UNI KASSEL
VERSITÄT

Universität Kassel

Bachelorarbeit

Isolierung von APGAS Benchmarks unter Anwendung von Containern in einer HPC-Umgebung

Fabian Wurmbach
33308410

Kassel, 15. September 2019

Gutachter:
Prof. Dr. Claudia Fohry
Prof. Dr. Albert Zündorf

Inhaltsverzeichnis

Selbstständigkeitserklärung **II**

1 Einleitung **1**

2 Grundlagen **3**

2.1 Virtualisierung . . . 3
2.2 Container . . . 5
2.3 Docker . . . 7
2.4 APGAS . . . 9

3 Container für APGAS **11**

3.1 Das Root-Problem . . . 11
3.2 Singularity . . . 11
3.3 UDocker . . . 12
3.4 Shifter . . . 14
3.5 APGAS-Container . . . 15

4 Implementierung **16**

4.1 Launcher . . . 16
4.2 Dockerimage . . . 17
4.3 Singularity-Container . . . 19
4.4 UDocker-Container . . . 20
4.5 Ausführung auf Cluster . . . 21
4.5.1 Singularity . . . 22
4.5.2 UDocker . . . 23

5 Performancevergleich **25**

5.1 Messungen . . . 25
5.2 Auswertung . . . 26
5.3 Zusammenfassung . . . 29

6 Fazit **30**

Literaturverzeichnis **III**

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Kassel, 15. September 2019

Fabian Wurmbach

1 Einleitung

Container werden in der Welt der Informatik immer beliebter. Sie ermöglichen die Ausführung von Programmen in einer isolierten Umgebung, in welcher Einflüsse von außen ausgeschlossen werden. Container werden auf einem Betriebssystem ausgeführt, jedoch werden sie gleichzeitig so vom laufenden Betriebssystem abgekapselt, dass sie ohne Seiteneffekte Programme ausführen können. Viele Firmen benutzen mittlerweile die Technologie der Container, um ihre Programme an Kunden zu verteilen.

Auch im wissenschaftlichen und forschenden Bereich werden Container immer häufiger benutzt, um Programme in einer High-Performance-Computing-Umgebung (HPC-Umgebung) auszuführen. Dies ermöglicht beispielsweise die Speicherung von Entwicklungsständen eines Programms. Hierbei wird die Art, wie der Container ausgeführt wird, gespeichert und es ist möglich einen Container mehrmals auszuführen. Dies ermöglicht Entwicklungsstände zu speichern und zu einem späteren Zeitpunkt erneut auszuführen.

Ziel dieser Arbeit ist es, die Eigenschaften eines Containers auf das parallele Programmiersystem APGAS zu übertragen. Es wird mit Hilfe verschiedener Containervirtualisierungssysteme, welche es ermöglichen Container zu erstellen, ein Container mit APGAS erstellt. Anschließend wir dieser Container in einer HPC-Umgebung ausgeführt und mit Fokus auf die Geschwindigkeit der Berechnung, mit der Ausführung APGAS ohne Container verglichen.

Zunächst wurde versucht mit Hilfe des Containervirtualisierungssystems Docker einen APGAS-Container zu erstellen. Jedoch trat das Problem auf, dass ein durch Docker erstellter Container nicht in einer HPC-Umgebung benutzt werden kann, da er Root-Rechte zur Ausführung benötigt. Deshalb wurde im Rahmen der Bachelorarbeit nach Alternativen gesucht: Die Systeme Singularity und UDocker

erlaubten es, einen APGAS-Container zu erstellen, der anschließend ohne Root-Rechte ausführbar ist.

Jedoch war es nötig den APGAS-Quellcode an die Syntax des jeweiligen Containervirtualisierungssystems anzupassen. Des Weiteren mussten Skripte zur Ausführung des APGAS-Containers geschrieben werden. Die mit Singularity beziehungsweise Udocker erstellten APGAS-Container wurden auf dem Kassel-Cluster getestet. Dabei führten beide Container das UTS-Benchmark und das BC-Benchmark aus. So war es möglich die Ausführungszeiten der beiden Container untereinander sowie mit der Ausführung ohne Container zu vergleichen. Es stellte sich heraus, dass die Ausführungszeit mit den beiden Containern ähnlich und jeweils geringfügig langsamer als ohne Container war.

Im zweiten Kapitel werden zunächst Grundlagen zu Containern und das Programmiersystem APGAS vorgestellt. Anschließend wird in Kapitel 3 das Problem, mit der Benutzung von Docker erläutert. Als Alternative werden dann die Containervirtualisierungssysteme Singularity und UDocker vorgestellt. Zusätzlich hierzu wird das System Shifter als Alternative vorgestellt. Anschließend wird überblicksmäßig erklärt, wie APGAS angepasst werden musste, damit es als Container funktioniert. Kapitel 4 gibt beschreibt die Anpassung im Detail und enthält insbesondere eine Anleitung wie ein APGAS-Container mithilfe von Singularity beziehungsweise UDocker erstellt wird. Im darauffolgenden Kapitel 5 wird die Performance der Container zueinander und zur normalen Ausführung von APGAS verglichen und analysiert. Abschließend gibt Kapitel 6 ein Fazit.

2 Grundlagen

In diesem Kapitel wird zunächst der Begriff der Virtualisierung erklärt, damit anschließend die Containervirtualisierung erklärt werden kann. Anschließend werden das verbreitete System Docker sowie das Programmiersystem APGAS vorgestellt.

2.1 Virtualisierung

Virtualisierung bietet die Möglichkeit einen Computer zu emulieren das heißt, es wird ein virtuelles Betriebssystem auf virtueller Hardware ausgeführt. Dies wird durch den sogenannten Hypervisor ermöglicht. Ein Hypervisor ist eine Software, die es ermöglicht virtuelle Hardware zu generieren. Hierdurch wird Hardware, welche physisch nicht existent ist, auf einem Computer nutzbar. Anschließend lässt sich auf dieser Hardware ein Betriebssystem aufsetzten. Dieses Konstrukt aus virtueller Hardware und Betriebssystem wird auch *Virtual Machine*genannt. Es gibt verschiedene Möglichkeiten, wie ein Hypervisor implementiert sein kann.

Zum einen gibt es Hypervisoren, die direkt auf der real existierenden Hardware sitzen und von dort aus virtuelle Hardware erzeugen. Der Hypervisor dient hier als Bindeglied von existenter Hardware und emulierter Hardware. Anschließend wird auf der virtualisierten Hardware ein virtuelles Gast-Betriebssystem erzeugt, das als Grundlage die virtuelle Hardware nutzt. Dieses muss jegliche Treiber und andere Betriebssystem spezifische Software erstellen. Bei dieser Art des Hypervisors ist es also irrelevant, welches Betriebssystem ausgeführt wird, da direkt auf die Hardware des Computers zugegriffen wird.

Der gleiche Hypervisor kann mehrere Sätze an virtueller Hardware erzeugen. Es ist also möglich, mehrere verschiedene Gast-Betriebssysteme nebeneinander auf einer realen Hardware laufen zu lassen. Falls die Gast-Betriebssysteme auf die existierende Hardware zugreifen müssen, um angeschlossene Geräte anzusprechen,

dient der Hypervisor als Verwaltungstool und kommuniziert zwischen den Systemen. Diese Art des Hypervisors ist in Abbildung 2.1 zu sehen.

