UNIKASSEL
VERSITÄT

Fachbereich 16 Elektrotechnik/Informatik
Studiengang Bachelor Informatik

Vergleich der HabaneroUPC++- und APGAS Bibliotheken

Bachelorarbeit

Jonas Scherbaum
Matrikelnummer: 30204135

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

Kassel, den 31.03.2016

Selbständigkeitserklärung

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

Kassel, den 31.03.2016

Jonas Scherbaum

Inhaltsverzeichnis

1 Einleitung 2

2 Grundlagen 4
- 2.1 Das PGAS Modell 4
- 2.2 APGAS – Eine Variante des PGAS-Modells 6

3 Vergleich 8
- 3.1 Allgemeines zu HabaneroUPC++ und dem APGAS-Framework 8
- 3.2 Inter-Place Parallelismus 11
- 3.3 Intra-Place Parallelismus 17

4 Experimente 26
- 4.1 K-Means 26
- 4.2 FFT2D 34
- 4.3 Testumgebung 45
- 4.4 Messergebnisse 45

5 Zusammenfassung und Fazit 49

Quellcodeverzeichnis 52

Abbildungsverzeichnis 53

Literaturverzeichnis 54

1 Einleitung

Das PGAS Modell (Partitioned Global Address Space) ist ein Programmiermodell, das speziell für die Bedürfnisse von verteilten Berechnungen entwickelt wurde. Es erlaubt den Speicher aller Knoten eines Clusters als einen gemeinsamen zusammenhängenden Speicher zu betrachten. Das Modell wurde in einer Reihe von Programmiersprachen implementiert, wie zum Beispiel in X10 [3], UPC [8] und Chapel [6]. Diese verwenden ihre eigenen Compiler und Sprachkonstrukte, um das PGAS Modell zu implementieren. Neue Programmiersprachen verbreiten sich jedoch sehr langsam, da Programmierer das Erlernen und die potentielle Inkompatibilität zu bereits existierenden Softwareprojekten in anderen Programmiersprachen als Hürde betrachten.

Die Bibliotheken HabaneroUPC++ [12] und das APGAS-Framework [17] umgehen diese Problematik, indem sie das PGAS Modell als Bibliothek für bereits weit verbreitete Programmiersprachen implementieren. Darüber hinaus implementieren beide Bibliotheken eine erweiterte Variante des PGAS Modells, das APGAS Modell (Asynchronous Partitioned Global Address Space). Dieses ermöglicht die asynchrone Ausführung von Tasks im PGAS Modell. Die Doppelbelegung des Begriffs APGAS für das Modell und das Framework wird im weiteren Verlauf dieser Arbeit differenziert, indem für das Framework der Begriff APGAS verwendet wird und für das Modell APGAS Modell.

In dieser Arbeit werden beide Bibliotheken miteinander verglichen, um ihre Gemeinsamkeiten und Unterschiede herauszustellen. Der weitere Verlauf dieser Arbeit gliedert sich wie folgt:

In Kapitel 2 wird das PGAS Modell, sowie dessen erweiterte Variante, das APGAS Modell, näher erläutert. Das Kapitel 3 vergleicht beide Bibliotheken miteinander und stellt die Gemeinsamkeiten und Unterschiede der Bibliotheken heraus. Anschließend werden in Kapitel 4 zwei Testprogramme beschrieben und miteinander verglichen, die jeweils mit beiden Bibliotheken implementiert wurden. Das letzte Kapitel 5 fasst die Ergebnisse des Vergleichs aus den Kapiteln 3 und 4 zusammen und beinhaltet ein Fazit.

2 Grundlagen

Dieses Kapitel fasst grundlegende Konzepte und Begriffe zusammen, die für das Verständnis beider Bibliotheken und deren Funktionsweisen essentiell sind. Dies umfasst das PGAS Modell, die asynchrone Variante APGAS, sowie die Begriffe Place, Async und Finish, siehe auch [16].

2.1 Das PGAS Modell

Das PGAS Programmier-Modell (Partitioned Global Address Space) stammt aus dem HPC (High Performance Computing) Bereich. Der Speicher aller Knoten in einem Cluster wird im PGAS Modell zu einem globalen gemeinsamen Speicher zusammengefasst. Auf diese Weise kann jeder Knoten auf jede Speicherzelle des globalen gemeinsamen Speichers zugreifen. Dieser wird im PGAS Modell in Places partitioniert. Eine solche Partition kann der gesamte Speicher eines Knotens sein oder nur ein Teil dessen. Es ist somit möglich mehrere Places pro Knoten zu definieren. Zugriffe auf den lokalen Speicher eines Knotens sind deutlich schneller als auf den eines entfernten, da diese über das Netzwerk durchgeführt werden müssen. Speicherzugriffe innerhalb eines Places sind immer lokal, der Zugriff auf den Speicher eines anderen Places hingegen ist potentiell ein entfernter Speicherzugriff. Befindet sich der Place, auf dessen Speicher zugegriffen wird, auf dem gleichen Knoten, ist dieser Zugriff ebenfalls lokal. Dieses Modell ist in Abb. 2.1 verdeutlicht, wobei diese lediglich einen Place pro Knoten annimmt.

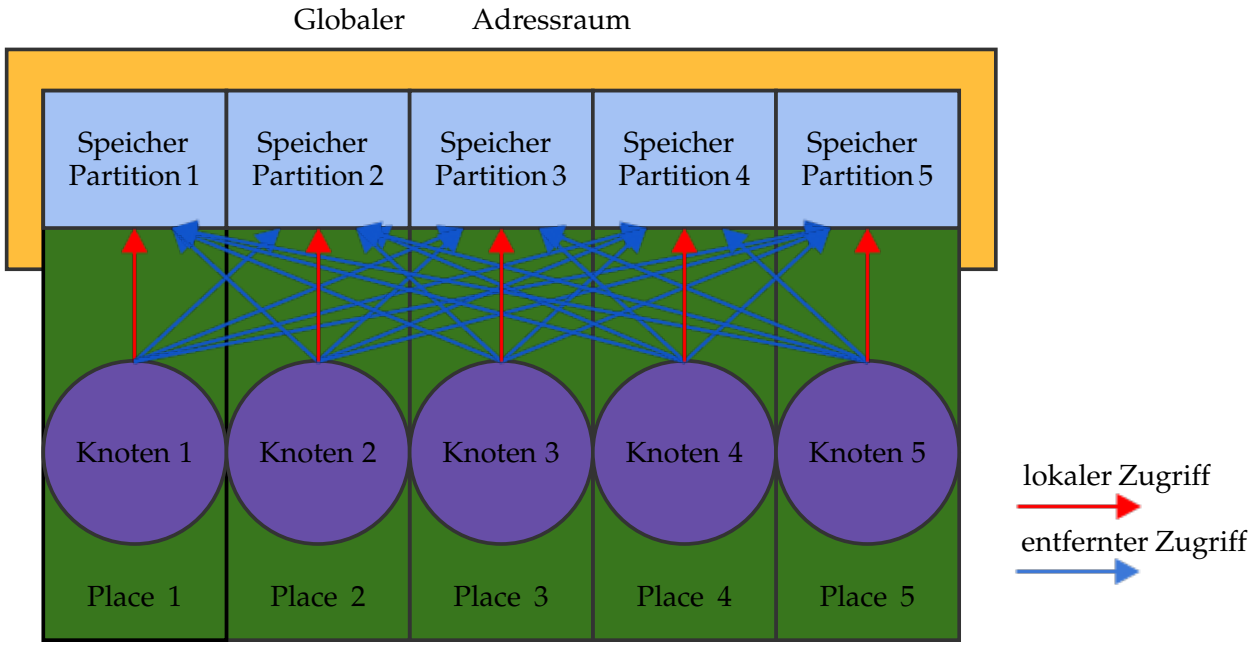

Abbildung 2.1: PGAS-Modell

Viele der PGAS Programmiersprachen arbeiten nach dem SPMD (Single Porgram Multiple Data) Ausführungsmodell, wie Co-Array Fortran [15] und Titanium [9]. Im SPMD Modell wird die Berechnung in einem einzelnen Programm festgelegt und dieses wird von jedem Prozess ausgeführt. Ein Prozess ist eine vom Betriebssystem verwaltete Instanz eines ausgeführten Programms, dem Speicher und CPU-Zeit zugeordnet wird. Jeder Prozess wird auf einem Knoten ausgeführt, wobei jedem beliebig viele Prozesse zugeteilt werden können. Die Berechnungen im PGAS Modell können in mehrere kleinere Teilberechnungen aufgeteilt und als Programmiermodell Threads bezeichnet werden. Die Laufzeitumgebung der PGAS Implementierung bildet diese auf Prozesse und Threads des Betriebssystems ab. Ein solcher Thread stellt einen leichtgewichtigen Prozess dar, der sich den Speicher und die CPU-Zeit eines Prozesses mit anderen Threads teilt. Mehrere Threads können sequentiell oder parallel zueinander ausgeführt werden und profitieren von modernen Multikernprozessoren, da sie auf diesen parallel zueinander ausgeführt werden können. Viele der aktuellen PGAS Implementierungen unterstützen sowohl multiple Prozesse, als auch Threads oder eine Kombination aus beidem. Das PGAS Modell legt weder die Abbildung, noch die Anzahl der Prozesse oder Threads fest.

Neben dieser Variante des PGAS Modells existiert noch eine weitere, die das erläuterte Modell um asynchrone Tasks erweitert. Dieses wird als APGAS (asynchronous PGAS) bezeichnet und wird im Folgenden näher erläutert.

2.2 APGAS – Eine Variante des PGAS-Modells

Das APGAS Modell ist aus der Entwicklung der PGAS Programmiersprache X10 entstanden, siehe [16]. Im APGAS Modell wird eine Berechnung als Aufgabe betrachtet, die in mehrere Teilaufgaben aufgeteilt wird, die asynchron zueinander abgearbeitet werden. Diese werden als Tasks bezeichnet und zur Abarbeitung einem Place übergeben. Es ist ebenfalls möglich, dass Tasks von einem Place auf einen anderen verschoben werden.

Die asynchronen Tasks des APGAS Modells werden durch zwei neue Konzepte eingeführt, Async und Finish. Neben diesen beiden neuen Konzepten wurde ebenfalls das Konzept der Places angepasst. Ein Place im APGAS Modell ist eine abstrakte Repräsentation eines Teils des globalen Adressraums, zusammen mit einem oder mehreren Worker Threads. Ein Worker Thread ist ein Thread, auf dem die asynchronen Tasks zur Ausführung abgebildet werden können. Das Konzept des Async wird dazu verwendet, um asynchrone Tasks zu definieren. Im APGAS Modell legt das Async nicht fest, welcher Place den asynchronen Task abarbeitet. Die meisten APGAS Implementierungen jedoch bieten Async Konstrukte an, um einen bestimmten Place festzulegen, der den asynchronen Task ausführt. Ein asynchroner Task kann selbst weitere asynchrone Tasks definieren. Diese neuen asynchronen Tasks werden als transitive Tasks bezeichnet und spannen so einen Task-Graphen auf. Der asynchrone Task, der weitere asynchrone Tasks definiert, repräsentiert dabei den Elternknoten im Graphen. Die neuen asynchronen Tasks werden als Kindknoten an den Elternknoten angehängt. Das neue Konzept des Finish wird dazu verwendet, um die asynchronen Tasks untereinander zu synchronisieren. Dies wird ermöglicht, indem eine Menge von asynchronen Tasks auf Terminierung überprüft und der aktuelle Programmablauf solange unterbrochen wird, bis alle asynchronen Tasks der Menge abgearbeitet wurden. Diese Überprüfung kann ebenfalls die transitiv gestarteten asynchronen Tasks einschließen, wie in X10. Es existieren jedoch auch Implementierungen des APGAS Modells, bei dem dies nicht zutrifft, wie in UPC++. Das Finish Konzept stellt den zentralen Synchronisierungsmechanismus im APGAS Modell dar, jedoch gibt es in den Implementierungen des APGAS Modells oftmals weitere Synchronisierungsmechanismen, wie zum Beispiel Lock-Variablen, Barrieren und atomare Variablen.

3 Vergleich

Die beiden Bibliotheken HabaneroUPC++ und APGAS sind zwei verschiedene Implementierungen des APGAS Modells. Sie verwenden beide Lambda Funktionen, um die asynchronen Tasks des APGAS Modells abzubilden. Die asynchronen Tasks können über die Places verteilt und parallel zueinander ausgeführt werden. Dies wird im weiteren Verlauf als Inter-Place Parallelismus bezeichnet. Dem gegenüber steht der Intra-Place Parallelismus, bei dem die asynchronen Tasks innerhalb eines Places parallel zueinander ausgeführt werden.

Der Vergleich der beiden Bibliotheken gliedert sich in drei Unterkapitel:

Das Unterkapitel 3.1 stellt beide Bibliotheken allgemein vor und erläutert die Implementierung der Lambda Funktionen in den jeweiligen Programmiersprachen C++ (HabaneroUPC++) und Java (APGAS-Framework). Der Hauptteil des Vergleichs teilt sich in die nächsten zwei Unterkapitel auf, die die Inter-Place (Unterkapitel 3.2) und Intra-Place (Unterkapitel 3.3) Parallelisierung voneinander trennen. In diesen werden jeweils die Konzepte und Konstrukte beider Bibliotheken miteinander verglichen.

