird die Dauer von Stehlvorgängen verkürzt, da das Job Queue Framework nur so viele Aktivitäten startet, wie es fest reservierte PEs hat. Eingehende Stehlvorgänge können also schneller bearbeitet werden, falls die andere laufende Anwendung nicht dauerhaft die geteilten PEs belegt.

Der `Malleable` Constraint ist für das Job Queue Framework immer nutzbar. Sollten durch das Betriebssystem PEs hinzugefügt werden, so können jederzeit neue Aktivitäten auf den entsprechenden Places gestartet werden. Falls Ressourcen entfernt werden, so wird einerseits eine Aktivität für das Bearbeiten von Arbeitspaketen beendet und zusätzlich auf dem betroffenen Place der Zähler für aktive Aktivitäten angepasst. Das Hinzufügen von PEs auf einem neuen Place p stellt auch kein Problem dar. Diese können einfach über at(p) async { .. } gestartet werden. Das Entfernen von einem Place allerdings kann lange dauern, da alle dort liegenden Arbeitspakete und Zähler für die Terminationsdetektion zu einem anderen Place transferiert werden müssen. Daher ist eine Einführung eines neuen Constraints `KeepPlaces` sinnvoll. In Verbindung mit dem `Malleable` Constraint könnte dieser genutzt werden, um zwar einzelne PEs jederzeit abgeben und hinzufügen zu können, aber einmal erhaltene Places zu behalten. Des Weiteren wäre die Einführung eines asynchronen Reinvade denkbar, sobald der `Malleable` Constraint implementiert wurde. Dadurch könnte das asynchrone Reinvade lediglich neue Constraints formulieren, während eine etwaige Änderung der Ressourcen zu einem späteren Zeitpunkt durch den Resize Handler des `Malleable` Constraints erfolgt.

4. Evaluation

Dieses Kapitel beinhaltet die Evaluation des implementierten Job Queue Frameworks im Hinblick auf den vorhandenen Overhead durch die dynamische Lastverteilung sowie die Skalierungseigenschaften des Job Queue Frameworks. Zu Beginn werden dafür die verwendeten Beispielanwendungen für die Evaluierung kurz erläutert, ehe das Job Queue Framework in puncto Overhead, Skalierung, Speedup und dynamischer Ressourcenanpassung evaluiert wird. Alle Evaluationen in dieser Arbeit wurden mit der OctoPOS Version vom 20. Oktober 2015 durchgeführt.

4.1. Testsysteme

Für die Evaluierung des Job Queue Frameworks wurden zwei Testsysteme verwendet, um ein möglichst vollständiges Bild über die Leistung des Job Queue Frameworks zu bekommen. Das in Unterabschnitt 4.1.1 genannte Testsystem bildet dabei die vorgestellte Architektur exakt ab, enthält allerdings leistungsschwache Prozessoren. Das in Unterabschnitt 4.1.2 beschriebene Testsystem hingegen verwendet aktuelle Hardware, aber bildet die vorgestellte Architektur nicht ab. So soll in der Evaluierung sowohl der Einfluss der Architektur als auch der Einfluss der Hardware ersichtlich werden.

4.1.1. CHIPit

Das CHIPit ist eine auf Xilinx Virtex 5 LX 330 FPGAs basierende FPGA-Plattform [22]. Das zum Testen verwendete System besteht aus 4 Kacheln. 3 Kacheln besitzen jeweils 4 Prozessoren. Auf der vierten Kachel befindet sich Speicher. Die Kacheln sind über ein Network-on-Chip verbunden [11]. Alle Prozessoren sind unmodifizierte Gaisler SPARC V8 LEON 3 Prozessoren [1] [21]. Die Prozessoren laufen mit 25Mhz, haben einen 16KiB Befehlscache und einen 8KiB L1 Datencache. Zusätzlich teilen sich die 4 Prozessoren auf einer Kachel einen 64KiB 4-Wege L2 Cache. Die X10 Anwendungen wurden mit einem modifizierten X10 Compiler kompiliert [2].

4.1.2. Linux x86 Gastsystem

Das zum Testen verwendete x86 Gastsystem hat einen Intel Core i7-2600 @3,40Ghz mit 4 Kernen und Hyperthreading. Das System hat 16GB Hauptspeicher. Die X10 Anwendungen wurden mit einem modifizierten X10 Compiler kompiliert [2].

4.2. Anwendungen

Für die Evaluation des Job Queue Frameworks wird auf 2 Anwendungen zurückgegriffen. Dies sind die *Integrate-Anwendung* und der *Unbalanced Tree Search Benchmark*(UTS). Beide Anwendungen erlauben es die Größe der Arbeitspakete zu variieren. So bedeutet eine angegebene *Job size k*, dass die eigentliche Berechnung innerhalb eines Arbeitspaketes k-mal durchgeführt wird und nicht, dass das Arbeitspaket k-mal größer ist.

4.2.1. Integrate-Anwendung

Die Integrate-Anwendung berechnet die Integrale verschiedener Funktionen rekursiv, sodass ein Arbeitspaket angibt welcher Bereich einer Funktion integriert werden soll. In jedem Integrationsschritt werden die Werte an beiden Intervallgrenzen berechnet und die Differenz davon mit einem Grenzwert verglichen. Unterscheiden sich beide Werte um mehr als den Grenzwert, wird das Intervall zweigeteilt und 2 neue Arbeitspakete erzeugt. Wenn die Differenz beider Werte kleiner als die vorgegebene Genauigkeit ist, wird das Ergebnis als das Mittel beider Werte zurückgegeben. Dies führt dazu, dass nahezu alle Arbeitspakete die gleiche Größe haben. Abhängig von der zu integrierenden Funktion kann für diese Integralberechnung eine statische Lastverteilung zu einer optimalen Arbeitsverteilung genutzt werden. Auch können Funktionen berechnet werden, bei denen eine einfache gleichmäßige Verteilung der Arbeitspakete zu einer ungleich verteilten Last führt, da jedes Integral nur bis auf die festgelegte Genauigkeit berechnet wird.

4.2.2. Unbalanced Tree Search Benchmark

Der Unbalanced Tree Search Benchmark dient zur Evaluation paralleler Anwendungen, die dynamische Lastverteilung benötigen [16]. Ziel des UTS Benchmarks ist es die Anzahl der Knoten eines erzeugten Baumes zu zählen. Die erzeugten Bäume können in Form, Tiefe, Größe und Balanciertheit parametrisiert werden. Dabei enthält jeder Knoten alle nötigen Informationen, um alle seine Kindknoten zu erzeugen. Dadurch ist es möglich den Baum in jeder möglichen Reihenfolge zu erzeugen, solange Eltern- vor Kindknoten

erzeugt werden. Die Balanciertheit drückt aus, inwieweit die Größe einzelner Teilbäume sich unterscheiden kann. Eine statische Lastverteilung führt zu einer ineffizienten Lastverteilung, da die Größe der Teilbäume nicht vorher berechenbar ist, sodass die vorhandene Arbeit nicht optimal auf die reservierten PEs verteilt werden kann. Der UTS Benchmark unterstützt zwei Typen von Bäumen, die erzeugt werden können. Zum einen sind dies Binomialbäume, in denen ein Knoten mit einer gegebenen Wahrscheinlichkeit p m Kindknoten hat und mit der Wahrscheinlichkeit 1-p keine Kindknoten hat. Die erwartete Arbeit für jeden Knoten des Baumes ist identisch, was eine effektiven Lastverteilung erschwert. Der zweite Typ sind geometrische Bäume. In geometrischen Bäumen ist die Anzahl an Kindern für einen Knoten k von der zu Grunde liegenden Verteilungsfunktion abhängig. Anders als bei Binomialbäumen ist die erwartete Größe eines Teilbaums für Knoten nahe der Wurzel höher als bei Knoten, die weiter von der Wurzel entfernt sind.

4.3. Overhead und Skalierung

Für die Evaluierung des Overheads des Job Queue Frameworks wurde die Integrate-Anwendung genutzt, um ein Integral der Sinus-Funktion zu berechnen. Da die Sinus-Funktion 2π-periodisch ist, kann bei einer Nutzung von k PEs und einem Intervall von $[0, 2\pi * k]$ jedes PE die gleiche Anzahl an Arbeitspaketen mit den gleichen Ergebnissen und der gleichen Dauer erhalten. Wenn die PEs von $0..k-1$ durchnummeriert sind, kann PE_i das Intervall von $[2\pi * i, 2\pi * (i+1)]$ berechnen. Die Last für dieses Problem ist so optimal verteilt. Dies kann direkt mittels claim. infect () implementiert werden. Zusätzlich wurde die Größe eines Arbeitspaketes variiert, um den relativen Overhead des Job Queue Frameworks bei gegebener Arbeitspaketgröße zu erhalten. Für die Bestimmung des Overheads wurden die in Unterabschnitt 4.1.2 und Unterabschnitt 4.1.1 beschriebenen Systeme zur Evaluierung des Job Queue Frameworks auf einem Place mit 4 PEs verwendet. Der Overhead bei der Verwendung von 3 Places und 12 PEs wurde mit dem in Unterabschnitt 4.1.1 beschriebenen System evaluiert. Alle Zeiten wurden innerhalb der Anwendung mit System.currentTimeMillis() und System.nanoTime() gemessen und umfassen die parallele Berechnung, das Erstellen des Claims und die Aufteilung der Arbeitspakete.