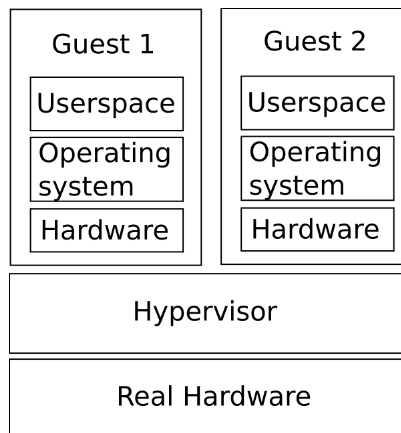

Abbildung 2.1: Hypervisor auf Hardware

[5]

Eine andere Art des Hypervisors ist in einem laufenden Betriebssystem anzusiedeln. Hier wird durch den Hypervisor wieder Hardware emuliert, auf der anschließend ein Betriebssystem ausgeführt werden kann. Dieser Hypervisor sitzt also nicht auf der realen Hardware, sondern wird als Software in einem laufenden Betriebssystem ausgeführt. Diese Art des Hypervisors ist im Gegensatz zur ersten Art vom Host-Betriebssystem abhängig, da die Hypervisorsoftware auf einem Betriebssystem ausgeführt wird. Diese zweite Art des Hypervisors muss jedoch auch jegliche Treiber ,für die emulierte Hardware, sowie andere Betriebssystem spezifische Software erstellen.

Virtualisierung ermöglicht das Testen von Software auf verschiedener Hardware und auf verschiedenen Betriebssystemen. Zum anderen ermöglicht sie Software in einer immer gleichen Umgebung auszuführen, um etwaige Seiteneffekte, die durch verschiedene Hardware oder andere Betriebssystemsoftware ausgelöst werden, zu umgehen.

2.2 Container

Eine Dritte Art der Virtualisierungen wird durch Container erreicht. Container werden durch Containervirtualisierungssysteme erzeugt und führen ein Betriebssystem aus. Man spricht hier von „Lightweight Operating System Virtualization“, also einem virtuellen Betriebssystem, das wenig Ressourcen benötigt.

Ein Containervirtualisierungssystem ist mit einem Hypervisor der zweiten, oben gennanten Art, zu vergleichen. Containervirtualisierungssysteme werden genauso auf einem Host-Betriebssystem ausgeführt. Zusätzlich hierzu benötigen Container meist ein Linux-Betriebssystem, da sie den Linux-Kernel benutzen. Der Unterschied zwischen einem Hypervisor und einem Containervirtualisierungssystem ist der, dass ein Containervirtualisierungssystem keine neue virtuelle Hardware emuliert, sondern die Hardware des Hosts durch den Linuxkernel mitbenutzt. Das bedeutet, dass Containervirtualisierung damit wesentlicher enger am Host-Betriebssystem hängt, da es das Host-System direkt benutzt, anstatt ein komplettes Virtuelles System zu erstellen und abzukapseln. Jedoch werden Container dank der isolierten Ausführung auf der Host-Hardware durch den Linux-Kernel so ausgeführt, dass es nicht zu Seiteneffekten kommen kann. Falls jedoch die Host-Hardware ausgelastet ist, wird auch der Container langsamer ausgeführt.

Der Linuxkernel ist zentraler Bestandteil eines Linux-Betriebssystems und ist zuständig für Prozessverwaltung und Datenorganisation. Durch die Fähigkeit des Linuxkernels Prozesse zu verwalten, ist es möglich, einen isolierten Bereich zu erschaffen, in welchem, durch Containervirtualisierungssysteme erstellte, Container anschließend ausgeführt werden können. [5] Dies bedeutet, dass Container zwar parallel zum Host ausgeführt werden, diese sind aber, anders als durch Hypervisor erstellte Betriebssysteme, nicht vom Host-Betriebssystem

abhängig.[4] Abbildung 2.2 zeigt, wie ein Container in einem laufendem Betriebssystem ausgeführt wird.

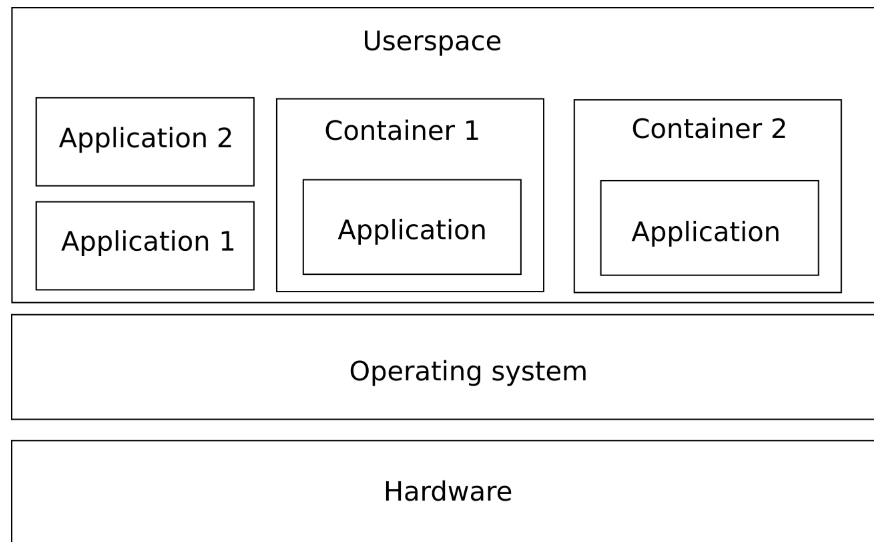

Abbildung 2.2: Container in Betriebssystem
[5]

Container haben weiterhin Zugriff auf die Namensräume des Linuxkernels. Dies führt dazu, dass sie Zugriff auf Netzwerkadressen anderer Computer des Netzwerks haben. Dies stellt sich später in dieser Arbeit als nützlich heraus. Des Weiteren ist der Benutzer, der einen Container startet, gleichzeitig der Benutzer innerhalb des Containers.

Ein Container beinhaltet nur die nötigsten Dateien zur Ausführung desjenigen Programms, welches in ihnen ausgeführt werden soll. Dies bietet einen Vorteil gegenüber der klassischen Ausführung mit Hyperviorn, da bei diesen ein komplettes Betriebssystem, samt Treibern, emuliert werden muss. Durch die isolierte Ausführung von Containern, ist es möglich, Software in einem Vakuum auszuführen, in welchem die Ausführung nicht durch andere Prozesse beeinträchtigt wird. Weiterhin ist es möglich, einen Container auf einem fremden Linux-Betriebssystem, auf welchem die Containervirtualisierungssoftware installiert ist, auszuführen.

2.3 Docker

Seit 2013/2014 bietet die Docker Inc. ihre Open Source Anwendung Docker an. Docker ist ein System, das es ermöglicht, Container zu erstellen und ist das Flaggschiff der heutigen Containervirtualisierungssysteme.

Um einen Docker-Container zu erstellen, ist zunächst ein Docker-Image vonnöten, aus welchem später eine beliebige Anzahl von Docker-Container erstellt werden kann.

Durch das von Docker betriebene Docker-Hub hat man Zugriff auf eine Vielzahl von Docker-Images, die von anderen Docker-Usern hochgeladen und zur freien Verfügung bereitgestellt wurden.[3] Als Beispiel folgen zur Veranschaulichung Befehle in Pseudocode mit ihrer Erklärung. Möchte man ein Docker-Image lokal, um einen Container zu erstellen, benutzen, muss der Befehl:

```
1 docker pull <name>:<version >.
```

ausgeführt werden. Anschließend wird das Docker-Image auf das eigene System geladen und man kann dieses Docker-Image benutzen. Dank des Versionen-Flags, das man einem Docker-Image anhängt, ist es möglich eine Versionskontrolle zu benutzen. Mit dem Befehl:

```
1 docker run <nameDesImages>
```

wird anschließend ein Container aus dem geladenen Docker-Image erstellt und ausgeführt.