3.1 Allgemeines zu HabaneroUPC++ und dem APGAS-Framework

HabaneroUPC++ ist in C++ im Standard 11 implementiert und basiert auf den beiden Bibliotheken UPC++ und HabaneroC++. UPC++ ist ebenfalls eine APGAS Implementierung, welche Inter-Place Parallelismus implementiert, siehe [22]. HabaneroUPC++

übernimmt das SPMD Ausführungsmodell von UPC++, welches im Kapitel 2.2 bereits erwähnt wurde. Diese besitzt jedoch keine Intra-Place Implementierungen, sodass asynchrone Tasks innerhalb von Places nicht möglich sind. Die Intra-Place Parallelisierung in HabaneroUPC++ wird durch die HabaneroC++ Work-Stealing Bibliothek implementiert, siehe [11], sodass asynchrone lokale Tasks in HabaneroUPC++ zur Verfügung stehen. Beide Bibliotheken werden in HabaneroUPC++ integriert, um Inter-Place mit Intra-Place Parallelismus zu verbinden. Dazu erweitert HabaneroUPC++ die Konzepte und Konstrukte beider Bibliotheken an einigen Stellen, sodass diese zusammen verwendet werden können. Die Konzepte und Konstrukte beider Bibliotheken werden in den folgenden Kapiteln 3.2, 3.3 näher erläutert. Das Verständnis der C++11 Lambda Funktionen ist Grundlage für das weitere Verständnis der HabaneroUPC++ Konzepte und Konstrukte, weshalb diese am Ende dieses Unterkapitels näher beschrieben werden.

APGAS ist in der Programmiersprache Java mit der Version 8 implementiert. Es stammt von den Entwicklern der Programmiersprache X10 und ist abgeleitet von der Resilient X10 Version 2.5.3, Details zur Resilient X10 Programmiersprache kann hier [7] gefunden werden. In APGAS wurden ausschließlich die Grundkonzepte und Konstrukte der X10 Programmiersprache, sowie die Fehlertoleranz-Eigenschaften übernommen. APGAS verwendet ein Task-paralleles Ausführungsmodell, bei dem das Programm in einer eigenen JVM (Java Virtual Machine) auf einem der Knoten gestartet wird. Dieser wird der erste Place zugeordnet, welcher im weiteren als Master bezeichnet wird. Anschließend startet die Laufzeitumgebung von APGAS alle weiteren Places, jeweils in einer eigenen JVM auf den anderen Knoten. Sind mehr Places als Knoten vorhanden, werden mehrere Places pro Knoten gestartet. Das Programm selbst wird als asynchroner Task auf dem Master Place ausgeführt. Die asynchronen Tasks werden in APGAS mit Lambda Funktionen abgebildet, welche im Folgenden näher erläutert werden.

APGAS führt für die Lambda Funktionen der asynchronen Tasks das funktionale Interface *SerializableJob* ein. Dieses implementiert das *Serializable* Interface von Java, damit die asynchronen Tasks serialisiert werden können. Dies ist nötig, um die asynchronen Tasks von einem Place zu einem anderen zu übertragen, was für gewöhnlich bedeutet, dass dies über das Netzwerk übertragen werden muss. Die Serialisierung umfasst die Lambda Funktion mitsamt aller Variablen, die der Lambda Funktion hinzugefügt wur-

den (siehe 3.1). Jeder Variablentyp, der der Lambda Funktion hinzugefügt wurde, muss ebenfalls serialisierbar sein. Aus diesem Grund müssen alle nicht primitiven Datentypen das Interface *Serializable* implementieren.

C++11 Lambda Funktionen, oder auch Closures genannt, sind Funktionen ohne Namen und können direkt an der Stelle, an der sie aufgerufen werden, definiert werden. Sie ähneln vom Prinzip her anonymen Klassen, wie sie z. B. von Java her bekannt sind. So wie anonyme Klassen können Lambda Funktionen wie Variablen behandelt, an Funktionen als Parameter übergeben, oder als Rückgabeparameter verwendet werden. Eine Lambda Funktion kann wie ein eigenständiges Objekt behandelt werden. Die Besonderheit von Lambda Funktionen ist, dass diese Zugriffe auf Variablen zulassen, die im gleichen Sichtbarkeitsbereich deklariert wurden, wie die Lambda Funktion selbst. Die verschiedenen Programmiersprachen, die Lambda Funktionen implementieren, tun dies jedoch unterschiedlich. In C++ haben Lambdas die folgende Syntax:

$$[]()->ret\_type\{...\}$$

Die eckigen Klammern *[]* werden als Capture Liste bezeichnet. In dieser Liste kann festgelegt werden, welche Variablen aus dem Sichtbarkeitsbereich in der Lambda Funktion zur Verfügung stehen sollen. Mit den Qualifizierern *&* und = kann angegeben werden, wie die Variablen zur Verfügung gestellt werden sollen. *&* wird verwendet, um eine Variable als Referenz zur Verfügung zu stellen. Dafür wird für die Lambda Funktion ein Pointer mit demselben Namen der Variablen erstellt. = wird verwendet, um den Inhalt einer Variablen zu kopieren und der Lambda Funktion als Kopie zur Verfügung zu stellen. Neben der Auflistung aller Variablen kann auch für alle Variablen mit *[&]* oder *[=]* festgelegt werden, dass alle als Referenz oder Kopie übernommen werden. Es ist auch möglich keine Variablen zur Verfügung zu stellen, indem die Klammern leer gelassen werden *[]*. Die einfachen Klammern *()* repräsentieren die Parameterliste der Funktion. Der optionale *ret_type* gibt den Rückgabewert der Funktion an. C++11 ist allerdings in der Lage diesen Typ automatisch zu erkennen. In den geschweiften Klammern *{}* wird der Funktionsrumpf definiert.

Intern werden Lambda Funktionen in C++ vom Compiler durch eine Klasse ersetzt. Diese Klasse überlädt den *() Operator*, sodass sie sich wie eine Funktion verhält. Der Konstruktor der Lambda Funktionsklasse wird verwendet, um die Variablen aus der Capture-Liste als Member-Variablen der Klasse zu übernehmen. Lambda Funktionen können in C++11 als Variablen vom Typ *std::function* verwendet werden.

Lambda Funktionen in Java 8 sind von anonymen Klassen abgeleitet. Dafür wurde das Interface Konstrukt in Java um funktionale Interfaces erweitert. Ein funktionales Interface ist definiert als ein Interface, dass nur eine einzige Methode besitzt. Eine Instanz eines solchen funktionalen Interfaces wird als Lambda Funktion bezeichnet. Lambda Funktionen in Java haben folgende Syntax:

$$(typevar, ...) -> \{...\}$$

Die einfachen Klammern () repräsentieren die Parameterliste der Funktion. Der Pfeil ist ein neues Token in Java 8 und kennzeichnet Lambda Funktionen. Die geschweiften Klammern bilden den Rumpf der Lambda Funktion. Dies kann eine einzelne Anweisung sein oder ein Anweisungsblock.

Lambda Funktionen in Java 8 werden vom Compiler in eine statische Methode der umgebenden Class-Datei übersetzt. Variablen auf die innerhalb der Lambda Funktion zugegriffen wird, die außerhalb der Lambda Funktion definiert wurden, werden der generierten statischen Methode als Funktionsparameter hinzugefügt.

3.2 Inter-Place Parallelismus

HabaneroUPC++ baut für die Implementierung des Inter-Place Parallelismus auf der UPC++ Bibliothek auf. UPC++ ist eine Portierung von UPC [8] in die Programmiersprache C++, bei der zusätzlich einige Erweiterungen vorgenommen wurden. Haba-

neroUPC++ erweitert ebenfalls an einigen Stellen UPC++. Die Implementierungen von UPC++, sowie die Erweiterungen in HabaneroUPC++, werden im Folgenden mit denen von APGAS verglichen.

Asynchrone entfernte Funktionsaufrufe

Asynchrone entfernte Funktionsaufrufe werden in UPC++ mit dem *async* Konstrukt umgesetzt. Dieses nimmt als Parameter einen Place, auf dem der asynchrone Task ausgeführt werden soll, und eine Lambda Funktion, die den Task beschreibt, entgegen. Syntaktisch ist dieses Konstrukt äquivalent zum *asyncAt* Konstrukt von APGAS. Das *async* Konstrukt in UPC++ kann jedoch keine Variablen der Capture-Liste an den entfernten Place übertragen. Diesen Nachteil gleicht HabaneroUPC++ mit seiner eigenen Implementierung des *asyncAt* Konstruktes aus, dass eine Kombination aus dem *async* Konstrukt und einer Kopierfunktion der UPC++ Bibliothek verwendet. Dies macht die *asyncAt* Konstrukte auch semantisch äquivalent.

UPC++ und APGAS verwenden ein ähnliches Synchronisierungsmodell für Abhängigkeiten zwischen asynchronen entfernten Tasks, ein *finish-async* Synchronisierungsmodell. Eine Abhängigkeit zwischen zwei asynchronen Tasks besteht, wenn ein asynchroner Task erst ausgeführt werden darf, nachdem ein anderer zuvor gestarteter asynchroner Task vollständig abgearbeitet wurde. UPC++ bietet zusätzlich ein Event basiertes Synchronisierungsmodell an, welches jedoch nicht mit in HabaneroUPC++ übernommen wurde. Das *finish-async* Synchronisierungsmodell von UPC++ unterstützt keine Terminierungserkennung von transitiven asynchronen Tasks. Diesen Nachteil gleicht HabaneroUPC++ mit einem weiteren neuen Konstrukt aus, dem *finish_spmd* Konstrukt. Der Name *finish_spmd* weißt auf das SPMD Ausführungsmodell hin, das von HabaneroUPC++ verwendet wird. Ein *finish_spmd* Konstrukt kann nur im SPMD Ausführungsmodell verwendet werden, weshalb diese nicht geschachtelt werden dürfen. Ein asynchroner Task wird nicht mehr im SPMD Ausführungsmodell ausgeführt, sondern asynchron dazu von nur einem Place. Dieses Konstrukt verbindet darüber hinaus die *async* Konstrukte von HabaneroC++ und UPC++, sodass in einem *finish_spmd* Konstrukt asynchrone Tasks von UPC++, als auch von HabaneroC++, auf Terminierung getestet werden können. Dies schließt transitive asynchrone Tasks beider Bibliotheken mit ein.

Innerhalb eines *finish_spmd* Konstruktes sind weitere *finish* Konstrukte erlaubt, nähere Details zu den *async* Konstrukten und dem *finish* Konstrukt folgen in Kapitel 3.3.

Das *async-finish* Synchronisierungsmodell synchronisiert lediglich die Ausführung der asynchronen Tasks, reicht jedoch nicht aus, um gemeinsame Speicherbereiche zu synchronisieren. Zugriffe auf gemeinsam genutzten Speicher aus asynchronen entfernten Tasks heraus, werden in HabaneroUPC++ mithilfe von *barriers* gesichert, die von UPC++ übernommen und geringfügig angepasst wurden. *barriers* definieren einen Abschnitt, in dem sich zu einem Zeitpunkt nur ein einziger asynchroner entfernter Task gleichzeitig aufhalten darf. APGAS erlaubt keine direkten Zugriffe auf Speicherbereiche des globalen Speichers, dies wird näher in Kapitel 3.2 erläutert. Aus diesem Grund entfällt die Notwendigkeit der Synchronisierung des globalen gemeinsamen Speichers für asynchrone entfernte Tasks in APGAS.

APGAS unterstützt die Fehlertolerante und elastische Ausführung von Programmen, indem es mögliche Fehler in Form von Exceptions abfängt. Der Programmierer ist so in der Lage, Fehler durch einen *try-catch* Block abzufangen und darauf zu reagieren. Der Programmierer hat die Möglichkeit den Fehler zu protokollieren und Gegenmaßnahmen einzuleiten. Eine solche Maßnahme kann zum Beispiel das Neustarten eines asynchronen Tasks auf einem anderen Place sein. Ein weiterer Fehlertoleranz-Mechanismus in APGAS bietet die Registrierung von Callback Funktionen. Die Callback Funktion wird dazu beim Starten eines Places in der Laufzeitumgebung registriert und wird ausgeführt, wenn der Place nicht mehr verfügbar ist. Innerhalb einer solchen Callback Funktion könnte versucht werden den Place neu zu starten. Die Elastizität von APGAS begründet sich in der Möglichkeit zur Laufzeit einzelne Places neu zu starten, weitere Places hinzuzufügen oder zu entfernen. Dies kann zum Beispiel durch den Benutzer über die Komandozeile geschehen. HabaneroUPC++ bietet keine Funktionalitäten für die fehlertolerante Ausführung von asynchronen Tasks. Die elastische Ausführung von Programmen in HabaneroUPC++ ist ebenfalls nicht möglich, da zum Zeitpunkt des Starts eines Programms bereits feststehen muss wie viele Places verwendet werden sollen. Außerdem können zur Laufzeit keine weiteren Places hinzugefügt oder entfernt werden.

PGAS Speichermodell Implementierungen

UPC++ implementiert das PGAS Speichermodell mit zwei verschiedenen Konzepten, verteilten Objekten und globalen Zeigern. Diese Konzepte werden von HabaneroUPC++ vollständig und ohne Änderungen übernommen. Die verteilten Objekte unterteilen sich nochmals in Skalare und Arrays. Beide Konzepte werden im Folgenden kurz erläutert.