Des Weiteren wurde die Skalierung des Job Queue Frameworks gemessen, in dem für ein gegebenes Problem die Anzahl an verfügbaren PEs gesteigert wurde. Im besten Fall, sofern die Problemgröße dies zulässt, kann so eine lineare Skalierung erreicht werden. Für diese Evaluierung wurde die Sinus-Funktion für die Integrate-Anwendung ausgewählt, die das Job Queue Framework mit $k \in [1, 12]$ PEs berechnet hat.

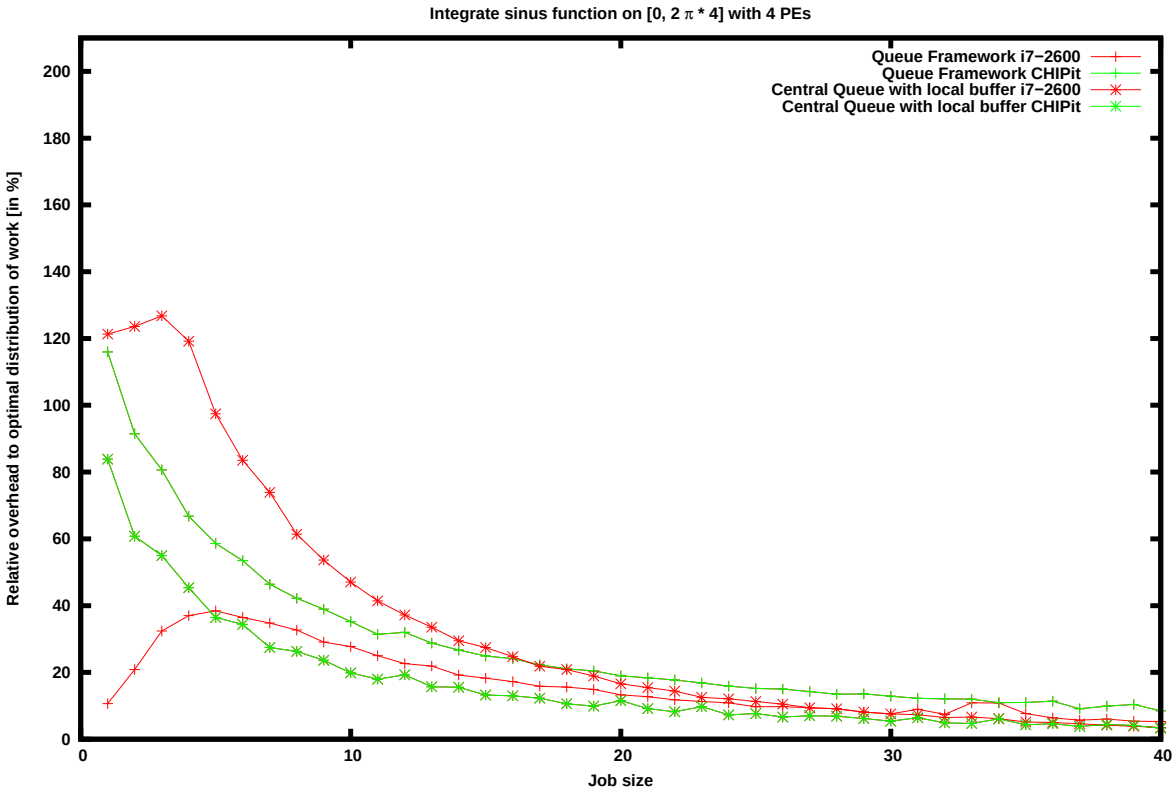


Abbildung 4.1.: Relativer Overhead des Job Queue Frameworks und einer zentralen Queue mit lokalem Puffer gegenüber der Verwendungen von claim. infect () bei 4 PEs und 1 Place. Bei steigender Größe der Arbeitspakete sinkt der Overhead des Job Queue Frameworks sowie der zentralen Queue drastisch. Eine *Job size* von k bedeutet in diesem Fall für die Berechnung eines Intervalls wurde k-mal Math.sin(x) aufgerufen, um die Größe der Arbeitspakete zu erhöhen.

4.3.1. Overhead

Die Abbildung 4.1 zeigt den Overhead des Job Queue Frameworks bei der Nutzung von 1 Place und 4 PEs. Daran wird deutlich, dass das Job Queue Framework auf dem x86 Testsystem mit i7-2600 einen deutlich geringeren Overhead hat als eine zentrale Queue mit lokalem Puffer. Bei der zentralen Queue mit lokalem Puffer puffert jedes PE auf einem anderen Place als Place 0 bis zu dreißig Arbeitspakete. Erst wenn ein PE mehr als dreißig Arbeitspakete in seiner lokalen Queue liegen hat, werden diese wieder in die zentrale Queue eingefügt. Zudem wird jede Aktivität auf Place 0 nach fünf bearbeiteten Arbeitspaketen neugestartet, um die eingehenden Enqueue- und Dequeue-Operationen von Aktivitäten von anderen Places zu bearbeiten, wenn mehr als 1 Place benutzt wurde. Besonders bei sehr kleinen Arbeitspaketgrößen ist der Overhead des Job Queue Frame-

works um ein vielfaches geringer als bei einer zentralen Queue mit lokalem Puffer. Bei zunehmender Arbeitspaketgröße nimmt der Overhead des Job Queue Frameworks sowie einer zentralen Queue mit lokalem Puffer stark ab. Auf dem CHIPit Testsystem ist die Situation jedoch genau umgekehrt. Da die einzelnen Prozessoren auf dem CHIPit nur mit 25Mhz getaktet sind und die Bearbeitung der Arbeitspakete daher sehr lange dauert, wirken sich die Nachteile der zentralen Queue, nämlich die Nutzung der *Compare-and-Set*-Instruktion, kaum auf die Laufzeit aus. Auf dem i7-2600 führt dies zu einer starken Verlangsamung, da das Compare-and-Set durch die Kürze der Arbeitspakete sehr oft fehlschlägt und wiederholt werden muss, insofern die Aktivitäten die meiste Zeit damit verbringen, Arbeitspakete aus der Queue zu entnehmen und einzufügen. Daher ist damit zu rechnen, dass der Overhead des Job Queue Frameworks sinkt, sobald für Invasive Computing leistungsfähigere Hardware eingesetzt wird. Zudem könnte der Overhead des Job Queue Frameworks noch weiter verringert werden, wenn anstelle von at und async C-Implementierungen der beiden Konstrukte genommen werden, wie von Mohr et al. überprüft [14]. So könnte auch auf die Registrierung der Aktivitäten in einem finish verzichtet werden, da das Framework sowieso eine Terminationsdetektion durchführen muss. Auf dem Testsystem mit Intel Core i7-2600 ist bei der Erhöhung der Job size erst ein kleiner Anstieg im Overhead erkennbar, bevor dieser bei weiterer Erhöhung der Job size sinkt. Dies liegt daran, dass die Arbeitspakete in diesem Fall so klein sind und daher so schnell bearbeitet werden können, dass das Starten der zweiten, dritten und vierten Aktivität auf Place 0 weitaus mehr Zeit in Anspruch nimmt. Das Framework sowie die zentrale Queue können dann am Ende der Berechnung Arbeitspakete, die an die vierte Aktivität verteilt wurden, an bereits fertige Aktivitäten verteilen.

Die Abbildung 4.2 zeigt den Overhead des Job Queue Frameworks bei der Nutzung von 3 Places und 12 PEs. Hier ist bereits erkennbar, dass das Job Queue Framework einen weitaus kleineren Overhead gegenüber der zentralen Queue mit lokalem Puffer hat. Dies liegt an der langen Ausführungszeit von at(p) und dem zusätzlich nötigen Neustarten der Aktivitäten auf Place 0, auf dem sich die zentrale Queue befindet.