Möchte man jedoch ein eigenes Image erstellen, kann man hierfür ein sogenanntes Dockerfile benutzen. Hier wird zunächst angegeben, welches Image als sogenanntes Basis-Image benutzt wird. Dieses Basis-Image muss entweder lokal - falls man ein anderes selbst erstelltes Image benutzen möchte, das nicht im Docker-Hub geladen ist - oder im Docker-Hub verfügbar sein.

Anschließend wird angegeben, welche Dateien in das Image geladen werden, welche Umgebungsvariablen gesetzt werden und welche Befehle ausgeführt werden sollen, wenn der Container gestartet wird.

Eine andere Form zur Erstellung eines Images ist das sogenannte Multi-Stage-Building. Hierbei wird im Dockerfile zunächst, wie beim vorherigen Fall, ein Basis-Image angegeben. Dieses kann man nun einer Variable zuordnen. Anschließend kann man ein zweites Basis-Image angeben und vom vorherigen Image teile in das zweite Image übernehmen. Dies ist sinnvoll, wenn man einen Container haben möchte, der ein bestimmtes Programm enthalten soll. Muss dieses Programm jedoch vorher noch kompiliert und gebaut werden, ist es von Vorteill ein Image als *builder* anzugeben und anschließend nach kompilieren des Programms, dieses kompilierte Programm dem finalen Image zu übergeben.

Nach Beendigung des letzten Befehls eines Docker-Containers wird der Container heruntergefahren. Anschließend kann jedoch aus dem gleichen Image ein neuer Container gestartet werden. Dieser Container wird genauso ausgeführt, wie der Vorige, da die Containervisualisierung vorgibt, dass Container im System abgekapselt von allen anderen Teilen des Betriebssystems ausgeführt werden.

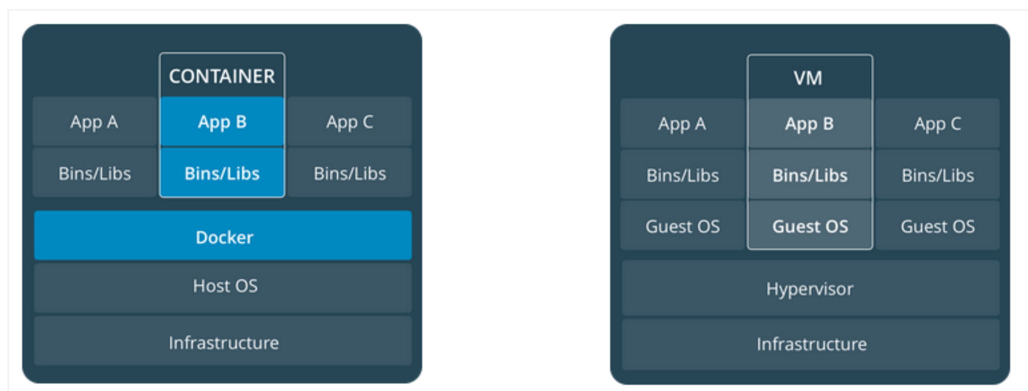

Abbildung 2.3: Docker Vs VM [1], [2]

Abbildung 2.3 zeigt auf der linken Seite, wie ein Container durch Docker im System anzusiedeln ist. Die Docker-Engine wird von dem jeweiligen System ausgeführt und über diese Engine werden Container gestartet. Diese Engine ist das Containervirtualisierungssystemen von Docker. Ein Container beinhaltet hier alle *BINS/LIBS* welche zum Ausführen einer Anwendung gebraucht werden. Es werden also, wie in Kapitel 2.2 erwähnt, nur die zur Ausführen einer Anwendung benötigten Dateien in den Container geladen.

Die rechte Seite der Abbildung 2.3 zeigt im Vergleich, wie eine Virtual Machine, die durch einen Hypervisor zur Verfügung gestellt wird, in einem Computersystem anzusiedeln ist. Wie in Kapitel 2 bereits erwähnt, wird durch den Hypervisor ein komplettes Betriebssystem emuliert und nicht nur die zur Ausführung der Anwendung benötigten Daten geladen. Ein Docker-Container benötigt also nur einen Bruchteil der Ressourcen, die zum Ausführen eines virtuellen Computers benötigt werden.

Docker benutzt das sogenannte *AuFS* (Advanced Multi-layeres Unification Filessystem)[3] zur Erstellung von Images. Das *AuFS* benötigt zwei verschiedene Dateisysteme, ein Beschreibbares und ein nicht Beschreibbares. Diese beiden Dateieinsysteme werden nun *übereinander gelegt*. Soll eine Datei gelesen werden, wird zunächst versucht, sie von dem Beschreibbaren Dateisystem zu lesen. Ist dies nicht möglich, wird sie vom nicht beschreibbaren versucht zu lesen. Im Beispiel von Docker sind diese beiden Dateisysteme das neu zu schreibende Docker-Image und das Basis-Docker-Image. Möchte man also ein neues Docker-Image schreiben, werden die Daten vom Basis-Image übertragen und anschließend neue Dateien vom Hostsystem hinzugefügt.

2.4 APGAS

Das Asynchronous Partitioned Global Address Space Programmiersystem ist ein paralleles Programmiersystem. APGAS wurde aus dem X10 Programmiersystem

heraus entwickelt und versucht dessen Vorteile den Java Entwicklern näher zu bringen. [10]. Das PGAS (Partitioned Global Address Space) Model sieht einen Computerknoten als einen sogennanten Place, um auf diesem Place etwas zu berechnen. Hierbei wird ein Place so modeliert, dass dieser einen lokalen Speicherabschnitt besitzt und eine gewisse Anzahl von Prozessoren zur Verfügung stehen hat. Weiterhin haben verschiedene Places Zugriff auf einen Globalen Addressraum. Die Umsetzung dieses Models ermöglicht schnellen Zugriff auf lokalen Speicher und langsameren Zugriff auf den, von allen Places erreichbaren, Globalen Addressraum.

Das APGAS Programmiersystem setzt dieses Model um und erweitert es durch Asynchronität. Hierdurch wird durch den *async* Befehl eine sogennante Activity erzeugt. Eine Activity ist ein Teilprogramm, welches ausgeführt werden kann. Durch den *async* Befehl ist es nun möglich eine solche Activity auf dem aktuellen Place zu starten und diesem dem lokalen *Fork-Join-Pool* der JVM (Java Virtual Machine) hinzuzufügen.[9] Hierbei wird diese Activity parallel ausgeführt. Es muss also nicht auf das Ergebnis der Activity gewartet werden sondern es wird ermöglicht währenddessen andere Berechnungen durchzuführen beziehungsweise andere Activitys zu starten. Weiterhin ist es möglich durch den *asyncAt* Befehl eine solche Activity auf einem anderen Place zu starten und sie dort dem *Fork-Join-Pool* der laufenden JVM hinzuzufügen.

Ein Beispiel-Programmdurchlauf sieht also so aus, dass zunächst auf einem Knoten das Programm gestartet wird und anschließend die verschiedene Places gestartet werden. Hiernach werden durch den ersten Knoten dynamisch verschiedene Activitys erstellt und den einzelnen Places zugeordnet, auf welchen diese Activitys berechnet werden.