Verteilte Objekte sind Datenobjekte, die im globalen Adressraum definiert und auf die von jedem Place aus zugegriffen werden kann. Ein verteilter Skalar ist eine einzelne Speicheradresse, auf dessen Wert direkt global zugegriffen werden kann. Der Wert des verteilten Skalars wird im Place 0 gespeichert. Ein verteiltes Array, ist ein Array, dass blockzyklisch über alle Places verteilt wird. Eine blockzyklische Verteilung teilt jedem Place einen Teil des Arrays zu, das der Blockgröße entspricht. Dies wird sooft wiederholt, bis die angegebene Gesamtgröße des Arrays erreicht ist. Die Blockgröße, sowie die Gesamtgröße des Arrays, werden bei der Deklaration des verteilten Arrays mit angegeben. Bei den verteilten Arrays handelt es sich lediglich um eindimensionale Arrays, für mehrdimensionale Arrays wird eine zusätzliche Bibliothek angeboten. Diese wird detailliert in [10] beschrieben. Zugriffe auf verteilte Objekte erfolgen, wie auf normale Variablen auch. Diese werden von der Laufzeitumgebung von UPC++ im Hintergrund über das Netzwerk ausgeführt.

Ein globaler Zeiger in UPC++ repräsentiert einen Zeiger auf eine Speicheradresse im globalen Adressraum, auf dessen Wert nur über Kopierfunktionen zugegriffen werden kann. Diese Kopierfunktionen werden im Folgenden noch näher erläutert. Ein solcher Zeiger kann von der Funktionsweise her mit einem normalen Zeiger in C++ verglichen werden. Die Operatoren des globalen Zeigers sind überladen, sodass auch Zeiger-Arithmetik mit globalen Zeigern möglich ist. Ein globaler Zeiger kapselt den lokalen C++ Zeiger auf die Speicheradresse, als auch den Place, auf dem die Speicheradresse liegt. Die Dereferenzierung eines globalen Zeigers ist nur aus dem Kontext des besitzenden Places möglich. Wird der globale Zeiger von einem anderen Place heraus dereferenziert, wird lediglich *Null* zurückgeliefert.

APGAS besitzt ein ähnliches Konzept für globale Speicheradressen, wie die globalen Zeiger von HabaneroUPC++, diese werden als globale Heap-Referenzen bezeichnet. Diese kapseln den zugehörigen Place und die lokale Speicheradresse direkt, anstatt einen Zeiger zu verwenden (Java besitzt keine Zeiger). Zugriffe auf die Speicheradresse werden direkt auf dem Place durchgeführt, auf dem sich die Speicheradresse befindet. Dazu wird jeder Zugriff in einem entfernten asynchronen Task gekapselt und dieser, mittels eines *asyncAt* Konstruktes, auf dem zugehörigen Place ausgeführt. Darüber hinaus können in APGAS globale Heap-Referenzen noch auf eine weitere Weise verwendet

werden. Dazu wird der globalen Heap-Referenz bei ihrer Initialisierung ein asynchroner Task und eine Liste an Places übergeben. Auf jedem dieser Places wird der asynchrone Task ausgeführt, der eine lokale Variable erstellt, die der globalen Heap-Referenz zugewiesen wird. Auf diese Weise wird für jeden Place eine identische Variable erstellt, die über die gleiche Heap-Referenz angesprochen werden kann.

Verteilte Objekte, wie in HabaneroUPC++ existieren in APGAS nicht, jedoch kann mithilfe der globalen Heap-Referenzen ein ähnliches Verhalten nachgeahmt werden. Eine globale Heap-Referenz kann verwendet werden, um einen verteilten Skalar abzubilden. Verteilte Arrays können durch die zweite Variante der globalen Heap-Referenzen ersetzt werden, indem im übergebenen asynchronen Task eine Verteilung über die Places vorgenommen wird.

Zusätzlich zu den globalen Zeigern besitzt UPC++ globale Reservierungs- und Freigabefunktionen, um Speicher auf entfernten Places für die globalen Zeiger zu reservieren. Diese übernimmt HabaneroUPC++ ebenfalls ohne Änderungen. Die Funktionen ähneln den bekannten Funktionen *malloc* und *free* der Programmiersprache C. Die Reservierungsfunktion besitzt jedoch einen zusätzlichen Parameter, mit dem der Place angegeben wird, auf dem der Speicher reserviert werden soll. Beide Funktionen können von jedem Place aus zu jedem Zeitpunkt aufgerufen werden, um Speicher auf einem beliebigen Place zu reservieren oder freizugeben.

Abschließend übernimmt HabaneroUPC++ ebenfalls noch die Kopierfunktionen von UPC++. Eine Kopierfunktion kann dazu genutzt werden Daten von einem Place zu einem anderen zu übertragen, indem der Kopierfunktion jeweils ein globaler Zeiger beider Places und die Menge der zu kopierenden Daten übergeben wird. Anschließend kopiert die Kopierfunktion die angegebene Menge an Daten aus der ersten Speicheradresse in die zweite Speicheradresse. Die Kopierfunktionen unterteilen sich in zwei Varianten, eine synchrone Variante und eine asynchrone Variante. In der synchronen Variante blockiert die aktuelle Ausführung des Programms, bis der Kopiervorgang abgeschlossen ist. Die asynchrone Variante blockiert nicht und führt den Kopiervorgang im Hintergrund aus. Asynchrone Kopiervorgänge können jedoch ebenfalls synchronisiert

werden, indem eine *fence* Funktion verwendet wird, die auf alle gestarteten Kopierfunktionen wartet oder der asynchronen Kopierfunktion ein Event Objekt mit übergeben wird. Auf dieses Event Objekt kann anschließend gewartet werden, um auf den damit verbundenen Kopiervorgang zu warten.

APGAS besitzt weder Reservierungs und Freigabefunktionen (aufgrund der *Garbage-Collection* in Java unnötig), noch explizite Kopierfunktionen. Die Kopierfunktionen sind aufgrund des Zugriffsmodells auf globale Heap-Referenzen nicht notwendig, da Daten die von einem Place zu einem anderen Place kopiert werden sollen, durch ein *asyncAt* Konstrukt übertragen werden können.

Kollektive Funktionen

UPC++ unterstützt eine Reihe von kollektiven Funktionen, die von HabaneroUPC++ übernommen werden. Kollektive Funktionen bieten eine wesentlich effizientere Möglichkeit Daten zwischen mehreren Places gleichzeitig zu verteilen. Dies ist in Situationen hilfreich, in denen beispielsweise ein Datensatz von einem Place auf viele Places verteilt werden soll (*broadcast*) oder wenn mehrere Places ihre Teilergebnisse auf einem Place zusammenfassen (*reduce*). Das APGAS-Framework besitzt keine solche kollektiven Funktionen.

3.3 Intra-Place Parallelismus

HabaneroUPC++ verwendet für die Intra-Place Parallelisierung die HabaneroC++ Work-Stealing Bibliothek und übernimmt dessen Work-Stealing Implementierung. HabaneroC++ ist eine in C++ geschriebene Bibliotheksimplementierung der Programmiersprache HabaneroC. Eine Übersicht bietet [19]. HabaneroC ist eine Portierung der Programmiersprache HabaneroJava, siehe [4], welche wiederum von X10 Version 1.5 (Juni 2007) abgeleitet wurde. HabaneroC++ verwendet intern eine in C geschriebene Bibliothek, die HClib [20], welche eine eingeschränkte Portierung von HabaneroC in eine Bibliotheksversion ist. Die HClib beinhaltet die Work-Stealing Laufzeitumgebung von HabaneroC++, welche in zwei Varianten verfügbar ist. Variante eins greift auf eine

Implementierung von OCR (Open Community Runtime) zurück, siehe [5], die zweite Variante ist eine eigene Implementierung der Work-Stealing Laufzeitumgebung von den Entwicklern der HClib. Alle HabaneroC++ Funktionen und Konstrukte werden auf die von HClib abgebildet. Im Folgenden werden die Implementierungen für die Intra-Place Parallelisierung von HabaneroUPC++ und dem APGAS-Framework verglichen.

Work-Stealing

Das APGAS-Framework, als auch die HabaneroUPC++ Bibliothek, verwenden für die effiziente parallele Ausführung der asynchronen Tasks innerhalb eines Places, je einen Work-Stealing Algorithmus. Beide Algorithmen ähneln sich in ihrer Implementierung, weshalb im Folgenden die ähnlichen Aspekte der beiden Work-Stealing Algorithmen erläutert werden, siehe Abb. 3.1. Im Anschluss wird auf die Unterschiede beider Algorithmen eingegangen.

Beide Work-Stealing Algorithmen besitzen eine Menge von Worker-Threads, wovon jeder Worker-Thread eine Task-Queue, im Weiteren als Worker-Queue bezeichnet, besitzt. Zu Beginn holen (*Take*) sich die Worker-Threads jeweils einen Task aus einer gemeinsamen Task-Queue und beginnen diesen abzuarbeiten. Diese gemeinsame Task-Queue unterscheidet sich in beiden Algorithmen, weshalb auf dessen Details im nächsten Abschnitt eingegangen wird. Während der Abarbeitung eines Tasks kann es dazu kommen, dass der Task neue Tasks erstellt. Diese reiht der Worker-Thread in seine eigene Worker-Queue ein. Wurde ein Task von einem Worker-Thread abgearbeitet oder dessen Abarbeitung blockiert, zum Beispiel aufgrund einer Synchronisierung, beginnt der Worker-Thread den nächsten Task seiner Worker-Queue abzuarbeiten. Besitzt der Worker-Thread keine weiteren Tasks mehr in seiner Worker-Queue, versucht dieser Tasks von einem anderen Worker-Thread zu stehlen (*Steal*). Dazu greift der Worker-Thread direkt auf die Worker-Queue des anderen Worker-Threads zu. Kann ein Worker-Thread keine weiteren Tasks mehr stehlen, versucht der Worker-Thread einen Task aus der gemeinsamen Task-Queue zu holen (*Take*). Die Tasks in den Worker-Queues sind aus zuvor ausgeführten Tasks entstanden, welche solange blockiert werden, bis die daraus entstandenen Tasks abgearbeitet wurden. Durch das Stehlen der Tasks aus den Worker-Queues,

können diese schneller abgearbeitet werden, wodurch der startende Task wiederum früher fortgesetzt werden kann. Ist in der gemeinsamen Task-Queue ebenfalls kein Task mehr enthalten, wird der Worker-Thread schlafen gelegt. Beide Implementierungen verwenden Deques für die Worker-Queues und die gemeinsame Task-Queue. Diese bieten drei verschiedene Operationen. Erstens eine *Push* Operation zum Hinzufügen eines Tasks am Anfang der Deque, zweitens eine *Pop* Operation zum Herausnehmen eines Tasks vom Anfang der Deque und drittens eine *Take* Operation zum Herausnehmen eines Tasks am Ende der Deque. Die Worker-Threads verwenden die *Push* und *Pop* Operationen, um Tasks der eigenen Worker-Queue hinzuzufügen oder zu entnehmen. APGAS, als auch HabaneroUPC++, starten des Programm selbst als asynchronen Task und fügen diesen beim Start der gemeinsamen Queue hinzu. Entfernte asynchrone Tasks werden durch die *asyncAt* Konstrukte der gemeinsamen Queue des jeweiligen Places hinzugefügt. Stehlversuche und die Entnahme von Tasks aus der gemeinsamen Task-Queue werden mit der *Take* Operation durchgeführt. Auf diese Weise wird die Synchronisierung der Task-Queues minimiert, welches die Effizienz der Work-Stealing Algorithmen verbessert.

APGAS verwendet als Implementierung des Work-Stealing Algorithmus, das Fork/Join-Framework [13] von Java. Das Fork/Join-Framework von Java beinhaltet einen Work-Stealing Algorithmus, wie dem der oben beschrieben wurde. Abbildung 3.1 illustriert die *Push*, *Pop* und *Take* Operationen, die der Work-Stealing Algorithmus verwendet. Als *Fork* Operation wird das Erstellen eines neuen Tasks innerhalb eines ausgeführten Tasks bezeichnet und fügt den Task der Worker-Queue des Workers-Threads hinzu. Eine *Join* Operation bezeichnet hingegen das Blockieren eines Tasks, bis alle transitiv gestarteten Tasks abgearbeitet wurden. Das Fork/Join-Framework führt bei der Ausführung eines Tasks darüber Buch, welche Tasks gestartet wurden und überprüft bei der *Join* Operation, ob diese Tasks abgearbeitet wurden. Welcher Worker-Thread die Tasks abarbeitet ist dabei irrelevant, da das Fork/Join-Framework die Buchführung global und unabhängig von den Worker-Threads durchführt. APGAS bildet das *async* Konstrukt auf die *Fork* Operation ab. Das *finish* Konstrukt wird auf die *Join* Operation abgebildet.