4.3.2. Einfluss der Arbeitspaketgröße

An den Messungen in Abschnitt 4.3 ist zu erkennen, dass die Arbeitspaketgröße einen großen Einfluss auf den relativen Overhead des Job Queue Frameworks hat. Für ein Arbeitspaket müssen bei der Benutzung von PEs auf nur einem Place 3 Schritte getan werden:

1. Dequeue eines Arbeitspaketes aus der Queue

2. Bearbeitung des Arbeitspaketes

3. Im Fall der Integration: Reduktion des Ergebnisses des Arbeitspaketes mit dem

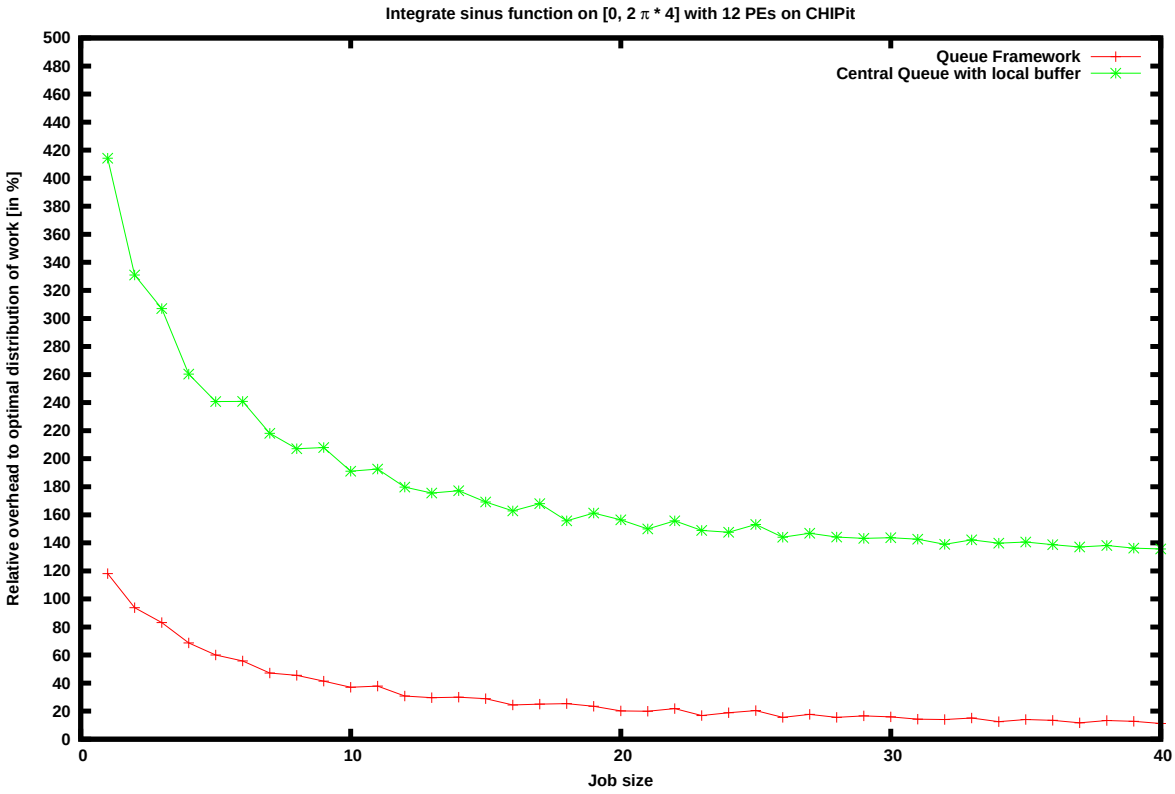


Abbildung 4.2.: Relativer Overhead des Job Queue Frameworks und einer zentralen Queue mit lokalem Puffer gegenüber der Verwendungen von claim. infect () mit 12 PEs und 3 Places. Bei steigender Größe der Arbeitspakete sinkt der Overhead des Job Queue Frameworks sowie der zentralen Queue drastisch. Eine *Job size* von k bedeutet in diesem Fall für die Berechnung eines Intervalls wurde k-mal Math.sin(x) aufgerufen, um die Größe der Arbeitspakete zu erhöhen.

aktuellen lokalen akkumulierten Ergebnis. Für die Integration ist das die Addition zweier Teilergebnisse.

Bei der Benutzung von PEs auf mehreren Places müssen zusätzlich alle laufenden Aktivitäten nach einer Menge an bearbeiteten Arbeitspaketen neugestartet werden.

	Dequeue	Integration	Reduktion
Intel Core i7-2600	283ns	430ns	234ns
AMD Opteron 2220	880ns	1309ns	734ns
Intel P8700	919ns	1251ns	824ns
CHIPit	49410ns	115006ns	19199ns

Tabelle 4.1.: Aufteilung der Bearbeitungszeit pro Integrate-Arbeitspaket

	Dequeue	Erzeugen von Kindknoten	Reduktion
Intel Core i7-2600	285ns	637ns	239ns
AMD Opteron 2220	864ns	1321ns	695ns
Intel P8700	943ns	1391ns	838ns
CHIPit	58310ns	218586ns	19180ns

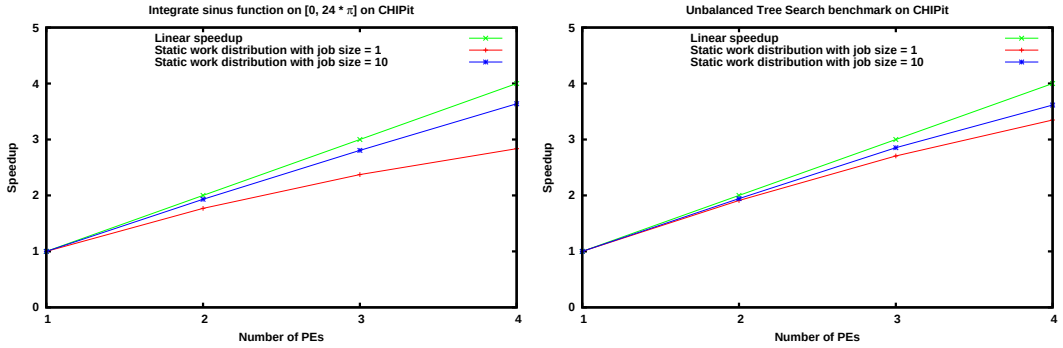
Tabelle 4.2.: Aufteilung der Bearbeitungszeit pro UTS-Arbeitspaket

In Tabelle 4.1 wird die Bearbeitungszeit eines Integrate-Arbeitspaketes mit der Bearbeitungszeit der restlichen Programmabschnitte verglichen. Dabei ist erkennbar, dass die Integrate-Arbeitspakete sehr klein sind und die Integration 45% der gesamten Bearbeitungszeit pro Arbeitspaket auf dem Testsystem mit Intel Core i7-2600 und 63% der gesamten Bearbeitungszeit pro Arbeitspaket auf dem CHIPit Testsystem ausmachen. Dies spiegelt sich auch in Abbildung 4.1 und Abbildung 4.2 wider. Die gesamte Ausführungszeit des Job Queue Frameworks ist im ungünstigsten Fall 118% der Laufzeit eines claim. infect () mit optimaler Arbeitsverteilung. Wenn das Integrate-Arbeitspaket um den Faktor fünf (das entspricht einer Integrate Job size von 40) vergrößert wird, liegt dieser Overhead bereits nur noch bei 11%. Bei der Nutzung von nur einem Place liegt der Overhead im ungünstigsten Fall bei 116% auf dem CHIPit und fällt bei Erhöhung der Arbeitspaketgröße schnell auf unter 10%. Für eine zentrale Queue mit lokalem Puffer liegt der Overhead hingegen bei bis zu 430% auf dem CHIPit und fällt bei der Nutzung von mehreren Places nicht unter 137%. Auf dem Testsystem mit Intel i7-2600 liegt der Overhead des Job Queue Frameworks im schlechtesten Fall nur bei 38% und fällt bei Erhöhung der Arbeitspaketgröße sehr schnell auf unter 10%.

Durch eine Erhöhung der Arbeitspaketgröße kann also der Overhead des Job Queue Frameworks sowie jener zentralen Queue mit lokalem Puffer verringert werden. Dies kann bei der Integrate-Anwendung geschehen, indem bei rekursiver Berechnung nicht nur 1 Intervall, sondern direkt mehrere Intervalle berechnet werden. Beim UTS Benchmark könnten mehrere Knoten in einem Arbeitspaket bearbeitet werden, um die Größe der Arbeitspakete zu erhöhen. Das CHIPit Testsystem benötigt für ein UTS-Arbeitspaket doppelt so lange wie für ein Integrate-Arbeitspaket, während das Testsystem mit Intel Core i7-2600 48% länger für ein UTS-Arbeitspaket braucht, wie in Tabelle 4.2 zu sehen ist. Der Einfluss des Overheads auf die gesamte Ausführungszeit ist daher beim UTS Benchmark kleiner als bei der Integrate-Anwendung.