3 Container für APGAS

In Abschnitt 3.1 wird erklärt, dass Docker nicht ausreicht, um einen Container für das APGAS-Programmiersystem zu erstellen. Deshalb werden als Alternative die Containervirtualisierungssysteme Singularity und UDocker vorgestellt, die es ermöglichen, Container in einer HPC-Umgebung zu benutzen. Anschließend wird konzeptionell darauf eingegangen, wie das APGAS-Programmiersystem angepasst werden musste, damit es als Container in einer HPC-Umgebung funktioniert.

3.1 Das Root-Problem

Wenn ein Container von einem Benutzer ausgeführt wird, wird dieser Benutzer auch innerhalb des Containers angelegt und der Benutzer innerhalb des Containers ist der Gleiche, wie außerhalb. Da ein Docker-Container Root-Rechte benötigt um ausgeführt zu werden, ist der Docker-Benutzer innerhalb des Containers Root. Innerhalb einer HPC-Umgebung arbeiten mehrere Benutzer gleichzeitig auf verschiedenen Rechnerknoten. Die Benutzer haben lediglich Eingeschränkte-Rechte. Hätte ein Benutzer Root-Rechte, könnte er Teile der HPC-Umgebung ändern und es würde zu einer Sicherheitslücke kommen.

Um trotzdem Container in einer HPC-Umgebung benutzen zu können, wurden andere Containervirtualisierungssysteme entwickelt, die es ermöglichen, als normaler Benutzer einen Container auszuführen. Zwei dieser Systeme – Singularity und UDocker – wurden in dieser Arbeit benutzt, um einen APGAS-Container zu erstellen, der anschließend auf dem Kassel-Cluster ausführbar ist.

3.2 Singularity

Singularity wurde, speziell für das wissenschaftliche Arbeiten mit Containern entwickelt. Die Container werden nicht tief im System verankert, sodass man

keine Root-Rechte benötigt um ihn auszuführen, sondern es wird eine einzelne Datei erstellt, welche den Container beinhaltet. Hierbei spricht man auch von Non-Root-Containern. Der Container wird bei der Ausführung an das Host-System angebunden und anschließend als eigener Prozess isoliert ausgeführt. Beim Ausführen des Containers ist der Benutzer innerhalb des Containers der gleiche wie außerhalb des Containers. Weiterhin ist es möglich, durch sogenannte *bind points* anzugeben, auf welche Dateien des Host-Rechners der Container zugreifen kann. [8]

Dennoch muss ein Singularity Container zunächst von einem Rootuser erstellt werden. Hierbei können zum einen bereits erstellte Container verwendet werden, welche ähnlich zu Docker aus dem sogenannten *shub* heruntergeladen werden können, um diese umzuschreiben oder andere Dateien dem Container hinzuzufügen.

Zum anderen kann man auch ein bestehendes Docker-Image benutzen, um einen Singularity Container zu erstellen. In einem def-file wird angegeben, woher das benötigte Docker-Image bezogen werden soll und wie dieses heißt.

Für beide Fälle muss ein sogenanntes runscript existieren. Es gibt an, welche Befehle ausgeführt werden sollen, wenn der Container gestartet wird.

Anschließend erhält man ein sif-File. Dieses kann mit Singularity eingelesen und darauffolgend ausgeführt werden. Hierfür werden keine Root-Rechte benötigt, weswegen man nach dem Erstellen des Containers diesen auf die gewünschte Zielplattform - auf welcher man selber keine Root-Rechte hat - transferieren und dort mittels Singularity ausführen kann.

3.3 UDocker

Genau wie Singularity versucht UDocker das Problem von Docker aufzugreifen, dass Root-Rechte zum Ausführen von Containern benötigt werden. UDocker bietet im Gegensatz zu Singularity jedoch lediglich eine Erweiterung zu Docker

und es ist nicht möglich, mit UDocker einen eigenen Container ohne ein Docker-Image zu erstellen. UDocker ist ein in Python geschriebenes System und versteht Container als eine Abkapselung von Prozessen. Es werden Docker-Container Daten analysiert und aus jenen ein UDocker-Container erstellt. Dank verschiedener Prozessverwaltungstools entsteht nun ein UDocker-Container, welcher keine Root-Rechte zum Ausführen benötigt.[6] Des Weiteren sind auch für die Installation von UDocker – im Gegensatz zu Singularity- keine Root-Rechte nötig.

Die Entwickler von UDocker stellten des Weiteren fest, dass nicht alle Funktionen von Docker in der wissenschaftlichen Nutzung benötigt werden [6]. Sie fokussierten deswegen ihren Schwerpunkt auf sechs Kernpunkte:

Mobilität Container sollten von jedem auf jedem möglichen System benutzt werden können

Zuweisen von Speichern Verschiedene Ordner sollen dem Container flexibel hinzugefügt werden können

Softwarezuweisung Dockerfiles sollen verwendet werden um Software einem Container zuzufügen

Wiederverwendbarkeit von Containern Container sollten direkt wiederverwendet werden können, ohne wiederholt aus einem Image erstellt werden zu müssen

Teilen Es sollte möglich sein, anderen seine Container einfach zu Verfügung zu stellen

Einfache Benutzeroberfläche UDocker soll einfach zu bedienen sein

Um einen UDocker-Container zu erstellen, wird zunächst ein Docker-Image benötigt. Dieses Docker-Image wird in das System geladen und dort in ein

UDocker-Image umgewandelt. Anschließend kann man das UDocker-Image benutzen, um einen UDocker-Container zu erstellen. Dieser Container ist nun im UDocker-Tool gespeichert und lässt sich ohne Abhängigkeit zum Image starten und wieder aufrufen.

Da nur ein tar-file des Docker-Images benötigt wird, ist es möglich, das UDocker-Tool zu benutzen ohne Root-Rechte zu haben. UDocker sieht sich ,im Gegensatz zu Singularity ,nicht als Alternative zu Docker, sondern möchte nur als Hilfsprogramm zu Docker agieren. Es können also nur Container erstellt werden, wenn vorher Docker-Images verfügbar sind.

3.4 Shifter

Ein weiteres Containersystem, das in einer HPC-Umgebung benutzt werden kann, ist Shifter. Shifter wird vom NERSC (National Energy Research Scientific Computing Center) implementiert und gewartet. Es wird auch dort am Cluster betrieben. Shifter benutzt, wie Singularity und UDocker, ein Docker-Image als Basis und daraus wird anschließend ein Container erstellt. Shifter erlaubt es, dem Benutzer verschiedene Docker-Images zu laden, die anschließend konvertiert werden. Diese konvertierten Images sind anschließend im Shifter-System gespeichert und können von hier als Container gestartet werden.

Shifter ist, anders als die in dieser Arbeit verwendeten Containervirtualisierungssysteme, eng mit dem Cluster, auf welchem es installiert ist, verbunden. Dies ermöglicht dem Nutzer einfache Bedienmöglichkeit. Es wird lediglich ein Image geladen und von Nutzer anschließend ausgeführt. Shifter verwaltet anschließend die Ausführung des Containers, indem diese Ausführung an das Verwaltungstool des Clusters weitergegeben. Nach Beendigung eines solchen *Shifter-Jobs* erhält der Benutzer eine Benachrichtigung genauso, wie er eine Benachrichtigung des Verwaltungstools bekommen würde.[7]

Shifter war für diese Arbeit nicht geeignet, da es nicht möglich war Shifter auf dem Kassel-Cluster zu installieren.