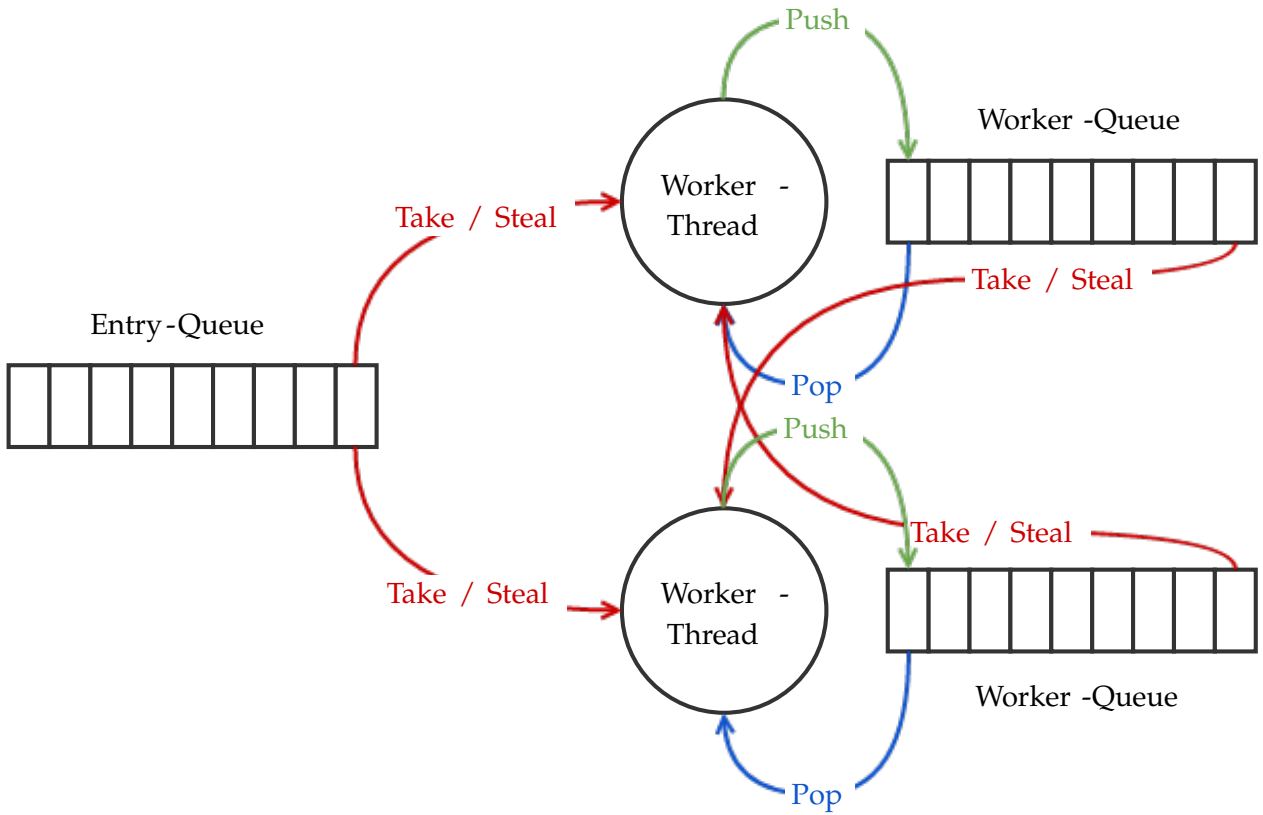

Abbildung 3.1: Work-Stealing Gemeinsamkeiten

Der Work-Stealing Algorithmus von HabaneroUPC++ (Abb. 3.2) verwendet an Stelle einer gemeinsamen Task-Queue zwei Task-Queues. Diese sind ebenfalls als Deque implementiert. Diese Task-Queues werden als In-Queue und Out-Queue bezeichnet. Beide Task-Queues werden einem speziellen Thread zugeordnet, dem *Communication-Thread*. Die UPC++ Laufzeitumgebung wird in einem eigenen Thread im Hintergrund ausgeführt, übernimmt sämtliche Netzwerk Transaktionen und speichert eingehende Nachrichten in einer Eingangswarteschlange. Der *Communication-Thread* führt die Hauptfunktion des Programms und alle UPC++ Funktionen aus. Während der Ausführung der Hauptfunktion des Programms werden die ersten Tasks erstellt und in die In-Queue und die Out-Queue eingefügt. Lokale Tasks werden in die In-Queue und entfernte Tasks werden in die Out-Queue eingefügt. Erreicht die Hauptfunktion ein *finish_spmd* Konstrukt, wird die Hauptfunktion des Programms blockiert und der *Communication-Thread* beginnt Tasks der Out-Deque (nur diese enthalten UPC++ Funktionen) abzuarbeiten. Nach jedem abgearbeiteten entfernten Task überprüft der *Communication-Thread*, ob neue Tasks von entfernten Places in der Eingangswarteschlange der UPC++ Laufzeitumgebung enthalten sind und fügt den ersten Task anschließend der In-Queue hinzu. Es wird lediglich ein Task der Eingangswarteschlange entnommen, damit der *Communication-Thread* nicht zu lange blockiert wird. Die Worker-Threads stehlen ausschließlich Tasks von anderen Worker-Threads oder aus der In-Queue. Stößt ein Worker-Thread während der Abarbeitung eines Tasks auf neue Tasks, fügt dieser diese in seine eigene Worker-Queue hinzu, wenn es sich um lokale Tasks handelt. Handelt es sich um entfernte Tasks, werden diese der Out-Queue hinzugefügt. Wurden alle Tasks abgearbeitet, wird die Hauptfunktion vom *Communication-Thread* fortgesetzt, bis ein weiteres *finish_spmd* Konstrukt erreicht oder das Programm beendet wird.

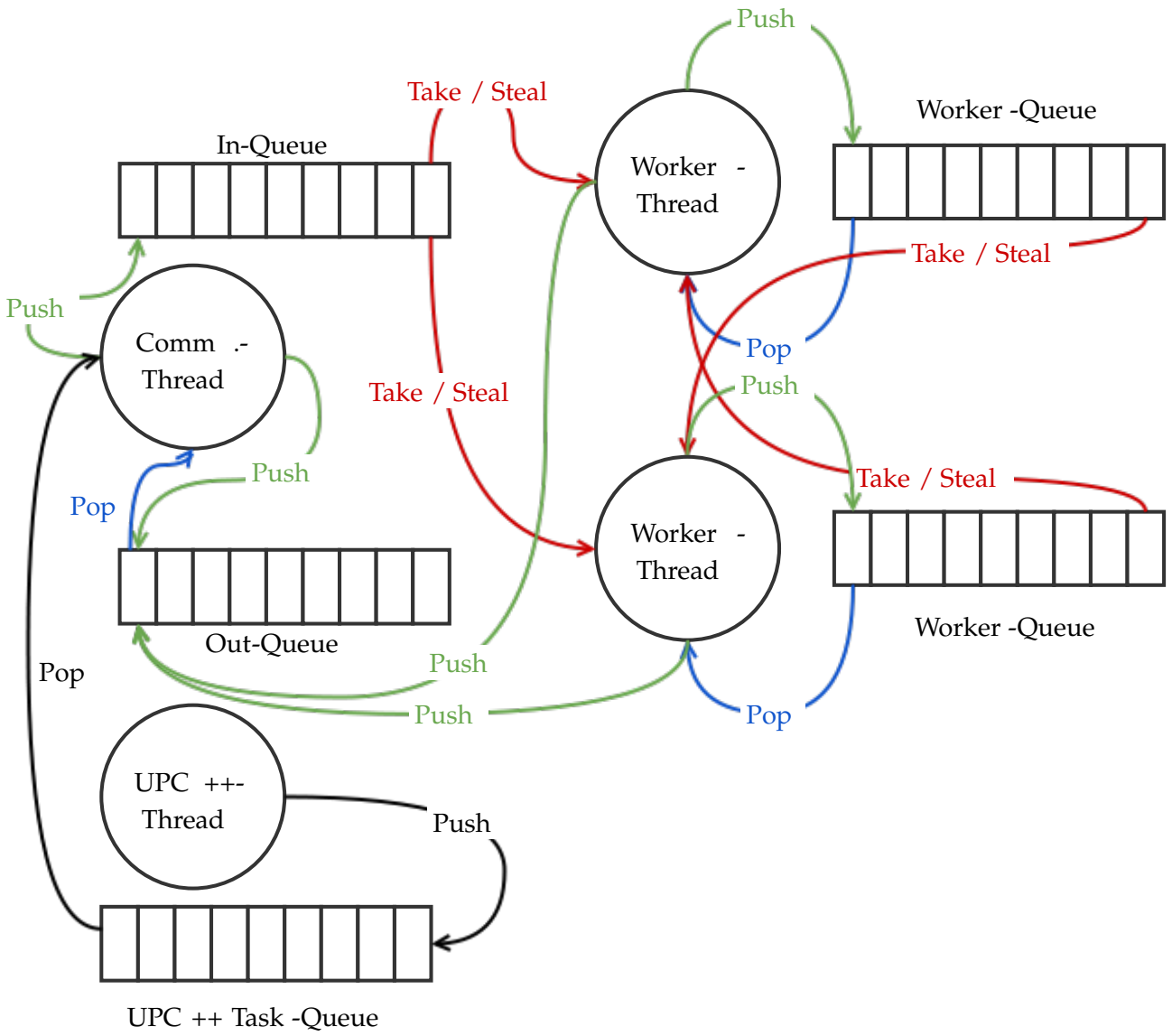

Abbildung 3.2: HabaneroUPC++ Work-Stealing

Asynchrone Tasks innerhalb von Places

APGAS, als auch HabaneroUPC++ verwenden ein *async* Konstrukt, um asynchrone Tasks zu erstellen und auf dem aktuellen Place auszuführen. Um diese lokalen asynchronen Tasks auf Terminierung zu testen, verwenden beide Bibliotheken ebenfalls ein ähnliches Konstrukt, das *finish* Konstrukt. APGAS verwendet dieses Konstrukt ebenfalls für entfernte asynchrone Tasks, wohingegen HabaneroUPC++ dafür ein weiteres Konstrukt verwendet, siehe Kapitel 3.2. Neben dem *async* Konstrukt verfügt HabaneroC++ noch über weitere Konstrukte für lokale asynchrone Tasks, welche im Anschluss näher erläutert werden. Diese Konstrukte sind in HabaneroUPC++ übernommen worden. APGAS besitzt neben dem *async* Konstrukt keine weiteren Konstrukte. Zugriffe auf gemeinsam genutzte Speicherbereiche innerhalb eines Places müssen synchronisiert werden. Diese Synchronisierung überlassen beide Bibliotheken der Standardimplementierungen der jeweiligen Programmiersprachen C++ und Java.

HabaneroC++ bietet ein weiteres Konstrukt, das *asyncAwait* Konstrukt. Dieses implementiert das Konzept von Data-driven Tasks (DDT), siehe [18]. Das Konzept stellt einen weiteren Synchronisierungsmechanismus, neben dem *finish* Konstrukt, zur Verfügung. Das *asyncAwait* Konstrukt nimmt zusätzlich zu dem auszuführenden asynchronen Task noch eine beliebige Anzahl an Data-driven Futures (DDF) entgegen. Diese werden als Ausführungsbedingungen für den asynchronen Task verwendet, sodass der asynchrone Task erst dann starten kann, wenn alle übergebenen DDFs erfüllt wurden. Ein DDF ist eine einfache Datenstruktur, die einen Statuswert (*empty* oder *full*) und eine Variable für Daten enthält. Andere asynchrone Tasks können eine *put* Operation auf DDF Objekten ausführen, um den Statuswert auf *full* zu setzen und der Variablen einen Wert zu übergeben (dies können auch komplexe Datenstrukturen sein). Auf diese Weise kann ein beliebiger Abhängigkeitsgraph zwischen asynchronen Tasks definiert werden. Wird der asynchrone Task ausgeführt, kann dieser auf den Wert der Variablen im DDF Objekt zugreifen.

Ein weiteres Konstrukt, ist das *forasync* Konstrukt, dass ebenfalls von HabaneroC++ nach HabaneroUPC++ übernommen wurde. Das *forasync* Konstrukt wird dafür verwendet, um eine For-Schleife mit mehreren asynchronen Tasks parallel auszuführen.

Dazu teilt das *forasync* Konstrukt die Iterationen der For-Schleife in Blöcke auf, die von jeweils einem asynchronen Task ausgeführt werden. Die Aufteilung der Iterationen wird mithilfe von zusätzlichen Parametern des *forasync* Konstrukts definiert. Die asynchronen Tasks des *forasync* Konstruktes müssen, wie alle anderen asynchronen Tasks, ebenfalls durch ein *finish* Konstrukt synchronisiert werden.

Hierarchische Place Bäume

HabaneroC++ übernimmt das Konzept der hierarchischen Place Bäume von HabaneroC, siehe [21]. Dieses Konzept erlaubt es den Speicher eines Computersystems in einem abstrakten hierarchischen Baum zu strukturieren und die Recheneinheiten des Computersystems bestimmten Teilen des Speichers zuzuordnen. Der Begriff Place wird in diesem Zusammenhang umdefiniert, sodass ein Place in einem hierarchischen Place Baum einem bestimmten Speichermodul entspricht. Aus diesem Grund muss ein Place in einem hierarchischen Place Baum abgegrenzt werden, zu einem Place im APGAS Modell. Für jeden Place des APGAS Modells wird in HabaneroUPC++ ein solcher hierarchischer Place Baum erstellt.