4.3.3. Skalierung

Bei der Bearbeitung von unabhängigen Arbeitspaketen auf k PEs ist es optimal, wenn ein zusätzliches PE einen Speedup von $\frac{k+1}{k}$ bringt. Um die Skalierungseigenschaften des Job Queue Frameworks zu evaluieren, wird daher der erreichte Speedup des Job Queue



a Skalierung der Integrate-Anwendung auf einem Place

b Skalierung des UTS Benchmarks auf einem Place

Abbildung 4.3.: Bei statischer Lastverteilung ohne Kommunikation zwischen laufenden Aktivitäten ist ersichtlich, dass weder die Integrate-Anwendung noch UTS einen optimalen linearen Speedup erreichen kann. Beide Anwendungen sind speicherintensiv durch das Erstellen, Verwalten und anschließende Bearbeiten von Arbeitspaketen, sodass der Speedup den das vierte PE auf einem Place bringt nur noch sehr gering ist. Dies verbessert sich bei steigender Arbeitspaketgröße, da die Anwendung dann rechenintensiver wird.

Frameworks mit dem einer statischen Lastverteilung verglichen. Die Abbildung 4.3 zeigt die Skalierungseigenschaften der beiden zur Evaluierung benutzten Anwendungen. Dort ist erkennbar, dass der limitierende Faktor bei der Skalierung beider Anwendungen die Speicheranbindung für einen Place ist. Denn auf dem CHIPit Testsystem liegt der Speicher für alle Places auf einer Kachel, die über ein NoC angebunden ist. Besonders bei kleiner Arbeitspaketgröße kann in diesem Fall kein linearer Speedup erreicht werden. Zudem ist der Heap pro Place auf 61MB begrenzt, was bei einer hohen Anzahl an erstellten Arbeitspaketen dazu führt, dass der Garbage Collector sehr häufig laufen muss. Dies verlangsamt beide Anwendungen zusätzlich.

Die Abbildung 4.4 zeigt den Speedup des Job Queue Frameworks mit der Integrate-Anwendung bei inkrementeller Erhöhung der PEs von 1 auf 12. Bei 4, 8 und 12 PEs ist jeweils eine deutlich langsamere Steigerung erkennbar, da die Speicheranbindung einen besseren Speedup verhindert. Des Weiteren ist die Steigerung bei mehr als 4 PEs niedriger, da das Neustarten der Aktivitäten dann nötig ist, was weiteren Overhead hinzufügt. Auch hier führt eine Erhöhung der Arbeitspaketgröße zu einem höheren Speedup.

Die Abbildung 4.5 zeigt, dass das Job Queue Framework für das Hinzufügen eines weiteren PEs einen vergleichbaren Overhead benötigt wie die Nutzung einer statischen Lastverteilung und der Benutzung von claim. infect (). In dieser Evaluation berechnen alle PEs das Integral über ein 2π langes Intervall der Sinus-Funktion. Dabei wurde die durch-

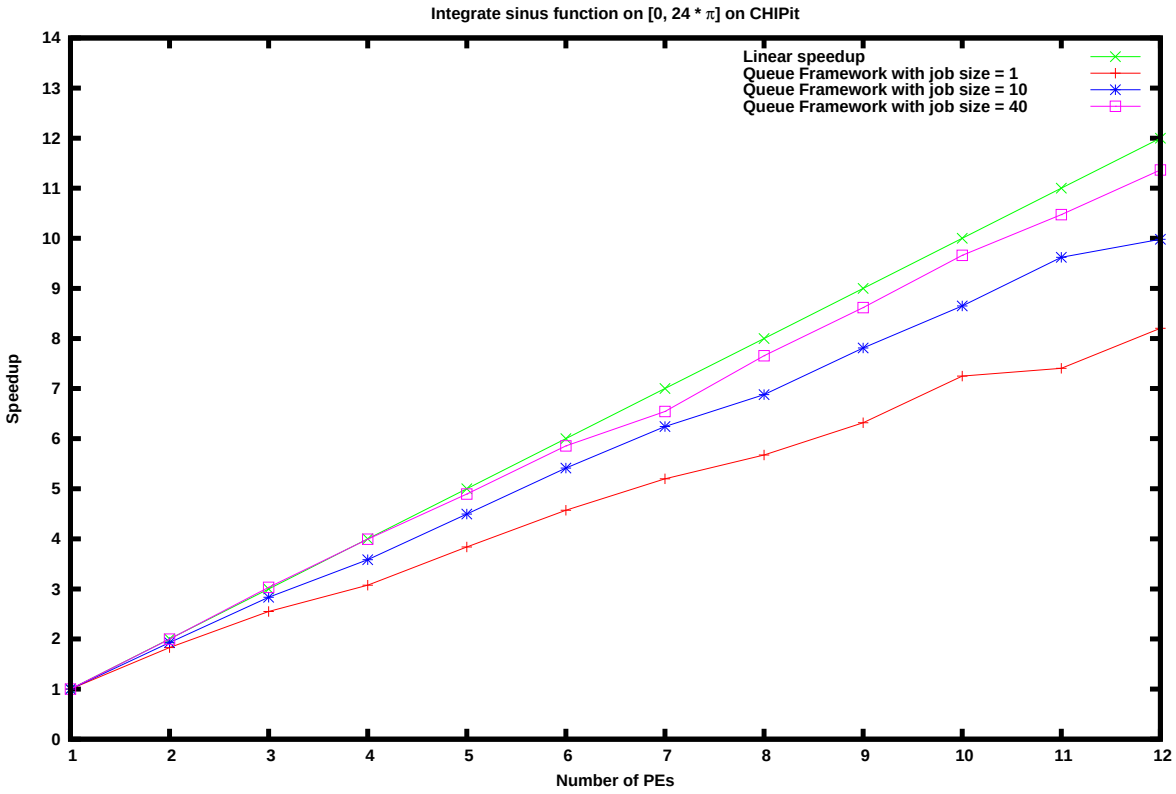


Abbildung 4.4.: Skalierung des Job Queue Frameworks bei der Erhöhung der Anzahl an reservierten PEs und der Variation der Arbeitspaketgröße bei der Berechnung des Integrals der Sinus-Funktion. Bei steigender Größe der Arbeitspakete steigt der Speedup des Job Queue Frameworks und nähert sich dem optimalen linearen Speedup. Eine *Job size* von k bedeutet für die Berechnung eines Intervalls wurde k-mal Math.sin(x) aufgerufen.

schnittliche Ausführungszeit über 10 Ausführungen mit der Ausführungszeit mit 1 PE verglichen. Das heißt, die Laufzeit des Job Queue Frameworks wurde mit der Laufzeit des Job Queue Frameworks mit 1 PE verglichen und die Laufzeit von claim. infect () wurde mit der Laufzeit von claim. infect () mit 1 PE verglichen. Bei einer optimalen Skalierung wäre also die Ausführungszeit der Anwendung mit jeder Anzahl von PEs gleich. Dies kann jedoch nicht erreicht werden, da Overhead entsteht, wenn ein neues PE hinzugefügt wird, der durch das Starten einer neuen Aktivität und dem Allokieren von Speicher auf jedem Place entsteht. Zusätzlich müssen verwendete Objekte serialisiert werden und auf jeden verwendeten Place kopiert werden. Dies betrifft claim. infect () und das Job Queue Framework in gleichem Maße. Dazu kommen die in Abbildung 4.3 gezeigten Skalierungsbeschränkungen der Integrate-Anwendung. Da der entstehende Overhead des Job Queue Frameworks in gleichem Maße steigt wie der einer statischen Lastverteilung ist kein Flaschenhals des Job Queue Frameworks erkennbar.