3.5 APGAS-Container

Um einen APGAS-Container zu erstellen, genügt es nicht den APGAS-Quelltext in einen Container zu packen, diesen zu kompilieren und anschließend auszuführen. Zusätzlich musste der APGAS-Quelltext für die Ausführung in einem Container angepasst werden.

Es musste des Weiteren eine Möglichkeit gefunden werden, um von innerhalb eines Containers neue Places auf anderen Knoten zu starten. Um dies zu ermöglichen wurde ein neuer Launcher für APGAS geschrieben, welcher *ssh* nutzte um auf anderen Knoten neue Places zu starten.

Hierbei musste zur Ausführung weitere Information in den Container gegeben werden, damit es möglich war verschiedene Places zu starten. Es musste dem APGAS-Container mitgeteilt werden, welcher Benutzer diesen Ausführt, damit bei der Erstellung neuer Places der Container sich mit den Benutzernamen des Benutzers und dessen *ssh-key's* dort anmelden konnte.

Als nächstes musste dem APGAS innerhalb des Containers mitgeteilt werden, in welchem Container-System es ausgeführt wurde, da verschiedene Containervirtualisierungssysteme verschiedene Aufrufe zum Starten benötigen.

Um anschließend den APGAS-Container zu starten, mussten Skripte zum starten der Container geschrieben werden, da APGAS mehrere Parameter zur Ausführung benötigt.

4 Implementierung

Dieses Kapitel erklärt, wie Singularity und UDocker verwendet wurden, um einen APGAS-Container zu erstellen. Es kann gleichzeitig als Anleitung gesehen werden, um einen APGAS-Container zu erstellen.

Zunächst werden die Anpassungen am APGAS-Quellcode erläutert. Anschließend werden die Schritte zum Erstellen des Containers dargestellt. Beide Containervirtualisierungssystemen benutzen hierbei ein Docker-Image als Basis für ihren Container. Jedoch unterscheiden beide Systeme sich in der Erstellung des Containers. Singularity erstellt eine Datei, welche zur Ausführung in das Singularity Programm gegeben wird. UDocker läd das Docker-Image in die UDocker-Engine hinein und der Container wird anschließend aus dieser Engine heraus gestartet.

Des Weiteren wird im folgenden unterschieden zwischen Standardvorgehen bei Erstellung eines Containers mit den jeweiligen Systemen und den Problemen die beim Erstellen eines APGAS-Containers überwunden werden mussten. Dabei wird zunächst erläutert, wie das Basis-Docker-Image erstellt wurde. Anschließend wird erläutert, wie die Containervirtualisierungssysteme Singularity und UDocker benutzt wurden.

4.1 Launcher

Bevor angefangen werden konnte, einen APGAS-Container zu erstellen, musste zunächst ein neuer Container-Launcher für APGAS geschrieben werden. Ein Launcher in APGAS ist der Teil des Programms, der neue Places auf verschiedenen Knoten startet. Hierbei werden aus dem Hostfile die verschiedenen verfügbaren Knoten ausgelesen und anschließend diesem Knoten ein Java-Befehl zum Starten von APGAS geschickt.

Das Problem, das hier auftrat war, dass entschieden werden musste, wie dieser Befehl an einen anderen Knoten geschickt wird, damit ein neuer Container gestartet wird. Da davon ausgegangen werden konnte, dass die, zum Ausführen eines Containers wichtigen Dateien bereits auf dem Knoten zur Verfügung standen, wurde das *ssh*-Protokoll benutzt, um den Befehl zum Starten eines Containers auf den Knoten zu schicken.

Es trat jedoch hier ein weiteres Problem auf. Der Container wusste nicht mit welchem Benutzernamen er sich auf dem anderen Knoten anmelden musste. Des Weiteren musste das jeweilige Containervirtualisierungssystem mit angegeben werden, da diese verschieden Syntax beim starten von Containern haben. Hierfür wurden neue Umgebungsvariable in APGAS hinzugefügt.

-Dapgas.container.clusterusername=<username> diese Variable gibt den Benutzernamens des ausführenden Benutzers an

-Dapgas.container.type= diese Variable gibt an, um welchen Container Typ es sich handelt. Es wird zwischen „udocker“ und „singularity“ unterschieden, da beide Containersysteme verschiedene Syntax zum Erstellen eines Containers besitzen

Zusätzlich hierzu musste weiterhin der *ssh*-Key des Benutzers mit in den Container gegeben werden, da ansonsten das Passwort des Benutzers für jeden neuen Place eingegeben werden muss.

4.2 Dockerimage

Um einen APGAS-Container mit Singularity oder UDocker zu erstellen, wird zunächst ein Basis Docker-Image benötigt. Ein Docker-Image benötigt immer ein anderes Image, das es als Grundlage benutzt. Um für APGAS einen Container zu erstellen, wurden zum einen eine Java-Version benötigt und zum anderen eine

SSH-Anwendung. Es wurde sich hier für ein Ubuntu Basis-Image entschieden, da für Ubuntu beides vorhanden war. Mit dem Befehl:

```
1 docker run ubuntu:19.04 bash -c "apt-get update && apt-get -y install
    openjdk-12-jdk && apt-get install ssh -y"
```

wurde ein ubuntu:19:04 Image aus dem Docker-Hub geladen und anschließend eine openjdk-12 Version für Java installiert, sowie eine *ssh*-Anwendung. Möchte man eine andere Version von Java verwenden muss der Befehl angepasst werden und die spezifische Version installiert werden. Anschließend wurde dieses neu erstellte Image mit dem Befehl:

```
1 docker commit <id> apgasimage
```

als *apgasimage* in Docker gespeichert und anschließend mit:

```
1 docker save apgasimage > apgasimage.tar
```

als .tar Datei gespeichert.

Nun konnte ein Dockerfile für den APGAS-Container geschrieben werden. Hierbei wurde das sogenannte Multibuilding von Docker verwendet. Die APGAS-Version, welche man benutzen möchte, muss sich in dem Ordner des Dockerfiles befinden und wird mithilfe des Befehls:

```
1 COPY ./apgas/ /apgascontainer/apgas
```

in den Compile-Container geladen und hier anschließend kompiliert.

Desweiteren musste sich der *ssh*-key des Benutzers in dem gleichen Ordner, wie das geschriebene Dockerfile befinden. Anschließend wurde das vorher erstellte *apgasimage* hinzugezogen und aus dem *Compile*-Container die kompilierte Version von APGAS dem Container hinzugefügt. Des Weiteren muss hier, wie bereits

erwähnt, der *ssh*-Key des Benutzers in den Container gegeben werden, damit dieser später auf dem Kassel-Cluster benutzt werden kann.

```
1 FROM apgasimage:latest
2 COPY --from=builder /apgascontainer /apgascontainer
3 COPY ./id_rsa /Root/.ssh/id_rsa
```

Nun befindet sich im Container an der Stelle */apgascontainer* die kompilierte Version des APGAS-Programms. Anschließend kann das Docker-Image erstellt werden. Hierzu muss man sich im Ordner des Dockerfiles befinden und den folgenden Befehl ausführen:

```
1 docker build -t apgascontainerdocker .
```

Nachdem das Image erstellt wurde befindet es sich im Docker-Daemon, also in Docker geladen und bereit, um aus ihm einen Container zu erstellen. Im Falle von Singularity ist es jedoch lediglich wichtig, dass dieses Image in Docker geladen ist, um hieraus einen Singularity-Container zu erstellen. Um auch die Vorbereitungen für UDocker abzuschließen muss das Docker-Image mit dem Befehl:

```
1 docker image save apgascontainerdocker >
    WORKSPACE/container/apgascontainerimage.tar
```

als .tar Datei gespeichert werden um später von UDocker verwendet zu werden.