Ein solches Speichermodul kann der gesamte Hauptspeicher oder nur ein Teil dessen sein. Ein Speichermodul kann darüber hinaus auch der L1, L2 oder L3 Cache eines Prozessors, der Speicher einer Grafikkarte oder eines anderen Gerätes mit eigenem Speicher und Recheneinheit sein. Der Place besitzt zusätzlich als Parameter den Typ des Speichermoduls, wie z. B. DRAM, Cache oder Grafikspeicher. Diese Places können in einem Baum hierarchisch verknüpft werden, um so einen Baum zu konstruieren. Den Blattknoten dieses Baumes werden Recheneinheiten zugeordnet, welche als Worker bezeichnet werden. In HabaneroC++ und auch in HabaneroUPC++, wird der hierarchische Place Baum zu Beginn aus einer XML Datei geladen und steht anschließend der Laufzeitumgebung und dem Programmierer zur Verfügung. Asynchrone Tasks können den Places im hierarchischen Place Baum vom Programmierer zugeordnet werden. Dafür besitzt jeder Place des Baumes eine eigene Task-Queue, in die die asynchronen Tasks eingereiht werden. Die Laufzeitumgebung stellt für jeden Worker eines Blattknotens einen Worker-Thread bereit.

Jeder Worker-Thread arbeitet die Tasks der Task-Queue seines Blattknotens ab. Enthält der Blattknoten keine weiteren Tasks mehr, versucht der Worker-Thread Tasks des Elternknotens zu stehlen. Dies wird solange über alle weiteren Vorgängerknoten fortgesetzt, bis der Wurzelknoten des Baumes ebenfalls keine weiteren Tasks mehr in seiner Task-Queue besitzt. Neue asynchrone Tasks, die während der Abarbeitung eines Tasks entstehen, können vom Programmierer einem spezifischen Place im Baum übergeben werden. Auf diese Weise hat der Programmierer zu jedem Zeitpunkt die Kontrolle darüber, wo sich der asynchrone Task und dessen Daten im Speicher befinden und welcher Worker-Thread den asynchronen Task potentiell abarbeitet.

Der Vorteil dieses Konzepts ist es, dass die Lokalität der Daten der Tasks möglichst optimal ausgenutzt werden kann, da der Programmierer dessen Position im Speicher und die Abbildung des asynchronen Tasks auf bestimmte zugehörige Worker-Threads genau steuern kann. Ein weiterer Vorteil des Konzepts ist es, dass heterogene Computersysteme mit den hierarchischen Place Bäumen besser beschrieben und dessen Stärken ausgenutzt werden können. Ein Beispiel hierfür wäre ein Programm, dass eine Berechnung auf der CPU, als auch auf der GPU ausführen kann. Dieses Programm kann nach dem Start den Hierarchischen Place Baum zuerst analysieren und falls dieser einen Place mit Grafikspeicher enthält, die Berechnungen auf dem Place ausführen, der die Grafikkarte beschreibt.

HabaneroUPC++ übernimmt dieses Konzept von HabaneroC++, schränkt dessen Funktionalität jedoch in seinen eigenen Konstrukten ein. Die Einschränkung besteht darin, dass die asynchronen Tasks immer an den Wurzelknoten übergeben werden. Es können jedoch die Konstrukte von HabaneroC++ in HabaneroUPC++ Programmen problemlos verwendet werden, sodass die vollständige Funktionalität der hierarchischen Place Bäume auch in HabaneroUPC++ zur Verfügung steht. APGAS besitzt kein solches Konzept, um die Architektur eines Places des APGAS Modells näher zu beschreiben.

4 Experimente

Dieses Kapitel vergleicht die Bibliotheken HabaneroUPC++ und APGAS mithilfe von zwei Testprogrammen. Das erste Testprogramm führt einen K-Means Algorithmus auf einer vorab festgelegten Anzahl von Datenpunkten im mehrdimensionalen Raum aus. Dieses Testprogramm stammt aus den Beispielen der APGAS Bibliothek [2] und wurde um eine Intra-Place Parallelisierung erweitert, sowie mit HabaneroUPC++ neu implementiert. Das zweite Testprogramm implementiert eine zweidimensionale diskrete Fourier-Transformation (DFT) von komplexen Zahlen. Es wurde aus dem UPC++ Beispiel [1] FFT2D nach HabaneroUPC++ und APGAS portiert.

4.1 K-Means

K-Means ist einer der am häufigsten verwendeten Algorithmen zur Clusteranalyse. Ziel des K-Means Algorithmus ist es, die Datenpunkte in k Partitionen (auch als Cluster bezeichnet) aufzuteilen, sodass die Summe der quadratischen Abweichungen von den Clusterzentren minimal ist. Ein Clusterzentrum ist der Mittelwert aller Datenpunkte einer der k Partitionen. Mathematisch wird im K-Means Algorithmus die folgende Funktion minimiert:

$$d = \sum_{i=1}^{k} \sum_{x_j \in S_i} \| x_j - c_i \|^2 \tag{4.1.0.1}$$

x_j bezeichnet einen Datenpunkt der Partition S_i, c_i das Clusterzentrum der Partition S_i. Die verwendete Variante des K-Means Algorithmus entspricht der Variante nach LLOYD [14] und besteht aus den folgenden Schritten:

1 **Initialisierung:** Wähle k zufällige Mittelwerte (*Means*): $m_1, ..., m_k$ aus dem Datensatz.

2 **Zuordnung:** Jeder Datenpunkt wird derjenigen Partition zugeordnet, bei der die Distanz am geringsten ist: $S_i = \{x_j :\| x_j - c_i \|^2 \leq \| x_j - c_{i*} \|^2 frallei* = 1, ..., k\}$

3 **Aktualisieren:** Berechne die Clusterzentren neu: $c_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j$.

Schritt 2 und 3 werden sooft wiederholt, bis eine bestimmte Anzahl an Iterationen erreicht wurde.

Das Testprogramm nimmt fünf Startparameter entgegen. Parameter eins und zwei bestimmen die Anzahl der Datenpunkte und der Iterationen. Die Anzahl der Clusterzentren, die Dimension der Datenpunkte sowie ein Schwellwert können mit den letzten drei Startparametern angegebene werden. Es wird ein Schwellwert verwendet, um eine untere Schranke zu definieren, bei der die Menge der Datenpunkte nicht länger für die Parallelisierung aufgeteilt wird, siehe Quellcode 4.3. Im Folgenden wird Quell. als Abkürzung für Quellcode verwendet.

Im Inter-Place Bereich wird die Parallelisierung des K-Means Algorithmus, anhand der APGAS Implementierung (siehe Quell. 4.2), illustriert. Diese verwendet zusätzlich den Typ *ClusterState*, der in Quell. 4.1 definiert ist.

```
1  Class ClusterState implements Serializable {
2    float[][] clusters = new float[cluster][dimension];
3    int[] clusterCounts = new int[cluster];
4  }
```

Quellcode 4.1: KMeans APGAS ClusterState Type

Zu Beginn des Programms (in Quell. 4.2) wird die angegebene Anzahl an Datenpunkten mit Zufallszahlen initialisiert (*Z. 4-7*). Die Initialisierung der Datenpunkte findet verteilt auf allen Places statt und diese werden der globalen Referenz *globalPoints* übergeben. In *Z. 8* wird die globale Referenz *globalClusterState* initialisiert, dessen Wert auf dem aktuellen Place liegt und so global verfügbar gemacht wird. In *Z. 9* wird ein Array zum Speichern der aktuellen Clusterzentren angelegt. Die folgende Schleife (*Z. 13-35*) iteriert

über die zu Beginn angegebene Anzahl an Iterationen. Innerhalb dieser Schleife findet die Inter-Place Parallelisierung (Quell. 4.3) statt. Dazu wird ein *finish* aufgerufen (*Z. 14-29*) und jedem Place ein asynchroner Task zugewiesen (*Z. 15,16-27*). Jeder asynchroner Task erstellt eine lokale *ClusterState* Variable (*Z. 19*) und holt sich die Datenpunkte (*Z. 20*), die dem Place zugeordnet sind, aus der globalen Variable *globalPoints*. In den *Z. 21-23* wird mit einem *finish* die Intra-Place Parallelisierung auf Terminierung getestet. Abschließend wird der neu berechnete *ClusterState* (*Z. 24-26*) mit einem asynchronen Task auf den *globalClusterState* aufsummiert. Am Ende jeder Iteration werden die Cluster des *globalClusterState* mithilfe der *clusterCounts* normalisiert (*Z. 31-35*).

```
1  public static void main(String[] args) {
2    // load parameters numPoints, iterations, cluster, dimension, threshold ...
3
4    GlobalRef<float[][]> globalPoints = new GlobalRef<>(places(), () -> {
5      // generate random points ...
6      return points;
7    });
8    GlobalRef<ClusterState> globalClusterState = new GlobalRef<>(new ClusterState());
9    float[][] globalCurrentCluster = new float[cluster][dimension];
10
11   // initialize globalCurrentCluster with first few points of place 0 ...
12   for (int iter = 0; iter < iterations; iter++) {
13     finish(() -> {
14       for (Place place : places()) {
15         asyncAt(place, () -> {
16           // copy currentClusters from globalCurrentClusters ...
17
18           ClusterState localClusterState = new ClusterState();
19           float[][] points = globalPoints.get();
20           finish(() -> {
21             calculateClusterCenters(0, points.length, points, currentClusters,
                   localClusterState);
22           });
23           asyncAt(globalClusterState.home(), () -> {
24             // add values of localClusterState to globalClusterState using synchronised ...
25           });
26         });
27       }
28     });
29
30     for (int c=0;c<clusters;c++) {
31       for(int d=0;d<dimension;d++) {
32         globalClusterState.clusters[c][d] /= globalClusterState.clusterCounts[c];
33       }
34     }
35     // copy clusters from globalClusterState.clusters to globalCurrentClusters ...
36     // set all values of globalClusterState to 0 ...
37   }
38 }
```

Quellcode 4.2: KMeans APGAS Inter-Place Implementation

In Quell. 4.3 ist die Intra-Place Parallelisierung des K-Means Algorithmus dargestellt. Diese ist in der Methode *calculateClusterCenters()* implementiert. Die Methode teilt das Array der Datenpunkte, auf denen das Clustering durchgeführt wird, rekursiv in zwei gleich große Teilarrays auf, bis die Anzahl der Datenpunkte im Teilarray unterhalb des Schwellwerts liegt. Für jedes Teilarray wird die Methode rekursiv, innerhalb eines neuen asynchronen Tasks, aufgerufen.

Die *calculateClusterCenters()* Methode benötigt die Parameter *startIdx*, *endIdx*, *points*, *currentClusters* und *localClusterState*. *startIdx* und *endIdx* bestimmen die Partition des *points* Arrays, das von der Methode *calculateClusterCenters()* verarbeitet werden soll. Der Parameter *currentClusters* enthält die aktuellen globalen Clusterzentren und *localClusterState* das aktuelle Zwischenergebnis der neuen Clusterzentren. Zu Beginn (*Z. 3*) wird die aktuelle Anzahl an Punkten bestimmt, die in der übergebenen Partition des Punkte Arrays enthalten sind. Anschließend wird überprüft, ob diese Anzahl unterhalb des Schwellwerts liegt (*Z. 4*). Trifft dies zu, werden für die Partition des Punkte Arrays die Zwischenergebnisse im *localClusterState* Parameter aktualisiert. Die Aktualisierung wird mit einem *Synchronized-Block* synchronisiert. Ist die Anzahl der Punkte oberhalb des Schwellwerts (*Z. 6-14*), wird der Mittelpunkt des Arrays berechnet und die Partition in zwei gleich große Teile partitioniert. Für jede neue Partition wird anschließend ein asynchroner Task gestartet, der die *calculateClusterCenters()* Methode rekursiv mit der neuen Partition aufruft (*Z. 8-10*, *11-13*).

```
1   public static void calculateClusterCenters(int startIdx, int endIdx,
      float[][] points, float[][] currentClusters, ClusterState
      localClusterState) {
2
3     int nPoints = endIdx-startIdx;
4     if (nPoints < threshold) {
5       // calculate new clustercenters for the points from startIdx to
          endIdx ...
6     } else {
7       int mid = nPoints/2+startIdx;
8       async(() -> {
9         calculateClusterCenters(startIdx, mid, points, currentClusters,
            localClusterState);
10      });
11      async(() -> {
```

```
12            calculateClusterCenters(mid, endIdx, points, currentClusters, ↩
                  localClusterState);
13          });
14        }
15      }
```

Quellcode 4.3: KMeans APGAS Intra-Place Implementation

Die Implementierung des K-Means Algorithmus mit HabaneroUPC++ ist der mit APGAS sehr ähnlich, weshalb im folgenden lediglich die Unterschiede aufgeführt werden. Die globalen Referenzen von APGAS werden durch globale Zeiger von UPC++ ersetzt (Quell. 4.4), die Verteilung der Daten jedoch ist in beiden Implementierungen gleich. Jeder Place besitzt einen Teil der Punkte und eigene Variablen zum Speichern der Zwischenergebnisse für seine Punkte (*Z. 1-3*). In den *Z. 4-8* wird für diese Variablen auf jedem Place Speicher reserviert. Die Initialisierung der Punkte und der ersten Clusterzentren erfolgt ähnlich zu der APGAS Implementierung.

```
1 globalPoints = new upcxx::global_ptr<float> [upcxx::ranks()];
2 newCenters = new upcxx::global_ptr<float> [upcxx::ranks()];
3 newCounts = new upcxx::global_ptr<int> [upcxx::ranks()];
4 for (size_t i = 0; i < upcxx::ranks(); i++) {
5     globalPoints[i] = upcxx::allocate<float>(i, numberOfPointsPerPlace * ↩
          DIMENSION);
6     newCenters[i] = upcxx::allocate<float>(i, CLUSTERS*DIMENSION);
7     newCounts[i] = upcxx::allocate<int>(i, CLUSTERS);
8 }
```