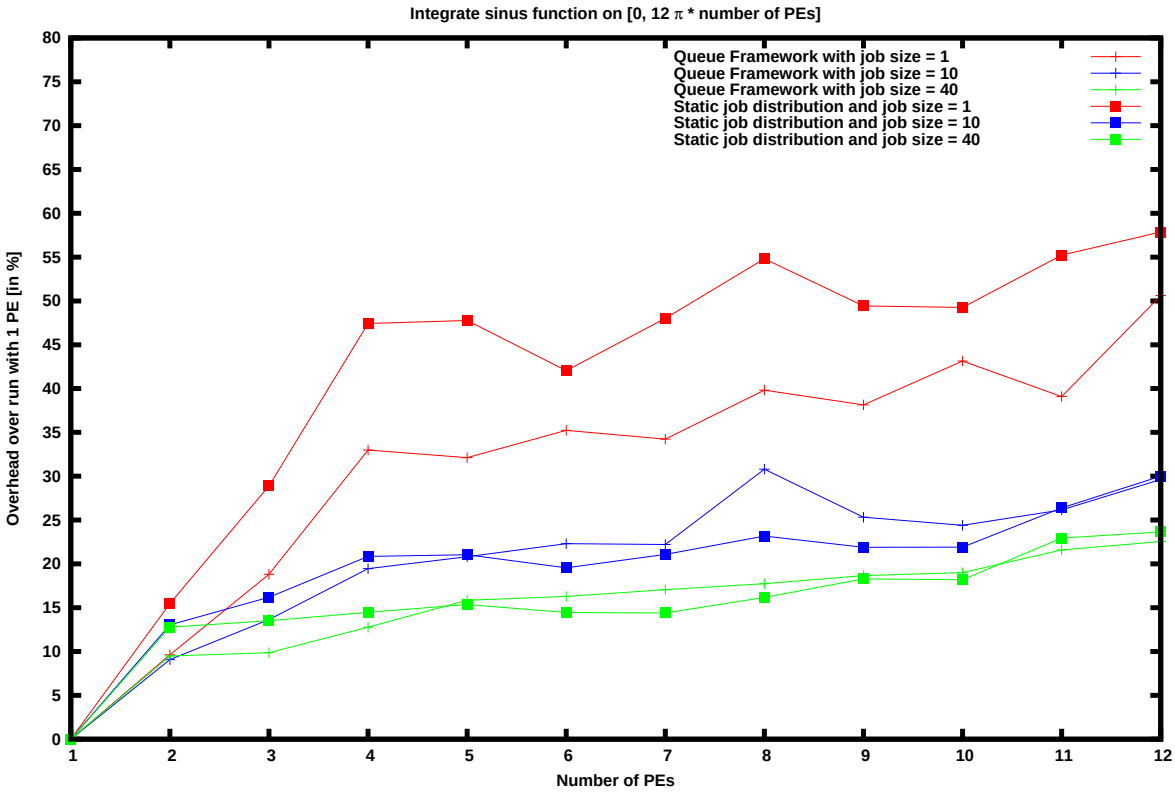


Abbildung 4.5.: Skalierung des Job Queue Frameworks bei der Erhöhung der Anzahl an reservierten PEs und der Variation der Arbeitspaketgröße bei der Berechnung des Integrals der Sinus-Funktion. Der Overhead des Job Queue Frameworks entspricht dem Overhead der auch bei der Benutzung einer statischen Lastverteilung mit Nutzung von claim. infect () auftritt, wenn ein neues PE hinzugefügt wird. Eine *Job size* von k bedeutet für die Berechnung eines Intervalls wurde k-mal Math.sin(x) aufgerufen.

4.4. Speedup

Dieser Abschnitt evaluiert den Speedup, den das Job Queue Framework beim UTS Benchmark erreicht hat. Dafür werden die Ausführungszeiten des Job Queue Frameworks, einer zentralen Queue und einer statischen Lastverteilung miteinander verglichen. Interessant wird der Speedup einer dynamischen Lastverteilung bei Anwendungen, bei denen die mögliche Bearbeitungsdauer eines Arbeitspaketes nicht berechenbar ist, sodass eine effektive statische Lastverteilung nicht möglich ist. In diese Kategorie von Anwendungen fallen unter anderem Such- und Optimierungsprobleme. Die Evaluation wurde mit dem UTS Benchmark durchgeführt, der diese Anwendungen simulieren kann.

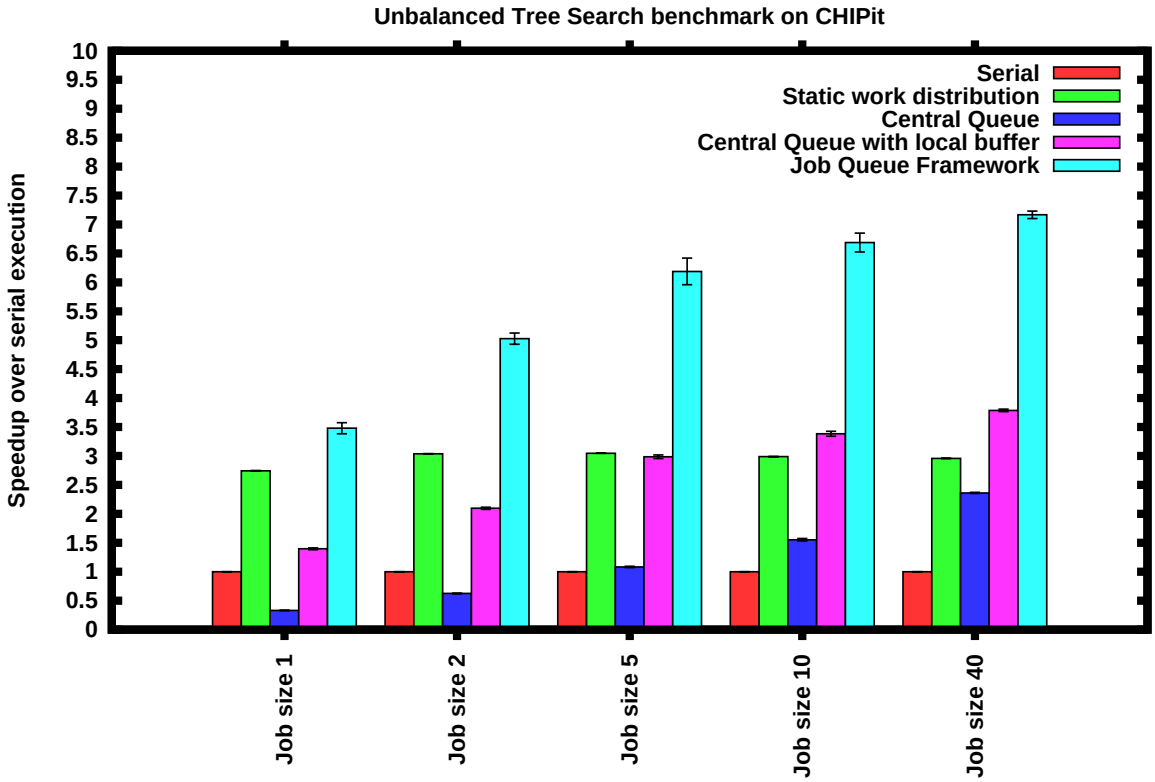


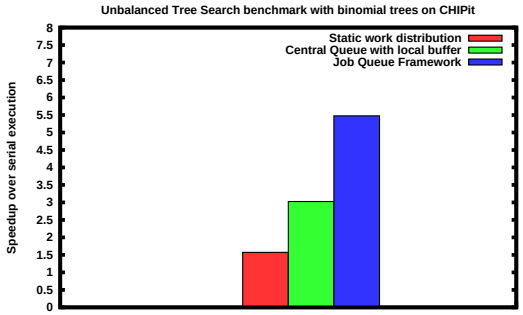
Abbildung 4.6.: Berechnung eines Binomialbaums mit 12 Kindern am Wurzelknoten und einer Wahrscheinlichkeit von $15/64$ für 6 Kindknoten und $49/64$ für 0 Kindknoten bei jedem weiteren Knoten. Der entstandene Baum wurde auf die Tiefe 17 beschränkt und hat 38481 Knoten. Der erreichte Speedup des Job Queue Frameworks gegenüber der optimierten zentralen Queue beträgt 2.49. Angegeben sind die Konfidenzintervalle zum Niveau 95%.

Die Abbildung 4.6 zeigt, dass das Job Queue Framework bei der Benutzung von 12 PEs einen Speedup von 3.48 gegenüber der seriellen Ausführung erreicht. Der Speedup gegenüber einer statischen Lastverteilung mit 12 PEs beträgt 1.27. Jeder Inkarnation des Infects wurde dabei ein Kindknoten des Wurzelknotens zugeteilt. Die Laufzeit wird in diesem Fall durch einen sehr großen Unterbaum bestimmt, der 34% aller Knoten enthält und nur auf einem PE berechnet wird. Die zentrale Queue ist ein Stack nach Scherer et al. [20]. Diese läuft auf Place 0, auf dem vier PEs reserviert sind. Die vier auf Place 0 laufenden PEs müssen bei der Benutzung der zentralen Queue immer neugestartet werden, damit Aktivitäten von anderen Places auf die Queue zugreifen können. Dies wurde nach fünf abgearbeiteten Arbeitspaketen getan. Nach wie vielen Arbeitspaketen die Aktivitäten neugestartet wurden, hatte allerdings kaum Einfluss auf die Gesamtlaufzeit, da diese durch at(queue.home){/∗return 1 job ∗/} und at(queue.home){/∗add all child node jobs∗/} bestimmt wird. Bei der zentralen Queue mit lokalem Puffer puffert jedes PE auf einem