4.3 Singularity-Container

Um einen Singularity-Container aus einem Docker-Image zu erstellen, muss zunächst eine Definitions-Datei (.def) angelegt werden. In dieser Datei wird angegeben, wo sich das Basis-Image für den Singularity-Container befindet. Im Falle des APGAS-Images befindet sich dieses geladen im Docker-Daemon. Hier wird also in der Definitions-Datei angegeben:

```
1 Bootstrap: docker-daemon
2 From: apgascontainerdocker:latest
```

Anschließend muss ein runscript angegeben werden. Dieses runscript beinhaltet alle Befehle, die ausgeführt werden sollen, wenn der Container gestartet wird. Im Falle des APGAS-Containers werden hier die verschiedenen Java-Befehle angegeben, die ausgeführt werden. Hierzu jedoch in Kapitel 4.5.1 mehr, da hier die Ausführung des APGAS-Containers auf dem Kassel-Cluster beschrieben wird. Um anschließend den Singularity-Container zu erstellen muss der Befehl:

```
1 sudo singularity build apgassingularity.sif apgascontainerdocker.def
```

ausgeführt werden. *apgassingularity.sif* gibt hierbei den Namen der Datei an, in welcher der Container schließlich gespeichert wird, und aus welchem er anschließend gestartet werden kann. *apgassingularity.def* gibt den Namen der Definitionsdatei für den Singularity-Container an.

Hier fällt auf, dass der Container mit einem sudo Befehl erstellt werden muss, weshalb es nicht möglich ist, den Container auf dem Cluster zu erstellen, sondern es von Nöten ist, dass der Container auf dem eigenen Computer erstellt werden und anschließend an den Cluster transferiert werden muss.

4.4 UDocker-Container

Im Vergleich zu einem Singularity-Container kann der UDocker-Container bereits auf dem Cluster erstellt werden, da hierfür keine Root-Rechte benötigt werden. Um einen UDocker-Container zu erstellen, muss lediglich ein Docker-Image in das UDocker-Tool geladen werden. Hierfür wurde das APGAS-Docker-Image bereits in einem .tar-file gespeichert. Nachdem das APGAS-Docker-Image mithilfe des Befehls:

```
1 udocker load -i ~/workspace/container/apgascontainerimage.tar
```

in das UDocker-Tool geladen worden ist, kann mithilfe des Befehls:

```
1 udocker create --name=apgasudocker apgascontainerdocker:latest
```

ein UDocker-Container erstellt werden. Hierbei gibt das Flag –name=apgasudocker den Namen des Containers an, damit dieser mit diesem Namen angesprochen werden kann. Anschließend kann der APGAS-Container mit dem Befehl:

```
1 ./udocker run apgasudocker
```

gestartet werden. Hierbei ist jedoch zu beachten, dass ,falls man Änderungen am Docker-Image vorgenommen hat und diesen neu laden möchte, das vorher geladene Image entfernen muss, da ansonsten Namenskonflikte auftreten könnten. Dies ist mit Hilfe des Befehls:

```
1 udocker rmi apgascontainerdocker:<version>
```

möglich. Anschließend muss der Conatiner entfernt werden. Die wird mit dem Befehl:

```
1 udocker rm apgasudocker
```

ausgeführt. Eine andere Variante um diese Konflikte zu vermeiden wäre es, das Versionen-Tag des Docker-Images der jeweiligen Version anzupassen.

4.5 Ausführung auf Cluster

Zur Ausführung der Beiden Containersysteme wurde die Docker-Version 18.09.5 zur Erstellung der Images genutzt. Hierbei wurde die *openjdk version „12.0.1“*

2019-04-16 zum kompilieren und ausführen benutzt. Die genutze Singularity Version ist 3.2.1-2 und die genutze Udocker Version ist 1.1.3. Die benutze APGAS-Version wurde aus dem git:

```
1 https://github.com/posnerj/PLM-APGAS beziehungsweise
2 https://github.com/posnerj/PLM-APGAS-Applications
```

mit Stand vom 17.April.2019 entnommen.

Zum ausführen der Container wurden zusätzlich die benötigten Knoten auf dem Cluster reserviert um eine Störungsfreie Ausführung zu garantieren.

4.5.1 Singularity

Um den Singularity-APGAS-Container auf dem Kassel-Cluster auszuführen, musste zunächst die Benennung des Containers an den Container-Launcher von APGAS angepasst werden. Des Weiteren musste der Installationsort von Singularity angegeben werden, da nicht davon auszugehen ist, dass ein Alias bei jedem Benutzer für Singularity gesetzt ist.

```
1 command.add(3, "singularity run apgassingularity.sif");
```

Außerdem musste das runscript des Singularity-APGAS-Containers so angepasst werden, dass es dynamisch ermöglicht verschiedene APGAS-Benchmarks zu starten. Um einen Singularity-APGAS-Container auf dem Kassel-Cluster mit einem bestimmten Benchmark zu starten ist folgender Befehl nötig:

```
1 singularity run apgassingularity.sif <BenchmarkName> <Places>
    <BenchmarkParameter>
```

Hierbei werden je nach eingegebenem Namen verschiedene Benchmarks gestartet, welche anschließend mit der angegebenen Anzahl von Places und dem gegebenen Parameter ausgeführt werden. Weitere Parameter wurden statisch im

Skript angegeben und müssen dort geändert werden beziehungsweise das Skript umgeschrieben werden um andere APGAS-Parameter dynamisch anzugeben.

Ein Problem, was hierbei auftrat, war, dass beim Aufruf eines neuen Places der Singularity Container neu gestartet wurde und gleichzeitig dessen runscript erneut aufgerufen wurde. Dies war jedoch nicht vorgesehen und das Skript springt in den Default-case und gab einen Fehler zurück. Hier wurde das runscript so geändert, dass bei einem Default-case lediglich der gegebene Parameter ausgeführt wird, bei einem neuen Aufruf auf einem Place also der Java-Befehl. Jedoch musste nun der Container-Launcher angepasst werden, damit der Java-Befehl als ein Parameter an das Runscript übergeben wird, da ansonsten lediglich *java* an den Container übergeben wurde. Aufgrund dieses Problems ist es nicht möglich eine Fehlermeldung bei Falschaufruf des Singularity-APGAS-Containers zu geben.

4.5.2 UDocker

Ein UDocker-APGAS-Container wird auch mithilfe von Skripten im Kassel-Cluster ausgeführt. Zunächst gibt es ein Script um einen UDocker-Container im udocker-Tool auf den Cluster zu laden. Hier wird lediglich, wie in Kapitel 4.4 erläutert, der vorherige UDocker-APGAS-Container und das Image entfernt und das neue Image geladen, zuzüglich Erstellung des Containers.

Das erstellte runscript für den UDocker-APGAS-Container, das nicht Teil des UDocker-Systems ist, funktioniert ähnlich wie das Singularity-Runscript. Die Syntax des Aufrufs ist ähnlich zu der von Singularity:

```
1 ./udockerRunscriptKasselCluster <BenchmarkName> <Places>
    <BenchmarkParameter>
```

Auch hier wird das Skript mit Namen des Benchmarks, der Anzahl der erwünschten Places und dem Parameter des Benchmarks ausgeführt.

Anschließend wird der UDocker-Container mit den gewünschten Parametern gestartet.