Quellcode 4.4: KMeans HabaneroUPC++ Global Pointers

Einen entscheidenden Unterschied zwischen HabaneroUPC++ und APGAS existiert jedoch, denn HabaneroUPC++ arbeitet nach dem SPMD-Ausführungsmodell. Dieses führt die *main*-Funktion des Programms auf jedem Place aus, was zur Folge hat, dass auch jeder *asyncAt* Aufruf auf jedem Place ausgeführt wird. Dies führt dazu, dass jeder Place einen asynchronen Task erstellt und auf dem angegebenen Place startet. Soll in HabaneroUPC++ nur ein asynchroner Task erstellt und gestartet werden, muss der *asyncAt* Aufruf auf einen Place beschränkt werden. In Quell. 4.5 ist die Beschränkung auf den Place 0 illustriert (*Z. 4*). Alle weiteren *asyncAt* und *async* Aufrufe, die innerhalb eines asynchronen entfernten Task gestartet werden benötigen keine solche Beschränkung, da diese von dem ausgeführten asynchronen entfernten Task aufgerufen werden. Innerhalb

der Inter-Place Parallelisierung wird ebenfalls die Intra-Place Parallelisierung, mithilfe der *calculateClusterCenters()* Funktion, durchgeführt. Diese unterscheidet sich lediglich darin, dass in der HabaneroUPC++ Implementierung ein *mutex* verwendet wird, um die Zwischenergebnisse zu synchronisieren.

```
1  for (size_t iter = 0; iter < ITERATIONS; ++iter) {
2    hupcpp::finish_spmd([] () {
3      for (size_t i = 0; i < upcxx::ranks(); i++) {
4        if (upcxx::myrank() == 0) {
5          hupcpp::asyncAt(i, [] () {
6
7            // inter-place implementation ...
8                              });
9        }
10     }
11   });
```

Quellcode 4.5: KMeans HabaneroUPC++ Inter-Place Implementation

Der Austausch der Zwischenergebnisse erfolgt in HabaneroUPC++ im Anschluss an die Inter-Place Parallelisierung und ist in Quell. 4.6 illustriert. In *Z. 1* wird sichergestellt, dass der Austausch der Daten ausschließlich von Place 0 koordiniert wird. Die Zwischenergebnisse der anderen Places wird asynchron in die temporären Variablen *tempClusters* und *tempCounts* (*Z. 3,4*) kopiert (*Z. 6-9*). Anschließend wird in *Z. 10* mit einem *asnyc_copy_fence* Aufruf auf alle Kopiervorgänge gewartet und die neuen Clusterzentren werden berechnet.

```
1  if (upcxx::myrank() == 0) {
2
3    float *tempClusters = new float[upcxx::ranks()*clustersSize];
4    int * tempCounts = new int[upcxx::ranks()*CLUSTERS];
5
6    for (size_t i = 0; i < upcxx::ranks(); i++) {
7      upcxx::async_copy(newCenters[i], tempClusters+i*clustersSize,
           clustersSize);
8      upcxx::async_copy(newCounts[i], tempCounts+i*CLUSTERS, CLUSTERS);
9    }
10   upcxx::async_copy_fence();
11
12   // combine results from all places into variables of place 0 ...
13   // calculate new globalCurrentClusters ...
14
15   delete[](tempClusters);
16   delete[](tempCounts);
```

```
17  }
```

Quellcode 4.6: KMeans HabaneroUPC++ Results exchange

4.2 FFT2D

Für die Berechnung der DFT verwendet das Testprogramm die Bibliothek FFTW3. Die Parallelisierung der zweidimensionalen DFT wurde mithilfe der Bibliotheken APGAS und HabaneroUPC++ implementiert. Die DFT stammt aus dem Bereich der Fourier-Analysis und bildet ein zeitdiskretes endliches Signal, das periodisch fortgesetzt wird, auf ein diskretes periodisches Frequenzspektrum ab. Eine hohe Bedeutung besitzt die DFT in der digitalen Signalverarbeitung. Die mathematische Definition der DFT für komplexe Vektoren hat die folgende Form:

$$\hat{a}_{\mathrm{k}} = \sum_{j=0}^{N-1} e^{-2\pi i \frac{jk}{N}} * a_{\mathrm{j}} \qquad f\ddot{u}r \;\; k = 0, ..., N-1 \tag{4.2.0.2}$$

$\hat{a}_{\mathrm{k}}$ sind die Fourier-Koeffizienten der Fourier-Transformierten $\hat{a} = (\hat{a}_0, ..., \hat{a}_{\mathrm{N\text{-}1}}) \in \mathbb{C}^{\mathrm{N}}$. Der komplexe Vektor $a = (a_0, ..., a_{\mathrm{N\text{-}1}}) \in \mathbb{C}^{\mathrm{N}}$ beinhaltet die Koeffizienten des zeitdiskreten endlichen Signals.

Die Definition der DFT für komplexe Vektoren kann für mehrdimensionale Signale erweitert werden. Die eindimensionale DFT-Gleichung wird dazu je einmal auf alle Koordinatenrichtungen angewendet. Für den zweidimensionalen Fall im Testprogramm gilt somit:

$$\hat{a}_{\mathrm{k,l}} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{\mathrm{m,n}} * e^{-2\pi i * \frac{mk}{M}} e^{-2\pi i * \frac{nl}{N}} \qquad f\ddot{u}r \;\; k = 0, ..., M-1 \; und \; l = 0, ..., N-1 \tag{4.2.0.3}$$

Diese Gleichung kann so umgestellt werden, dass die Berechnung der DFT in der ersten Koordinatenrichtung M in folgende Variable substituiert werden kann:

$$w_{\mathrm{m,l}} = \sum_{n=0}^{N-1} a_{\mathrm{m,n}} * e^{-2\pi i * \frac{nl}{N}} \tag{4.2.0.4}$$

Die Variable $w_{m,l}$ aus Gleichung 4.2.0.4 kann anschließend in die Ausgangsgleichung 4.2.0.3 eingesetzt werden:

$$\hat{a}_{\mathrm{k,l}} = \sum_{m=0}^{M-1} w_{\mathrm{m,l}} e^{-2\pi i * \frac{mk}{M}} \tag{4.2.0.5}$$

Im Testprogramm wird die Variable $w_{m,l}$ der Gleichung 4.2.0.4 parallel berechnet, indem die Datenpunkte des Eingangssignals über die Koordinatenrichtung M auf die Places verteilt werden. Die berechneten $w_{m,l}$ Variablen werden anschließend drei mal transponiert und zwischen den Places ausgetauscht. Die drei Transponierungen sind notwendig, um eine Transponierung der gesamten Zwischenergebnismatrix (Zusammenfassung aller Zwischenergebnisse der Places in einer Matrix) abzubilden. Abschließend wird das Endergebnis der zweidimensionalen DFT mit den Variablen $w_{m,l}$ und der Gleichung 4.2.0.5 berechnet.

```
1 typedef struct {
2     double real;
3     double imag;
4 } complex_t;
```

Quellcode 4.7: FFT2D HabaneroUPC++ complex_t Type

Die Implementierung verwendet den Datentyp *complex_t* (Quell. 4.7), um komplexe Zahlen zu repräsentieren. Dieser Typ besitzt zwei Double Variablen für den Real- und Imaginär-Teil der komplexen Zahl. In Quell. 4.8 wird die *main* Funktion des Programms illustriert. Diese initialisiert zu Beginn die HabaneroUPC++ Laufzeitumgebung (*Z. 5*). Anschließend wird die Anzahl der Places abgerufen (*Z. 7*), sowie die *rowsPerPlace* bestimmt (*Z. 8*). Das Eingangssignal wird in einem Array von globalen Zeigern über alle Places verteilt (*input, Z. 9*). Für das Endergebnis wird ebenfalls ein Array von globalen Zeigern verwendet, dass das Endergebnis über alle Places verteilt (*output, Z. 10*). Die Zwischenergebnisse werden in globalen Zeigern gespeichert, die im globalen Zeiger des *allWeights* Arrays abgelegt werden. In *Z. 11* wird der Speicher für den *allWeights* Zeiger auf Place 0 reserviert. Der Speicher für die *input* und *output* Zeiger, sowie die

Zeiger der Zwischenergebnisse im *allWeights* Zeiger wird in den *Z. 13 - 17* auf jedem Place reserviert. Anschließend wird ein *finish_spmd* Konstrukt verwendet (Z. 19), um die asynchronen entfernten Tasks in *Z. 22 - 24* auf Terminierung zu testen. Es wird für jeden Place in einer for-Schleife (*Z. 20*) ein asynchroner entfernter Task gestartet, der die *fft* Funktion aufruft (*Z. 23, Quell. 4.9*). Die Besonderheit in der HabaneroUPC++ Implementierung ist hier, wie im vorhergehenden K-Means Testprogramm auch, die Überprüfung auf den Place 0 (*Z. 21*), damit nur Place 0 die asynchronen entfernten Tasks startet. Nachdem die asynchronen entfernten Tasks abgearbeitet wurden, wird der Speicher der globalen Zeiger auf jedem Place freigegeben (*Z. 29-33*). Abschließend werden die *input* und *output* Arrays freigegeben (*Z. 34,35*), bevor das Programm beendet wird (*Z. 37*).

```
1  int main(int argc, char **argv) {
2      // read program arguments
3      hupcpp::launch(&argc, &argv, [] () {
4
5          size_t places = upcxx::ranks();
6          size_t rowsPerPlace = rows / places;
7          input = new upcxx::global_ptr<complex_t> [places];
8          output = new upcxx::global_ptr<complex_t> [places];
9          allWeights = upcxx::allocate<upcxx::global_ptr<complex_t> >
               (upcxx::myrank(), places);
10
11         for (size_t i = 0; i < places; i++) {
12             input[i] = upcxx::allocate<complex_t>(i, rowsPerPlace*columns);
13             output[i] = upcxx::allocate<complex_t>(i, rowsPerPlace*columns);
14             allWeights[i] = upcxx::allocate<complex_t>(i,
                   rowsPerPlace*columns);
15         }
16
17         hupcpp::finish_spmd([places] () {
18             for (int i = 0; i < places; i++) {
19                 if (upcxx::myrank() == 0) {
20                     hupcpp::asyncAt(i, [] () {
21                         fft(rows, columns);
22                     });
23                 }
24             }
25         });
26
```

```
27          for (int i = 0; i < places; i++) {
28              upcxx::deallocate(input[i]);
29              upcxx::deallocate(output[i]);
30              upcxx::deallocate(allWeights[i].get());
31          }
32          delete [] input;
33          delete [] output;
34      });
35      return 0;
36  }
```

Quellcode 4.8: FFT2D HabaneroUPC++ Implementation, main

Die asynchron über alle Places ausgeführte *fft()* Funktion wird in Quell. 4.9 illustriert. In dieser werden zu Beginn die lokalen Zeiger der *input* und *output* Matrizen von den globalen Zeigern für den jeweiligen Place abgerufen (*Z. 4,5*). Als nächstes wird die Fourier-Transformation für die erste Dimension des Eingangssignals berechnet (*Z. 8*). Die in Place 0 gespeicherten *allWeights* Zeiger werden in *Z. 11* asynchron in die in *Z. 10* erstellte *localAllWeights* Zeiger kopiert. Für die nachfolgenden Schritte wird in *Z. 13* ein temporäres Array des Typs *complex_t* erstellt. In dieses Array wird das Ergebnis der ersten Transponierung gespeichert (*Z. 14*). Die zweite Transponierung benötigt zusätzlich die Menge der zu jedem Place zu übertragenen Zwischenergebnisse, diese wird in *Z. 16* bestimmt und in der Variablen *msgsPerPlace* gespeichert. Diese wird in den *Z. 17-19* dazu verwendet, um die zweite Transponierung für jeden Place durchzuführen. In *Z. 20* wird auf den asynchronen Kopiervorgang von *Z. 11* gewartet, damit in den *Z. 22-24* die Zwischenergebnisse asynchron zwischen den Places ausgetauscht werden können. Auf diese Kopiervorgänge wird in *Z. 25* gewartet. Abschließend werden die Zwischenergebnisse in *Z. 27* das dritte mal transponiert. Als letzten Schritt wird die zweite Dimension Fourier-transformiert.

```
1  void fft(size_t rows, size_t columns) {
2
3      // determine rowsPerPlace, columnsPerPlace, places, myID ...
4      complex_t *myInput = input[myID].raw_ptr();
5      complex_t *myOutput = output[myID].raw_ptr();
6
7      // generate random data for myInput ...
8      runFFT(rowsPerPlace, myInput, myOutput);
9
10     upcxx::global_ptr<complex_t> *localAllWeights = new ↩
           upcxx::global_ptr<complex_t> [places];
11     upcxx::async_copy(allWeights, ↩
           upcxx::global_ptr<upcxx::global_ptr<complex_t> > ↩
           (localAllWeights), places);
12
13     complex_t *temp = new complex_t[rowsPerPlace*columns];
14     transpose(myOutput, temp, rowsPerPlace, columns);
15
16     size_t msgsPerPlace = rowsPerPlace * columnsPerPlace;
17     for (size_t i = 0; i < places; i++) {
18         transpose(temp + i*msgsPerPlace, myOutput + i*msgsPerPlace, ↩
               columnsPerPlace, rowsPerPlace);
19     }
20     upcxx::async_copy_fence();
21
22     for (size_t i = 0; i < places; i++) {
23         upcxx::async_copy(myOutput+i*msgsPerPlace, localAllWeights[i] + ↩
               myID*msgsPerPlace, msgsPerPlace);
24     }
25     upcxx::async_copy_fence();
26
27     transpose(localAllWeights[myID].raw_ptr(), temp, columns, rowsPerPlace);
28     runFFT(rowsPerPlace, temp, myOutput);
29 }
```