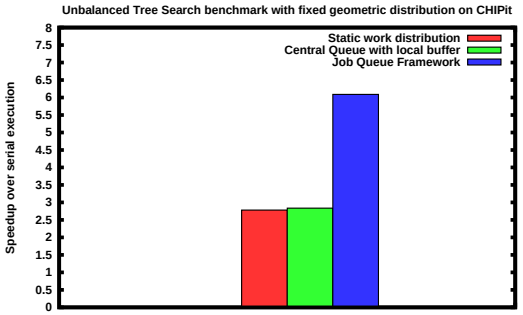
anderen Place als Place 0 bis zu dreißig Arbeitspakete. Erst wenn ein PE mehr als dreißig Arbeitspakete in seiner lokalen Queue liegen hat werden diese wieder in die zentrale Queue eingefügt. Das Job Queue Framework bearbeitet den Wurzelknoten zu Anfang und verteilt dann die daraus entstandenen Arbeitspakete auf die 3 Places. Der Speedup des Job Queue Frameworks gegenüber der zentralen Queue mit lokalem Puffer beträgt bei 12 PEs 2.49. Die zentrale Queue skaliert zudem schlecht mit steigender Anzahl an PEs. Dies liegt zum einen daran, dass die PEs auf dem Place mit der zentralen Queue bei einer ausreichend hohen Steigerung der Anzahl an PEs die eingehenden Aktivitäten nicht mehr schnell genug bearbeiten können. Zum anderen ist aktuell die Queue für i-lets auf einem PE auf eine Größe von 4 begrenzt. Bei mehr als sechzehn eingehenden Aktivitäten sind alle Queues auf einem Place voll und weitere gestartete Aktivitäten können nicht mehr durch Hardware auf PEs zur Ausführung angestoßen werden, sondern lösen einen Interrupt im OctoPOS Betriebssystem aus und werden in einer Software-Queue gespeichert.

Wird die Größe der Arbeitspakete um den Faktor acht vergrößert (UTS Job size 10), so erreichen das Job Queue Framework sowie die beiden zentralen Queues bessere Werte gegenüber der statischen Lastverteilung mit claim. infect (), da der Overhead der verteilten Operationen dann weniger ins Gewicht fällt. Das Job Queue Framework erreicht dann einen Speedup von 6.69 gegenüber der seriellen Ausführung und einen Speedup von 2.24 gegenüber der Verwendung der statischen Lastverteilung mit 12 PEs. Gegenüber der zentralen Queue mit lokalem Puffer konnte so ein Speedup von 1.98 erreicht werden. Je höher dabei die Arbeitspaketgröße wird, desto höher wird auch der erreichte Speedup des Job Queue Frameworks gegenüber der statischen Lastverteilung.

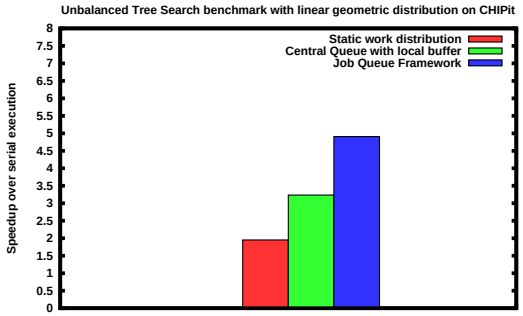
In Abbildung 4.7 ist erkennbar, dass das Job Queue Framework bei der Benutzung von 12 PEs für jeden Baumtyp einen höheren Speedup erreicht als eine statische Lastverteilung und eine zentrale Queue mit lokalem Puffer. Eine statische Lastverteilung kommt bei keinem Baumtyp über einen durchschnittlichen Speedup von 2.78 gegenüber einer seriellen Ausführung hinaus, da die entstehenden Bäume meist sehr unbalanciert sind. Zudem ist vorher nicht berechenbar, welche Unterbäume welche Größe haben werden, sodass eine dynamische Lastverteilung hier für eine effiziente Bearbeitung nötig ist. Eine zentrale Queue mit lokalem Puffer sowie das Job Queue Framework erreichen somit für jeden Baumtyp einen höheren Speedup. Gegenüber der statischen Lastverteilung erreicht das Job Queue Framework im Mittel über alle Bäume und Baumtypen einen Speedup von 3.07. Der Speedup, den das Job Queue Framework gegenüber der zentralen Queue mit lokalem Puffer erreichen kann, liegt gemittelt über alle Baumtypen und Bäume bei 1.76. Sowohl die zentrale Queue als auch das Job Queue Framework leiden dabei unter dem begrenzten Heap, den hohen Kosten für ein at sowie der begrenzten Skalierung des UTS Benchmarks auf einem Place.



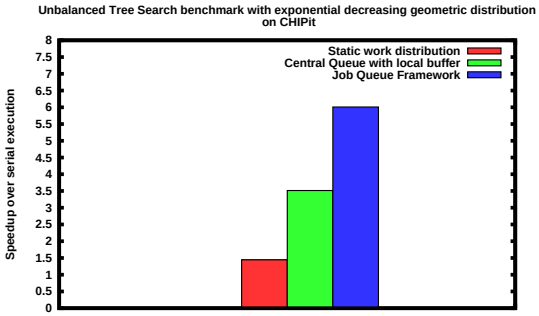
a Binomialbaum



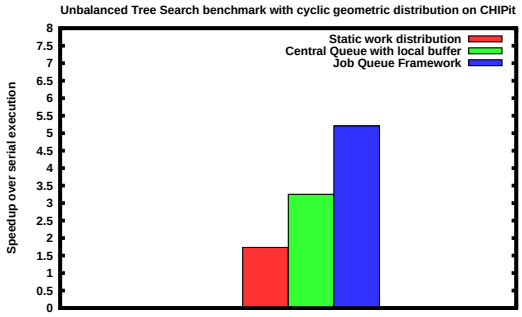
b Geometrischer Baum mit fester Verteilungsfunktion



c Geometrischer Baum mit linearer Verteilungsfunktion



d Geometrischer Baum mit exponentieller Verteilungsfunktion



e Geometrischer Baum mit zyklischer Verteilungsfunktion

Abbildung 4.7.: Für jeden Baumtyp erreicht das Job Queue Framework einen höheren durchschnittlichen Speedup als eine statische Lastverteilung und eine zentrale Queue mit lokalem Puffer. Pro Baumtyp wurden mehr als 15 verschiedene Bäume mit jeweils 20000-75000 Knoten erstellt. Die Arbeitspakete wurden in ihrer Größe erhöht (Job size 10), um den UTS Benchmark rechenintensiver zu machen und eine bessere Skalierung des Benchmarks zu erreichen.

4.5. Dynamische Ressourcenanpassung

Um die Genauigkeit der dynamischen Ressourcenanpassung zu evaluieren, wird auf den UTS Benchmark zurückgegriffen, indem ein Baum erstellt wird dessen Wurzelknoten acht Kindknoten hat. Jeder dieser Kindknoten hat wiederum bis zur Tiefe von 10000 immer nur einen Kindknoten. In der Tiefe 10000 jedoch haben drei der acht Knoten keinen weiteren Kindknoten mehr. Die maximale Tiefe des Baums ist begrenzt auf 30000. Der restliche Baum kann in diesem Fall also durch fünf PEs berechnet werden. Das Job Queue Framework hat alle 0.25 Sekunden überprüft, ob die Constraints geändert werden sollten. Wenn dies der Fall war, wurde ein Reinvade durchgeführt.

In Abbildung 4.8 ist erkennbar, dass das Job Queue Framework die Anzahl an benötigten PEs durch die erwartete Wartezeit pro PE gut einschätzen kann und die reservierten PEs durch ein Reinvade entsprechend anpasst. Das Ende zweier Teilbäume führt dazu, dass auf Place 1 zwei Aktivitäten keine Arbeitspakete mehr haben und auch keine mehr stehlen können. Dies führt zu einer zeitnahen Reduzierung der reservierten PEs seitens des Job Queue Frameworks. Kurze Zeit später hat auch eine Aktivität keine Arbeitspakete mehr auf Place 0. Die zwischenzeitlichen Schwankungen entstehen durch die Tatsache, dass das Job Queue Framework kein Work-Stealing von einem anderen Place zulässt, solange auf einem Place noch Arbeitspakete vorhanden sind. Und da ein Reinvade in diesem Fall immer zuerst PEs von Place 1 entfernt, wären für eine schnellstmögliche Berechnung 6 PEs nötig. Da allerdings 1 PE auf Place 0 dauerhaft keine Arbeitspakete zur Verfügung hat, versucht das Job Queue Framework die Anzahl an reservierten PEs immer wieder nach unten zu korrigieren. Die reservierten PEs steigen allerdings kurze Zeit später wieder, da auf Place 1 weitere Aktivitäten die Ausführung beschleunigen können. Es ist zu sehen, dass Anpassungen der Ressourcen sowohl nach oben als auch nach unten sehr nah an der optimalen Anzahl an reservierten Ressourcen liegen. Kurz vor Ende der Ausführung haben die Aktivitäten nach und nach keine Arbeitspakete mehr. Dies führt abermals zu einer Reduzierung der reservierten PEs, sodass am Ende der Ausführung nur noch 2-4PEs reserviert sind. Dabei ist erkennbar, dass das Job Queue Framework bereits kurze Zeit, nachdem einzelne Aktivitäten keine Arbeitspakete mehr haben, erkannt hat, dass Ressourcen freigegeben werden können. Wird die Zeit der periodischen Überprüfung der aktuellen Constraints niedrig gewählt, kann sehr schnell reagiert werden, um die reservierten Ressourcen in diesem Fall nach unten anzupassen. Wird die Zeit großzügiger gewählt, erfolgt diese Anpassung entsprechend langsamer. Allerdings ist der Overhead auf dem Place, auf dem der Claim erstellt wurde bei einer höheren Zeit geringer, da das Job Queue Framework seltener den Status auf allen Places für ein Reinvade überprüfen muss.