Jedoch wird das UDocker-Runscript nur einmal aufgerufen, da bei UDocker-Containern der Befehl direkt an den Container übergeben werden kann und kein Runscript erstellt werden muss. Der Container-Launcher startet also auf dem neuen Place einen Container mit dem jeweiligen Java-Befehl des auszuführenden Benchmarks.

5 Performancevergleich

In diesem Kapitel werden APGAS-Benchmarks der jeweiligen Container-Systeme mit der der JVM verglichen und anschließend ausgewertet. Zunächst wird erläutert, wie die jeweiligen Messungen ausgefallen sind und anschließend werden diese analysiert. Anschließend an die Analyse der Messungen werden die Ergebnisse zusammengefasst.

5.1 Messungen

Alle Messungen wurden auf dem Kassel-Cluster ausgeführt. Dieses ist in unterschiedliche Partitionen eingeteilt. Die Messungen wurden auf der Partition FB16 durchgeführt. Diese enthält 12 Doppelprozessorsysteme, die je zwei Intel Xeon 6-Kern Prozessoren mit Infinband-Vernetzung enthalten. Alle Programme wurden mit der *openjdk-12 Java-Version 12.0* kompiliert und ausgeführt. Es wurden jeweils zwei verschiedene Benchmarks für alle drei Systeme ausgemessen und jeder Benchmark wurde für jedes System pro Place 5 mal ausgeführt. Die beiden Containersysteme wurden, wie in Kapitel 4 erläutert, gestartet.

5.2 Auswertung

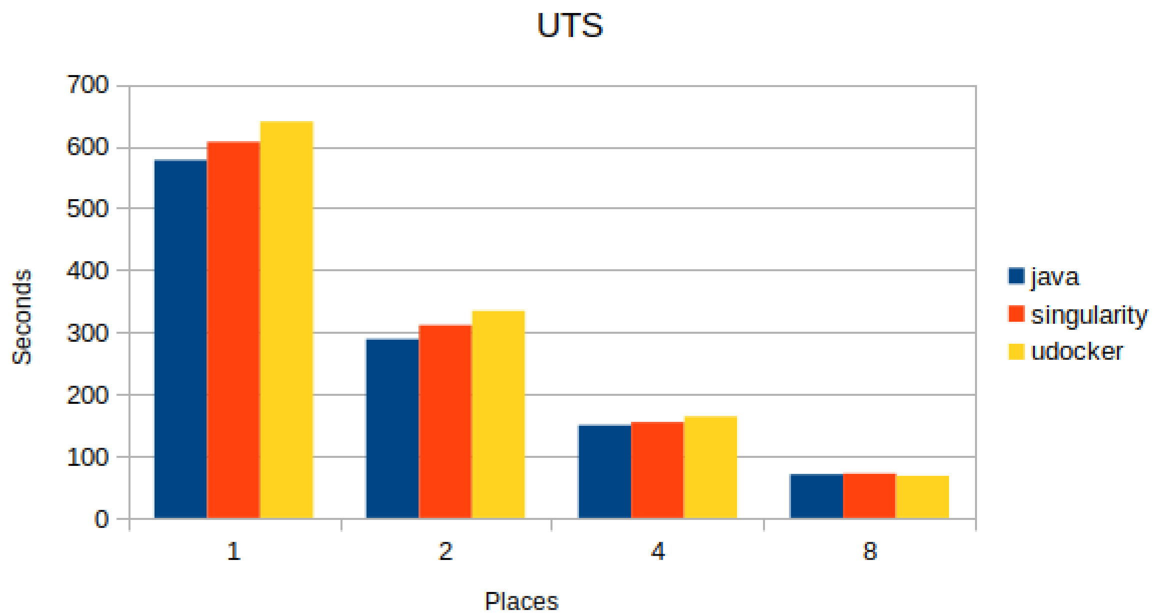

Abbildung 5.1: Vergleich UTS

Anzahl Places	Java	udocker	singularity
2	1,99989	1,91607	1,94867
4	3,86004	3,90809	3,92414
8	8,22011	9,44092	8,47795

Tabelle 5.1: Speedup UTS

Abbildung 5.1 zeigt die Ausführungszeiten des UTS-Benchmarks. Man erkennt, dass die Java-Ausführung auf 1, 2, 4 Places schneller ist, als die Ausführung der Container. Lediglich auf 8 Places ist die Java-Ausführung langsamer als die Ausführung des Udocker-Containers.

Bei Vergleich der beiden Container-Ausführungen erkennt man, dass die Ausführung des Singularity-Containers schneller auf 1, 2, 4 Places ist gegenüber der Ausführung des UDocker-Containers.

Den Speedup einer Ausführung berechnet man durch: Ausführung auf 1 Place / Ausführung X Place. Bei Betrachtung der Speedups des UTS-Benchmarks fällt auf, dass diese Werte auf 2 Places gegen 2 approximieren beziehungsweise bei 4 Places gegen 4 approximieren. Dies wird durch das Ahmdalsche Gesetz[**Amhdahl**] unterstützt, welches besagt, dass die Beschleunigung einer parallelen Ausführung eines Programmes nie vollständig parallelisiert werden kann, da es immer Teile des Programmes gibt, welche Sequentiell ausgeführt werden müssen. Hierraus folgt das es nicht möglich sein kann, dass ein Programm bei N Places einen Speedup von N erreicht. Jedoch kann es bei Messungen vorkommen, dass der Speedup den erwarteten Wert übersteigt. Dies folgt kann auf verschiedene Faktoren bei Ausführung des Programms zurückzuführen sein. Des Weiteren wird dieses Phänomen bei genügend Messwerten verschwinden.

Betrachtet man jedoch die Ausführung auf 8 Places fällt auf, dass der Speedup aller Ausführungen 8 übersteigt. Bei der Udocker-Ausführung wird der erwartete Wert im Vergleich zu den anderen Ausführungen sogar weit überstiegen. Dies lässt darauf schließen, dass diese Ausführung nicht repräsentabel ist. Jedoch ist zu erkennen, dass sich die Ausführungszeiten der 3 Systeme bei 8 Places angenähert haben.

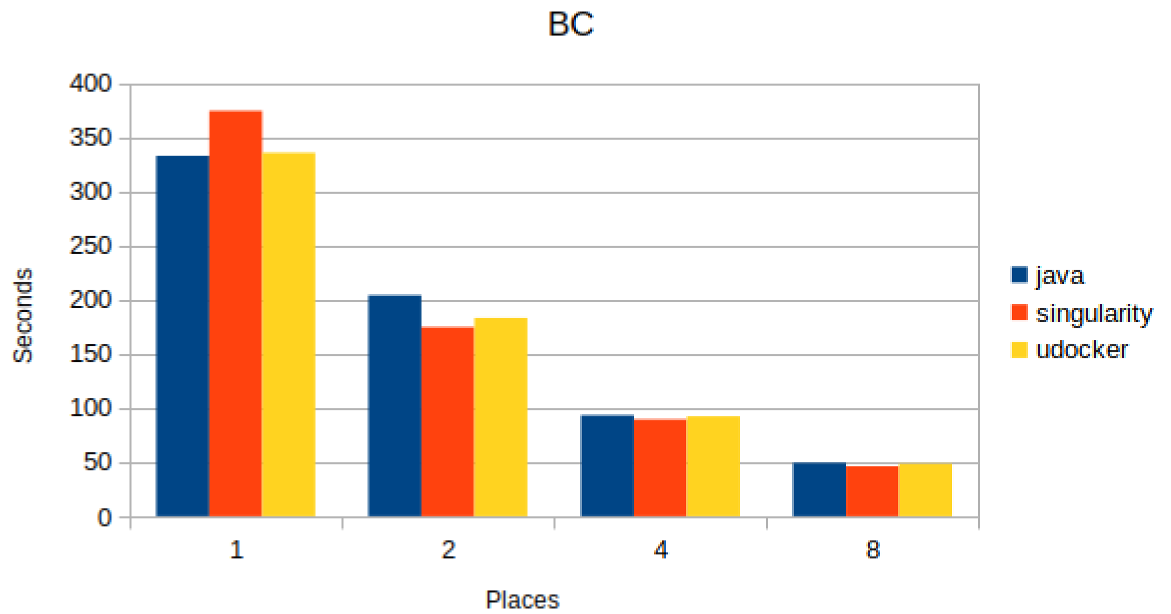

Abbildung 5.2: Vergleich BC

Anzahl Places	Java	udocker	singularity
2	1,6303	1,83422	2,14413
4	3,57142	3,63986	4,11883
8	6,74340	6,93767	8,13339