Quellcode 4.9: FFT2D HabaneroUPC++ Implementation, fft

Die APGAS Implementierung der zweidimensionalen Fourier-Transformation verwendet ebenfalls die Bibliothek FFTW3, um die Fourier-Transformationen zu berechnen. Diese nimmt ausschließlich eindimensionale Arrays entgegen, weshalb die Hilfsklasse *Matrix* implementiert wurde (Quell. 4.10), die eine zweidimensionale komplexe Matrix

auf ein eindimensionales Array abbildet. Die wichtigste Methode der *Matrix* Klasse ist *transpose()*, welche die Matrix transponiert.

```
1  private static class Matrix implements Serializable {
2
3    private double[] innerArray;
4    private int nx;
5    private int ny;
6
7    void transpose() {
8      // transpose the innerArray ...
9    }
10
11   // additionally helpermethods ...
12 }
```

Quellcode 4.10: FFT2D APGAS Matrix Type

Die APGAS Implementierung (Quell. 4.11) verwendet globale Referenzen, um das Eingangssignal (*input*), das Endergebnis (*output*) und die Gewichte (*allWeights*), auf die Places zu verteilen (*Z. 7-15*). Jeder Place erhält eine *input* und *output* Variable, sowie Place 0 eine Liste mit globalen Referenzen zu den Gewichten aller Places. Die Implementierung führt nun vier Schritte durch, um die zweidimensionale Fourier-Transformation durchzuführen. Im ersten Schritt *Z. 17* wird die Liste der Gewichts-Referenzen asynchron initialisiert (Quell. 4.12). Anschließend wird in *Z. 18* die erste Dimension des Eingangssignals transformiert und das Ergebnis transponiert (Quell. 4.13). Der dritte Schritt in *Z. 19* tauscht die Gewichte aus der ersten Transformation zwischen den Places aus (Quell. 4.14). Zum Schluss wird in Schritt 4 *Z. 20* die Gewichtsmatrix des vorherigen Schrittes transponiert und die zweite Dimension der Fourier-Transformation wird ausgeführt (Quell. 4.15). Die vier Schritte werden im Folgenden näher erläutert.

```
1  public static void main(String[] args) {
2
3    // read in the commandline arguments rowCount, columnCount...
4
5    // partition the rows over all Places
6    int rowsPerPlace = rowCount / numPlaces;
7    GlobalRef<Matrix> input = new GlobalRef<>(places(), () -> {
8      // generate random points for Input...
9      return points;
10   });
11
12   GlobalRef<Matrix> output = new GlobalRef<>(places(), () -> new
         Matrix(rowsPerPlace, columnCount));
13
14   ArrayList<GlobalRef<Matrix>> weights = new ArrayList<>(numPlaces);
15   GlobalRef<ArrayList<GlobalRef<Matrix>>> allWeights = new
         GlobalRef<>(weights);
16
17   distributeWeights(columnCount, rowsPerPlace, allWeights);
18   calculateFirstFFTDimension(input, output, rowCount, columnCount);
19   exchangeWeights(output, allWeights);
20   transposeAndFinishFFT2D(allWeights, output, rowCount, columnCount);
21 }
```

Quellcode 4.11: FFT2D APGAS Main

Die Initialisierung der Gewichts-Referenzen aus Schritt 1 erfolgt in der *distributeWeights()* Methode (Quell. 4.12). Dazu wird jedem Place ein asynchroner Task übergeben, in dem lokal eine *Matrix* Variable erstellt (*Z. 8*) und einer globalen Referenz übergeben wird (*Z. 9*). Diese wird anschließend mit einem weiteren asynchronen Task vom Place 0 der Liste *allWeights* hinzugefügt (*Z. 10-15*).

```
1  public static void distributeWeights(
2    int columnCount, int rowsPerPlace,
3    GlobalRef<ArrayList<GlobalRef<Matrix>>> allWeights) {
4
5    finish(() -> {
6      for(Place place: places()) {
7        asyncAt(place, () -> {
8          Matrix localWeight = new Matrix(columnCount, rowsPerPlace);
9          GlobalRef<Matrix> localWeightRef = new GlobalRef<>(localWeight);
10         asyncAt(allWeights.home(), () -> {
11           ArrayList<GlobalRef<Matrix>> weightList = allWeights.get();
12           synchronized(weightList) {
13             weightList.add(place.id, localWeightRef);
14           }
15         });
16       });
17     }
18   });
19 }
```

Quellcode 4.12: FFT2D APGAS Weights Distribution

Schritt 2 führt die Methode *calculateFirstFFTDimension()* aus (Quell. 4.13). Diese startet für jeden Place einen asynchronen Task, der für das Eingangssignal (*input*) die Fourier-Transformation mit der FFTW3 Bibliothek durchführt (*Z. 6*) und das Ergebnis in *output* abspeichert. Anschließend wird das Ergebnis noch transponiert (*Z. 7*) und damit für den nächsten Schritt vorbereitet.

```
1  public static void calculateFirstFFTDimension(GlobalRef<Matrix> input,
2    GlobalRef<Matrix> output, int rowCount, int columnCount) {
3    finish(() -> {
4      for (Place place: places()) {
5        asyncAt(place, () -> {
6          fft(input.get(), output.get(), rowCount, columnCount);
7          output.get().transpose();
8        });
9      }
10   });
11 }
```

Quellcode 4.13: FFT2D APGAS First FFT

Der Austausch der Gewichte wird von der Methode *exchangeWeights()* durchgeführt (Quell. 4.14). Diese startet für jeden Place einen asynchronen Task, der das Ergebnis der ersten Fourier-Transformation aus der *output* Variable abruft (*Z. 9*) und anschließend in eine Partition für jeden Place aufteilt (*Z. 12-15*). Jede dieser Partitionen wird ein weiteres mal transponiert und in die *tmp* Variable kopiert. In *Z. 17* wird die Liste der Gewichtsreferenzen abgerufen, über die in einer For-Schleife iteriert und für jede globale Referenz ein asynchroner Task erstellt wird, um den jeweiligen Teil der *tmp* Variablen der Matrix der globalen Referenz hinzuzufügen (*Z. 18-25*).

```
1  private static void exchangeWeights(GlobalRef<Matrix> output,
2    GlobalRef<ArrayList<GlobalRef<Matrix>>> localAllWeights) {
3
4    finish(() -> {
5      for (Place place : places()) {
6        asyncAt(place, () -> {
7
8          // determine numberOfPlaces, rowsPerPlace, columnsPerPlace,
               pointsPerPlace ...
9          double[] out = output.get().getInnerArray();
10         double[] tmp = new double[out.length];
11
12         for (Place destinationPlace : places()) {
13           // create new matrix for the destinationplace and transpose it ...
14           // copy new matrix innerArray into tmp ...
15         }
16
17         ArrayList<GlobalRef<Matrix>> globalRefsList = localAllWeights.get();
18         for (GlobalRef<Matrix> placeWeights : globalRefsList) {
19           asyncAt(placeWeights.home(), () -> {
20             Matrix localWeights = placeWeights.get();
21             synchronized (localAllWeights) {
22               // copy correct part of tmp Array into correct position of
                     the localWeights matrix
23             }
24           });
25         }
26       });
27     }
28   });
29 }
```

Quellcode 4.14: FFT2D APGAS exchange Weights

Im letzten Schritt wird die Methode *transposeAndFinishFFT2D()* verwendet (Quell. 4.15), um die Fourier-Transformation für die zweite Dimension zu berechnen. Es wird dafür für jeden Place ein asynchroner Task erstellt (*Z. 4,5*), der die Gewichtsmatrix des jeweiligen Places abruft (*Z. 6,7*), diese transponiert (*Z. 8*) und die Fourier-Transformation ausführt (*Z. 9*).

```
1  private static void ↵
       transposeAndFinishFFT2D(GlobalRef<ArrayList<GlobalRef<Matrix>>> ↵
       localAllWeights,
2    GlobalRef<Matrix> output, int rowCount, int columnCount) {
3    finish(() -> {
4      for (Place place: places()) {
5        asyncAt(place, () -> {
6          ArrayList<GlobalRef<Matrix>> weightsList = localAllWeights.get();
7          Matrix localWeights = weightsList.get(here().id).get();
8          localWeights.transpose();
9          fft(localWeights, (Matrix)output.get(), rowCount, columnCount);
10       });
11     }
12   });
13 }
```

Quellcode 4.15: FFT2D APGAS Second FFT

4.3 Testumgebung

Das ITS-Servicezentrum der Universität Kassel betreibt einen Linux-Cluster für wissenschaftliche Anwendungen mit hohem Speicher und CPU-Bedarf. Der Cluster besteht aus einem Zugangsrechner und weiteren Rechnern für die eigentlichen Arbeitsaufträge. Programme können auf dem Cluster über den Zugangsrechner als Stapelverarbeitungsaufträge (BatchJobs) gestartet werden. Die Stapelverarbeitungsaufträge werden von dem System des Clusters verarbeitet und die darin enthaltenen Aufträge werden auf die anderen Rechner verteilt, sobald die dafür notwendigen Ressourcen (CPU, Speicher, Rechnerknoten) verfügbar sind. Der Linux-Cluster besteht aus AMD-Operon Mehrprozessorsystemen und ist in mehrere Partitionen aufgeteilt:

- **public:** homogene Rechner-Architektur mit 2-, 6- oder 16-Kern Doppelprozessorsysteme
- **minijobs:** maximal 60 Minuten Rechenzeit, Gigabit-Ethernet, 2-,6- oder 16-Kern Doppelprozessorsysteme
- **exec:** Infiniband Vernetzung, 16-Kern Doppelprozessorsysteme
- **mpi:** Infiniband Vernetzung, 16-Kern Doppelprozessorsysteme

4.4 Messergebnisse

Dieser Abschnitt präsentiert und diskutiert die Laufzeitergebnisse der HabaneroUPC++ und APGAS Implementierungen der Testprogramme K-Means und FFT2D. Das Testprogramm K-Means wurde auf dem ITS-Cluster mit 1, 2, 4, 8 und 16 Places als auch mit 1, 2, 4, 8 und 16 Threads ausgeführt. Als Startparameter wurden 20.000.000 Punkte, 50 Iterationen, 5 Cluster der Dimensionalität 4 und ein Schwellwert von 2.500 gewählt. Die Laufzeiten der APGAS- und HabaneroUPC++-Implementierung sind in den Tabellen 4.1 und 4.2 zusammengefasst.

Für die Intra-Place Parallelisierung der APGAS Implementierung für K-Means ergibt sich ein Speedup von 11,31, wenn die Laufzeiten für 1 Place und 16 Threads zugrunde

Places	Threads				
	1	2	4	8	16
1	3868 ms	1807 ms	905 ms	502 ms	342 ms
2	2077 ms	926 ms	467 ms	320 ms	302 ms
4	1042 ms	512 ms	240 ms	220 ms	220 ms
8	502 ms	239 ms	119 ms	102 ms	87 ms
16	247 ms	119 ms	62 ms	52 ms	52 ms

Tabelle 4.1: APGAS KMeans Laufzeiten

Places	Threads				
	1	2	4	8	16
1	-	11665 ms	7879 ms	6769 ms	6452 ms
2	-	7314 ms	4959 ms	6524 ms	6610 ms
4	-	9624 ms	6730 ms	10790 ms	85322 ms
8	-	6288 ms	7884 ms	57880 ms	174533 ms
16	-	timeout	timeout	timeout	timeout

Tabelle 4.2: HabaneroUPC++ KMeans Laufzeiten

gelegt werden. Der Speedup für die Inter-Place Parallelisierung beträgt 15,66 bei 16 Places und 1 Thread. In der Kombination beider Parallelisierungen ergibt sich ein Speedup von 74,38 für 16 Places und 16 Threads.

Die Laufzeiten der HabaneroUPC++ Implementierung für den K-Means Algorithmus können für 1 Thread nicht gemessen werden, da ein HabaneroUPC++ Programm immer mit mindestens 2 Threads gestartet werden muss, da ein Thread immer für den Communication Worker reserviert wird, siehe Kapitel 3.3. Aus diesem Grund wird für die sequentielle Laufzeit der Wert für 2 Threads herangezogen, um den Speedup zu bestimmen. Unter dieser Voraussetzung ergibt sich ein Speedup von 1,81 für die Intra-Place Parallelisierung für 16 Threads. Der Speedup für die Inter-Place Parallelisierung beträgt 1,86. Im Vergleich zu der APGAS Implementierung sind die Speedup Werte wesentlich geringer, für die Laufzeit von 8 Places und 16 Threads ist der Speedup geringer als 1,0 (0,07). Außerdem konnten die Laufzeiten für 16 Places nicht mehr ermittelt werden, da das Programm aufgrund von zu hohen Laufzeiten vorzeitig vom Cluster-System beendet wurde. Die HabaneroUPC++, als auch die HabaneroC++ Bibliotheken, befanden sich zum Zeitpunkt dieser Arbeit noch in der Entwicklung. In Rücksprache mit den

Places	Laufzeit
1	9629 ms
2	7604 ms
4	6151 ms
8	5480 ms
16	5123 ms

Tabelle 4.3: APGAS FFT2D Laufzeiten

Entwicklern der Bibliotheken wurde versucht eine Ursache für die Laufzeiten zu finden, diese konnte jedoch bis zum Ende der Arbeit nicht gefunden werden. Eine Vermutung war, dass der Work-Stealing Scheduler zu viele Stehlversuche durchgeführt und somit die CPU mit diesen blockiert hat.