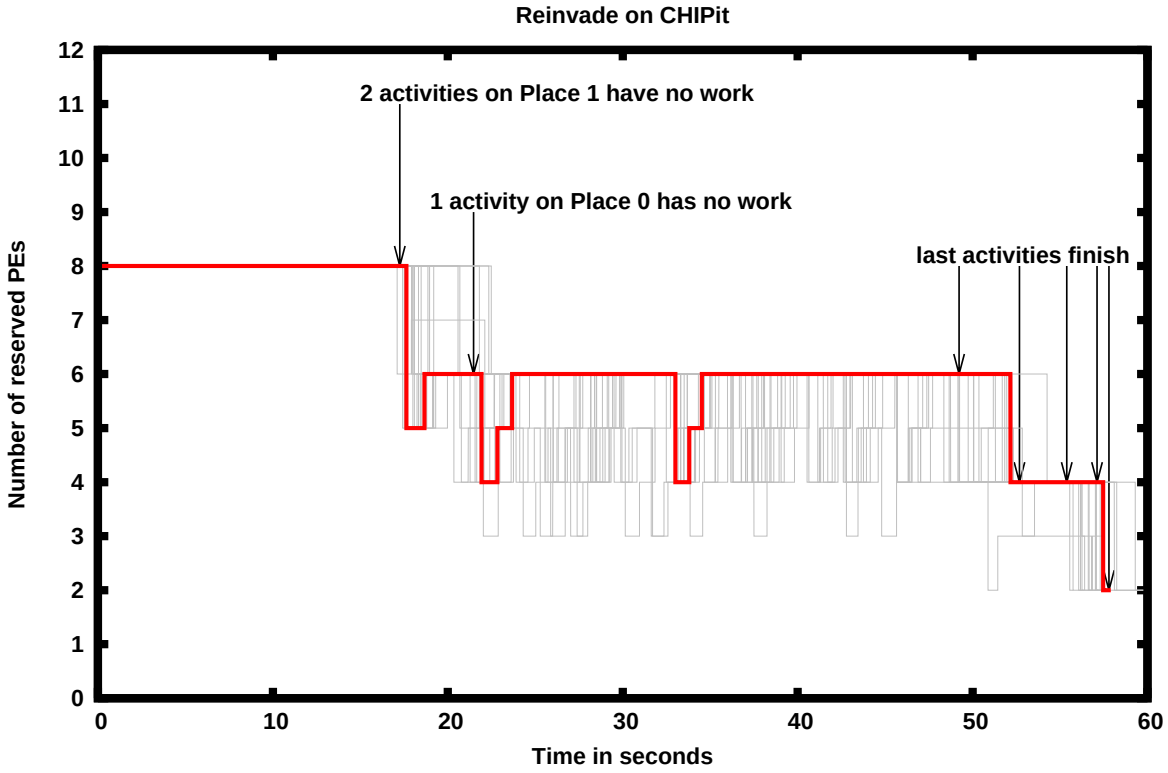


Abbildung 4.8.: Alle abgebildeten Linien zeigen die Anzahl reservierter PEs zu einem gegebenen Zeitpunkt. Es ist zu erkennen, dass das Job Queue Framework den Zeitpunkt, an dem auf Place 1 zwei Teilbäume nicht weiter wachsen, erkennt und Zahl der reservierten PEs reduziert. Nach dieser Reduzierung der PEs schwankt die Anzahl der reservierten PEs zwischen 3 und 6, da auf Place 0 eine Aktivität keine Arbeitspakete mehr besitzt, aber ein Reinvade immer wieder ein PE auf Place 1 entfernt. Nachdem die Aktivitäten am Ende auslaufen, werden die Ressourcen durch das Job Queue Framework zeitnah weiter reduziert. Die Zeit an dem einzelne Aktivitäten keine Arbeitspakete mehr hatten, ist für den rot hervorgehobenen Durchlauf eingezeichnet. Alle weiteren Durchläufe sind grau hervorgehoben.

5. Fazit und Ausblick

Das Ergebnis dieser Arbeit ist ein konfigurierbares und effizientes Job Queue Framework zur verteilten Bearbeitung von Arbeitspaketen. Die Konfigurierbarkeit ermöglicht es einem Anwender, das Job Queue Framework optimal auf seine Bedürfnisse hin anzupassen. Der Fokus bei der Implementierung des Job Queue Frameworks lag neben der Minimierung des Overheads durch die Lastverteilung auf einer effizienten Verteilung der vorhandenen Last, um einen hohen Speedup zu erreichen.

Für die Implementierung des Job Queue Frameworks wurden dynamische Lastverteilungsverfahren miteinander verglichen. Auch mussten die Eigenschaften von *Invasive Computing* und des zu Grunde liegenden Betriebssystems *OctoPOS* analysiert werden, um ein geeignetes dynamisches Lastverteilungsverfahren verwenden zu können. Eine weitere Herausforderung, die gelöst wurde, ist die Möglichkeit der dynamischen Ressourcenreservierung während der Laufzeit, die sich an die aktuelle Auslastung des Job Queue Frameworks anpasst. Da die Anforderungen des Job Queue Frameworks zudem stark von dem bearbeiteten Problem abhängen, ist das Job Queue Framework durch verschiedene Parameter konfigurierbar. Dies umfasst die dynamische Ressourcenreservierung und die Konfiguration der Lastverteilung.

Das Job Queue Framework wurde hinsichtlich der Aspekte Overhead, Skalierbarkeit, Speedup und dynamischer Ressourcenanpassung evaluiert. Dabei konnte zum einen gezeigt werden, dass der Overhead des Job Queue Frameworks gegenüber einer statischen Lastverteilung sinkt, je größer die Arbeitspakete sind. Bezüglich der Skalierbarkeit konnte gezeigt werden, dass das Job Queue Framework bei einer Steigerung der reservierten PEs fast linear skaliert, wenn die zu Grunde liegende Anwendung eine lineare Skalierung zulässt. Je größer dabei die Arbeitspakete sind, desto näher liegt die Skalierung des Job Queue Frameworks an der optimalen linearen Skalierung. Bei sehr kleinen Arbeitspaketen verhinderte der Overhead der verteilten Operationen sowie die Speicheranbindung der einzelnen Places eine optimale Skalierung, sodass bei einem theoretisch möglichen Speedup von 12 bei der Nutzung von 12 PEs nur ein Speedup von 8.2 gegenüber der Bearbeitung mit 1 PE erreicht werden konnte. Bei um den Faktor fünf größeren Arbeitspaketen lag der erreichte Speedup mit 11.36 nahe dem optimalen Wert. Des Weiteren hat die Evaluierung gezeigt, dass bei einer Steigerung der Problemgröße sowie der reservierten PEs das Job Queue Framework den gleichen zusätzlichen Overhead aufweist wie eine statische Lastverteilung mit `claim.infect()`. Dieser nimmt zudem mit steigender Arbeitspaketgröße ab, da dieser durch das Starten mehrerer Aktivitäten, dem Allokieren

von Speicher und dem Laufen des Garbage Collectors verursacht wird.