Tabelle 5.2: Speedup UTS

Abbildung 5.2 zeigt die Ausführungszeiten des BC-Benchmarks. Hier ist zu erkennen, dass wie im UTS-Benchmark die Ausführung des Singularity-Containers schneller ist, als die des UDocker-Container. Jedoch fällt hier auf, dass die Java-Ausführung langsamer ist, als die der beiden Container. Des Weiteren fällt auf, dass die Ausführung des Singularity-Containers auf 1 Place deutlich langsamer ist, als die anderen Ausführungen.

Betrachtet man den Speedup der einzelnen Ausführungen, so ist zu erkennen, dass der Speedup der Java-Ausführung deutlich niedriger ist, als

der der Container-Ausführungen. Des Weiteren fällt auf, dass der Speedup des Singularity-Containers konstant über dem erwarteten Wert liegt. Dies lässt sich jedoch auf eine unglückliche Ausführung des Singularity-Containers auf 1 Place zurückschließen, da dieser auf 1 Place deutlich langsamer, als die anderen Ausführungen war. Weiterhin ist auch hier zu erkennen, dass sie die Ausführungszeiten der 3 Systeme sich bei 8 Places annähern.

5.3 Zusammenfassung

Betrachtet man die Ausführung beider Benchmarks nebeneinander, so fällt auf, dass die Ausführung von APGAS ohne Container zwar beim BC-Benchmark langsamer als die der Container ist, hier jedoch auch der Speedup der Ausführung unter den erwarteten Werten ist. Auf der anderen Seite zeigt das UTS-Benchmark einen repräsentabeleren Wert der Java-Ausführung und hier ist gut zu sehen, dass die Ausführung schneller als die der Ausführung der Container ist. Lediglich die Ausführung des Udocker-Containers auf 8 Places ist hier schneller als die Java-Ausführung, dies lässt sich jedoch auf Schwankungen bei der Ausführung zurückführen.

Bei Vergleich der beiden Containervirtualisierungssysteme ist zu erkennen, dass die Ausführung des Singularity-Containers schneller ist als die des Udocker-Containers. Lediglich bei Ausführung des BC-Benchmarks und der Ausführung des UTS-Benchmarks auf 8 Places ist der Udocker-Container schneller. Dies lässt sich jedoch auch wie vorher erwähnt auf Schwankungen während der Ausführung zurückführen.

Es ist also Festzustellen, dass die Ausführung ohne Container schneller ist. Jedoch ist dieser Unterschied in der Ausführungszeit nicht sehr groß. Weiterhin ist die Ausführung des Singularity-Containers im Allgemeinen schneller als die Ausführung des UDocker-Containers. Bei Ausführung auf 8 Places sind alle Ausführungen ungefähr gleich schnell.

6 Fazit

In dieser Bachelorarbeit wurde, das APGAS-Programmiersystem mithilfe der Containervirtualisierungssysteme Singularity und Udocker in einer HPC-Umgebung ausgeführt. Hierfür wurde zunächst versucht das Docker-Virtualisierungssystem verwendet um damit einen APGAS-Container zu kreieren. Jedoch fiel auf, dass Docker nicht benutzt werden konnte, da hierfür Root-Rechte benötigt wurden, welche nicht in einer HPC-Umgebung vorhanden sind. Deswegen wurden zwei weitere Containervirtualisierungssysteme – Singularity und UDocker – verwendet, um einen APGAS-Container zu erstellen.

Nach anschließender Ausführung der Container auf dem Kassel-Cluster und Vergleich mit der Ausführung APGAS‘ ohne Container, wurde festgestellt, dass die Container im Vergleich zwar eine etwas langsamere Ausführungszeit besaßen, jedoch als Alternative zur herkömmlichen Ausführung genutzt werden können.

Die Technologie der Containervirtualisierung birgt den Vorteil, dass Entwicklungsstände leicht gespeichert und wieder aufgerufen werden können. Dies ist sinnvoll um etwaige Entwicklungsstände von APGAS zu speichern und wissenschaftlichen Arbeiten zuzufügen. Dies ermöglicht eine leichte Ausführung alter Programmstände, ohne den gesamten Quellcode neu zu kompilieren. Dank der leichten Ausführung ist es nun möglich alte APGAS-Benchmarks, die einer wissenschaftlichen Arbeit beigefügt sind, nachzuvollziehen.

Des Weiteren ist es möglich einen Container auf einer neuen HPC-Umgebung auszuführen. Hierfür wird lediglich benötigt, dass die Containervirtualisierungssysteme auf der HPC-Umgebung installiert sind.

Eine offene Aufgabe für künftige Arbeiten ist die Einbindung des APGAS-Containers in Verwaltungstools von HPC-Umgebungen, damit Container nicht manuell gestartet werden müssen.

Literaturverzeichnis

[1] Accessed: 2019-08-15. URL: https://docs.docker.com/images/Container%402x.png.

[2] Accessed: 2019-08-15. URL: https://docs.docker.com/images/VM%402x.png.

[3] B. Bashari Rad, H. Bhatti und M. Ahmadi. "An Introduction to Docker and Analysis of its Performance". In: *IJCSNS International Journal of Computer Science and Network Security* 173 (März 2017), S. 8.

[4] L. Benedicic, F. A. Cruz, A. Madonna und K. Mariotti. *Portable, high-performance containers for HPC*. 2017. eprint: arXiv:1704.03383.

[5] M. Eder. "Hypervisor- vs. Container-based Virtualization". eng. In: (2016). DOI: 10.2313/net-2016-07-1_01. URL: http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf.

[6] J. Gomes, E. Bagnaschi, I. Campos, M. David, L. Alves, J. Martins, J. Pina, A. López-García und P. Orviz. *Enabling rootless Linux Containers in multi-user environments: The udocker tool*. 2018. DOI: 10.1016/j.cpc.2018.05.021.

[7] D. M. Jacobsen und R. S. Canon. *Contain This, Unleashing Docker for HPC*. 2015.

[8] G. M. Kurtzer, V. Sochat und M. W. Bauer. *Singularity: Scientific containers for mobility of compute*. 2017. DOI: 10.1371/journal.pone.0177459.

[9] J. Posner und C. Fohry. *Hybrid work stealing of locality-flexible and cancelable tasks for the APGAS library*. 2018. DOI: 10.1007/s11227-018-2234-8.

[10] O. Tardieu. *The APGAS Library: Resilient Parallel and Distributed Programming in Java 8*. 2015. doi: 10.1145/2771774.2771780.