Das zweite Testprogramm FFT2D wurde ebenfalls auf dem ITS-Cluster in verschiedenen Konfigurationen ausgeführt. Die Fourier-Transformation wurde im FFT2D Programm nur im Inter-Place Bereich parallelisiert, weshalb für die Anzahl der Threads konstant 2 verwendet wurde. Für die Places wurden 1, 2, 4, 8 und 16 verwendet. Als Startparameter wurden 4096 Zeilen und Spalten angegeben. Das FFT2D Programm testet die Performance des Datentransfers zwischen den Places, da insgesamt 2 Gigabyte an Daten ausgetauscht werden müssen. Die Laufzeiten für HabaneroUPC++ und APGAS sind in den Tabellen 4.3 und 4.4 zusammengefasst.

Der Speedup für 8 Places für die APGAS Implementierung beträgt 1,76, HabaneroUPC++ erreicht einen Speedup von 6,13. Die einzelnen Laufzeiten von HabaneroUPC++ sind jedoch deutlich höher als die von APGAS. Das zweite Testprogramm in der HabaneroUPC++ Implementierung wird ebenfalls von dem Fehler in den Bibliotheken beeinflusst, weshalb an dieser Stelle kein adäquater Vergleich der Laufzeiten zwischen APGAS und HabaneroUPC++ möglich ist.

Places	Laufzeit
1	43941 ms
2	26214 ms
4	13091 ms
8	7161 ms
16	76615 ms

Tabelle 4.4: HabaneroUPC++ FFT2D Laufzeiten

5 Zusammenfassung und Fazit

HabaneroUPC++ und APGAS implementieren beide das APGAS Modell und haben einige Gemeinsamkeiten, jedoch auch eine Reihe von Unterschieden. Beide Bibliotheken lassen sich in die Bereiche Inter-Place und Intra-Place Parallelisierung einteilen. Im Bereich der Inter-Place Parallelisierung implementieren beide asynchrone entfernte Tasks mit dem *asyncAt* Konstrukt und beide testen diese mit einem Konstrukt auf Terminierung. HabaneroUPC++ verwendet dazu *finish_spmd* und APGAS *finish*, beide sind jedoch semantisch äquivalent. In HabaneroUPC++ sind direkte Zugriffe auf globale Speicherbereiche möglich, weshalb diese Bibliothek ein weiteres Konstrukt für die Synchronisierung solcher Zugriffe bereitstellt, das *barrier* Konstrukt. APGAS bietet Mechanismen an, um eine fehlertolerante und elastische Ausführung der Programme zu ermöglichen, HabaneroUPC++ besitzt keine solche Mechanismen.

Das globale Speichermodell beider Bibliotheken ähnelt sich ebenfalls, jedoch unterscheiden sich die verwendeten Konstrukte. HabaneroUPC++ verwendet verteilte Objekte und globale Zeiger, APGAS bietet dafür globale Referenzen. Globale Referenzen und globale Zeiger ähneln sich vom Grundkonzept, werden jedoch unterschiedlich implementiert. HabaneroUPC++ besitzt zusätzlich Reservierungs- und Freigabefunktionen für den globalen Speicher, ähnlich zu den aus C bekannten Funktionen. Außerdem bietet HabaneroUPC++ synchrone und asynchrone Kopierfunktionen, um einen globalen Speicherbereich in einen anderen zu kopieren. APGAS verwendet hingegen zum Kopieren von Daten asynchrone Tasks.

HabaneroUPC++ und APGAS unterscheiden sich ebenfalls in ihrem Ausführungsmodell, HabaneroUPC++ verwendet das SPMD und APGAS ein Task-paralleles Ausführungsmodell. Besonders deutlich wird dies bei der Programmierung von Haba-

neroUPC++ Programmen, denn dort muss zu jedem Zeitpunkt darauf geachtet werden, in welchem Kontext ein asynchroner Task gestartet wird. Ob aus einem asynchronen Task oder aus der main Funktion heraus. Eine weitere Besonderheit von HabaneroUPC++ sind kollektive Funktionen, welche nur in einem SPMD Ausführungsmodell funktionieren und daher in APGAS nicht zur Verfügung stehen.

Im Bereich der Intra-Place Parallelisierung besitzen beide Bibliotheken ebenfalls einige Gemeinsamkeiten und Unterschiede. Beide verwenden einen Work-Stealing Algorithmus, um asynchrone lokale Tasks effizient auf die zur Verfügung stehenden Prozessoren zu verteilen. Diese Algorithmen ähneln sich stark in beiden Bibliotheken, jedoch besitzt HabaneroUPC++ einen dedizierten Thread, der ausschließlich für die Kommunikation mit anderen Places verwendet wird, der *Communication Thread*. Aus diesem Grund fällt in HabaneroUPC++ immer einer der Prozessoren für den *Communication Thread* weg.

Asynchrone lokale Tasks werden in beiden Bibliotheken mit einem *async* Konstrukt implementiert. Auch die Terminierungserkennung ist in beiden Bibliotheken mit einem *finish* Konstrukt implementiert. Darüber hinaus besitzt HabaneroUPC++ noch zwei weitere Async Konstrukte, *asyncAwait* und *forasync*. Außerdem ist es in HabaneroUPC++ möglich die Architektur des Computersystems eines Places näher zu beschreiben, indem diese mit einem Hierarchischen Place Baum beschrieben wird.

Beide Bibliotheken besitzen eine hohe Ähnlichkeit zueinander, HabaneroUPC++ besitzt jedoch viele Erweiterungen und zusätzliche Konzepte, die APGAS nicht besitzt wie Hierarchische Place Bäume, verteilte Objekte, *asyncAwait* und *forasync*. Außerdem verwendet HabaneroUPC++ das SPMD-, APGAS das Task-Parallele Ausführungsmodell. Dies bietet einige Vorteile, wie kollektive Funktionen, erhöht allerdings auch die Komplexität der Programme. APGAS hingegen bietet eine fehlertolerante und elastische Ausführung der Programme, was in HabaneroUPC++ nicht möglich ist. Aufgrund des aktuellen Entwicklungsstandes der HabaneroUPC++ und HabaneroC++ Bibliotheken war es nicht möglich einen praktischen Vergleich anhand der Laufzeiten durchzuführen.

Für die HabaneroUPC++ Bibliothek sind neben den Fehlerkorrekturen noch eine Reihe von Erweiterungen geplant. Darunter zählt zum Beispiel ein globales Work-Stealing. Ein solches existiert bereits für X10 und wird als GLB (Global Load Balancing) bezeichnet. Eine Portierung dessen zu APGAS kann in naher Zukunft erwartet werden. Die Forschung und Entwicklung der beiden APGAS Bibliotheken sind noch nicht abgeschlossen, weshalb in Zukunft noch mit einigen Neuerungen gerechnet werden kann.

Quellcodeverzeichnis

4.1 KMeans APGAS ClusterState Type 27
4.2 KMeans APGAS Inter-Place Implementation 29
4.3 KMeans APGAS Intra-Place Implementation 30
4.4 KMeans HabaneroUPC++ Global Pointers 31
4.5 KMeans HabaneroUPC++ Inter-Place Implementation 33
4.6 KMeans HabaneroUPC++ Results exchange 33
4.7 FFT2D HabaneroUPC++ complex_t Type 35
4.8 FFT2D HabaneroUPC++ Implementation, main 36
4.9 FFT2D HabaneroUPC++ Implementation, fft 38
4.10 FFT2D APGAS Matrix Type 39
4.11 FFT2D APGAS Main 40
4.12 FFT2D APGAS Weights Distribution 41
4.13 FFT2D APGAS First FFT 41
4.14 FFT2D APGAS exchange Weights 43
4.15 FFT2D APGAS Second FFT 44

Abbildungsverzeichnis

2.1 PGAS-Modell 5

3.1 Work-Stealing Gemeinsamkeiten 20

3.2 HabaneroUPC++ Work-Stealing 22

Literaturverzeichnis

[1] *FFT2D UPC++ Example*. https://bitbucket.org/upcxx/upcxx/src/b122d9d77b4a7a2130cc78489d0140d8a54906ba/examples/fft2d/?at=master, . – abgerufen: 31.03.2016

[2] *KMeans APGAS Example*. https://github.com/x10-lang/x10/blob/master/apgas.examples/src/apgas/examples/KMeans.java, . – abgerufen: 31.03.2016

[3] *The X10 Programming Language*. http://www.x10-lang.org, . – abgerufen: 06.03.2016

[4] CAVÉ, V. ; ZHAO, J. ; SHIRAKO, J. ; SARKAR, V. : Habanero-Java: The New Adventures of Old X10. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. New York, NY, USA, 2011 (PPPJ '11), S. 51–61

[5] CENTER, I. O. S. T.: *Open Community Runtime*. https://01.org/open-community-runtime/, . – abgerufen: 06.03.2016

[6] CHAMBERLAIN, B. ; CALLAHAN, D. ; ZIMA, H. : Parallel Programmability and the Chapel Language. In: *Int. J. High Perform. Comput. Appl.* 21 (2007), Aug., Nr. 3, S. 291–312

[7] CUNNINGHAM, D. ; GROVE, D. ; HERTA, B. ; IYENGAR, A. ; KAWACHIYA, K. ; MURATA, H. ; SARASWAT, V. ; TAKEUCHI, M. ; TARDIEU, O. : Resilient X10: Efficient Failure-aware Programming. In: *SIGPLAN Not.* 49 (2014), Febr., Nr. 8, S. 67–80

[8] EL-GHAZAWI, T. ; SMITH, L. : UPC: Unified Parallel C. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. New York, NY, USA, 2006 (SC '06)

[9] K. A. YELICK, G. P. C. M. B. L. A. K. P. N. H. S. L. G. D. G. P. C. L. Semenzato S. L. Semenzato ; AIKEN, A. : Titanium: A High-Performance Java Dialect. In: *Concurrency: Practice and Experience* 10 (1998), Nr. 11-13

[10] KAMIL, A. ; ZHENG, Y. ; YELICK, K. : A Local-View Array Library for Partitioned Global Address Space C++ Programs. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. New York, NY, USA, 2014 (ARRAY'14), S. 26:26–26:31

[11] KUMAR, V. : *Habanero-C++*. `http://habanero-rice.github.io/hcpp/`, . – abgerufen: 04.03.2016

[12] KUMAR, V. ; ZHENG, Y. ; CAVÉ, V. ; BUDIMLIĆ, Z. ; SARKAR, V. : HabaneroUPC++: A Compiler-free PGAS Library. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. New York, NY, USA, 2014 (PGAS '14), S. 5:1–5:10

[13] LEA, D. : A Java Fork/Join Framework. In: *Proceedings of the ACM 2000 Conference on Java Grande*. New York, NY, USA, 2000 (JAVA '00), S. 36–43

[14] LLOYD, S. P.: Least squares quantization in PCM. In: *IEEE Transactions on Information Theory*, 1982, S. 129–137

[15] NUMRICH, R. W. ; REID, J. : Co-array Fortran for Parallel Programming. In: *SIGPLAN Fortran Forum* 17 (1998), Aug., Nr. 2, S. 1–31

[16] SARASWAT, V. ; ALMASI, G. ; BIKSHANDI, G. ; CASCAVAL, C. ; CUNNINGHAM, D. ; GROVE, D. ; KODALI, S. ; PESHANSKY, I. ; TARDIEU, O. : The asynchronous partitioned global address space model. In: *The First Workshop on Advances in Message Passing*, 2010, S. 1–8

[17] TARDIEU, O. : The APGAS Library: Resilient Parallel and Distributed Programming in Java 8. In: *Proceedings of the ACM SIGPLAN Workshop on X10*. New York, NY, USA, 2015 (X10 2015), S. 25–26

[18] TASIRLAR, S. ; SARKAR, V. : Data-Driven Tasks and Their Implementation. In: *Proceedings of the 2011 International Conference on Parallel Processing*. Washington, DC, USA, 2011 (ICPP '11), S. 652–661

[19] UNIVERSITY, R. : *Habanero-C Übersicht*. `https://wiki.rice.edu/confluence/display/HABANERO/Habanero-C`, . – abgerufen: 06.03.2016

[20] UNIVERSITY, R. : *HClib: Eine Bibliotheksimplementierung der Habanero-C Programmiersprache*. `http://habanero-rice.github.io/hclib/`, . – abgerufen: 06.03.2016

[21] YAN, Y. ; ZHAO, J. ; GUO, Y. ; SARKAR, V. : Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In: *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg : Springer-Verlag, 2010 (LCPC'09), S. 172–187

[22] ZHENG, Y. ; KAMIL, A. ; DRISCOLL, M. B. ; SHAN, H. ; YELICK, K. : UPC++: A PGAS Extension for C++. In: *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Washington, DC, USA : IEEE Computer Society, 2014 (IPDPS '14), S. 1105–1114