In einer Reihe von Anwendungen, in der die Größe einzelner Teilprobleme unberechenbar ist, wurde die Ausführungszeit des Job Queue Frameworks mit anderen Lastverteilungsverfahren verglichen. Dabei konnte bei der Nutzung von 12 PEs ein durchschnittlicher Speedup von 3.07 gegenüber statischer Lastverteilung erreicht werden, sowie ein Speedup von 1.76 gegenüber einer zentralen Queue mit lokalem Puffer. Das Job Queue Framework besitzt außerdem die Fähigkeit, die reservierten Ressourcen während der Laufzeit anzupassen. Es konnte gezeigt werden, dass das Job Queue Framework schnell auf Änderungen der benötigten Ressourcen reagieren kann. Das heißt es konnte festgestellt werden, wann Aktivitäten keine Arbeitspakete mehr zur Bearbeitung besitzen und wann ausreichend Arbeitspakete vorhanden sind, damit zusätzliche reservierte PEs zu einem Speedup führen. Die evaluierte Reaktion des Job Queue Framework lag dabei nahe am Optimum an reservierten Ressourcen. Zudem ist die dynamische Ressourcenanpassung durch den Anwender konfigurierbar.

Mit dieser Arbeit wurde die Grundlage gelegt, um weitere Lastverteilungsverfahren im Rahmen von *Invasive Computing* zu evaluieren. In zukünftigen Arbeiten ist einerseits eine Evaluierung von anderen Lastverteilungsstrategien möglich. Andererseits gibt es Möglichkeiten die in diesem Framework verwendete Metrik für die Schätzung der aktuellen Auslastung weiter zu verbessern und damit die Anpassung an die optimale Anzahl reservierter Ressourcen zu optimieren.

Das Job Queue Framework wurde als asynchron Malleable invasive Anwendung entworfen, sodass sich die Flexibilität des Job Queue Frameworks noch weiter erhöht, sobald der `Malleable` Constraints durch das Betriebssystem unterstützt wird. Zum einen ist dann seitens des Betriebssystems eine bessere Anpassung an die Gesamtauslastung des Systems möglich, da das Job Queue Framework dann zu jeder Zeit neue PEs erhalten kann sowie bereits reservierte PEs entfernt werden können. Zum anderen ist es seitens des Job Queue Frameworks möglich, die aktuellen Constraints noch deutlich häufiger anzupassen mit der Intention, dass das Betriebssystem dann gemäß der neuen Constraints die Ressourcen anpassen kann. Denn das Reinvade durch das Job Queue Framework ermöglicht lediglich, die Ressourcen zum Zeitpunkt des Reinvades anzupassen. Wenn an diesen Zeitpunkt eine andere Anwendung sämtliche restlichen Ressourcen belegt, könnten die Ressourcen des Job Queue Frameworks nie erhöht werden. Durch den `Malleable` Constraint kann das Betriebssystem die Ressourcen allerdings unmittelbar nach Ablauf der anderen Anwendung anpassen. So wäre auch die Einführung eines asynchronen Reinvades denkbar, dass lediglich die Constraint einer Anwendung anpasst und der Resize Handler der Anwendung aufgerufen wird, sobald das Betriebssystem zu einem späteren Zeitpunkt die Ressourcen der Anwendung aufgrund der neuen Constraints anpasst.

Literaturverzeichnis

[1] Aeroflex Gaisler: *LEON 3*, Mai 2015. http://www.gaisler.com/leonmain.html.

[2] Braun, M., S. Buchwald, M. Mohr und A. Zwinkau: *An X10 Compiler for Invasive Architectures*. Techn. Ber. 9, Karlsruhe Institute of Technology, 2012. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112.

[3] Buchwald, S., M. Mohr und A. Zwinkau: *Malleable Invasive Applications*. In: *Proceedings of the 8th Working Conference on Programming Languages (ATPS'15)*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2015. to appear.

[4] Burton, F. W. und M. R. Sleep: *Executing functional programs on a virtual tree of processors*. In: *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, S. 187–194. ACM, 1981.

[5] Chase, D. und Y. Lev: *Dynamic circular work-stealing deque*. In: *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, S. 21–28. ACM, 2005.

[6] Dijkstra, E. W. und C. S. Scholten: *Termination detection for diffusing computations*. Information Processing Letters, 11(1):1–4, 1980.

[7] Dinan, J., S. Krishnamoorthy, D. B. Larkins, J. Nieplocha und P. Sadayappan: *Scioto: A framework for global-view task parallelism*. In: *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, S. 586–593. IEEE, 2008.

[8] Dinan, J., D. B. Larkins, P. Sadayappan, S. Krishnamoorthy und J. Nieplocha: *Scalable work stealing*. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, S. 53. ACM, 2009.

[9] Dinan, J., S. Olivier, G. Sabin, J. Prins, P. Sadayappan und C. W. Tseng: *Dynamic load balancing of unbalanced computations using message passing*. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, S. 1–8. IEEE, 2007.

[10] Eager, D. L., E. D. Lazowska und J. Zahorjan: *A comparison of receiver-initiated and sender-initiated adaptive load sharing.* Performance evaluation, 6(1):53–68, 1986.

[11] Heisswolf, J.: *A Scalable and Adaptive Network on Chip for Many- Core Architectures.* Dissertation, Karlsruhe Institute of Technology (KIT), 2014.

[12] Hendler, D., I. Incze, N. Shavit und M. Tzafrir: *Flat combining and the synchronization-parallelism tradeoff.* In: *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, S. 355–364. ACM, 2010.

[13] Min, S. J., C. Iancu und K. Yelick: *Hierarchical work stealing on manycore clusters.* In: *5th Conf. on Partitioned Global Address Space Prog. Models*, 2011.

[14] Mohr, M., S. Buchwald, A. Zwinkau, C. Erhardt, B. Oechslein, J. Schedel und D. Lohmann: *Cutting Out the Middleman: OS-Level Support for X10 Activities.* In: *Proceedings of the fifth ACM SIGPLAN X10 Workshop*, X10 '15, S. 13–18, New York, NY, USA, 2015. ACM.

[15] Oechslein, B., J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann und W. Schröder-Preikschat: *OctoPOS: A parallel operating system for invasive computing.* In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, S. 9–14, 2011.

[16] Olivier, S., J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan und C. W. Tseng: *UTS: An unbalanced tree search benchmark.* In: *Languages and Compilers for Parallel Computing*, S. 235–250. Springer, 2007.

[17] Ravichandran, K., S. Lee und S. Pande: *Work stealing for multi-core hpc clusters.* In: *Euro-Par 2011 Parallel Processing*, S. 205–217. Springer, 2011.

[18] Saraswat, V., G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky und O. Tardieu: *The asynchronous partitioned global address space model.* In: *The First Workshop on Advances in Message Passing*, S. 1–8, 2010.

[19] Saraswat, V. A., P. Kambadur, S. Kodali, D. Grove und S. Krishnamoorthy: *Lifeline-based global load balancing.* In: *ACM SIGPLAN Notices*, Bd. 46, S. 201–212. ACM, 2011.

[20] Scherer III, W. N., D. Lea und M. L. Scott: *Scalable synchronous queues.* In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, S. 147–156. ACM, 2006.

[21] SPARC International Inc: *The SPARC architecture manual, version 8.*

[22] Synopsis Inc.: *CHIPit Platinum Edition and HAPS-600 series ASIC emulation and rapid prototyping system – hardware reference manual.*

Erklärung

Hiermit erkläre ich, Norman Christopher Böwing, dass ich die vorliegende Masterarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, 02.12.2015 Unterschrift

Danke

Hiermit bedanke ich mich herzlich bei dem Betreuer meiner Masterarbeit, Andreas Zwinkau, der mich bei der Bearbeitung tatkräftig unterstützt hat und jederzeit für Fragen erreichbar war.

Außerdem möchte ich mich bei Thomas Agrikola und Lucas Schulte im Walde bedanken, die diese Arbeit Korrektur gelesen haben. Vielen Dank!

A. Anhang

A.1. Parameter für die Evaluation

A.1.1. Speedup

Für die Berechnung des Speedups in Abbildung 4.6 wurde ein Baum mit folgenden Parametern genutzt:

- **type** = BINOMIAL
  seed = 17
  bf_0 = 12
  bf = 6
  nonLeafProb = $\frac{15}{64}$
  depth = 17

Für die Berechnung der Speedups in Abbildung 4.7 wurden folgende Parameter genutzt:

- **type** = BINOMIAL
  seeds = 500-580
  bf_0 = 12
  bf = 6
  nonLeafProb = $\frac{15}{64}$
  depth = 17

- **type** = GEOMETRIC
  distribution = FIXED
  seeds = 500-580
  bf_0 = 4
  depth = 7

- **type** = GEOMETRIC
  distribution = LINEAR

seeds = 500-580
bf_0 = 4
depth = 11

- **type** = GEOMETRIC
  distribution = EXPDEC
  seeds = 500-580
  bf_0 = 4
  depth = 13

- **type** = GEOMETRIC
  distribution = CYCLIC
  seeds = 500-580
  bf_0 = 4
  depth = 15