Bachelorarbeit im Rahmen des Studiengangs
Scientific Programming

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 – Medizintechnik und Technomathematik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Entwicklung einer Container Image Verwaltung für HPC-Systeme

29. August 2019

Ruben Simons

Selbstständigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

....
Unterschrift

Diese Arbeit wurde betreut von:
1. Prüfer: Prof. Dr.-Ing. Andreas Terstegge
Fachhochschule Aachen
2. Prüfer: Benedikt von St. Vieth, B. Sc.
Forschungszentrum Jülich (JSC)

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziel der Arbeit . . . 2

2 Grundlagen **3**
2.1 Von physikalischen Servern zu Containern . . . 3
2.2 Virtualisierung mit Containern . . . 4
2.2.1 Docker . . . 4
2.2.2 Singularity . . . 5

3 Realisierung **7**
3.1 Anforderungsanalyse . . . 7
3.1.1 Anforderungen . . . 7
3.1.2 Ansätze für die Umsetzung bestimmter Problemstellungen . . . 8
3.2 Datenbankstruktur . . . 9
3.3 Implementierung . . . 11
3.3.1 Rechtemanagement . . . 11
3.3.2 Kommandozeileninterface . . . 13
3.3.3 Datenbankanbindung . . . 14
3.3.4 Templating . . . 15
3.3.5 Import von Dockerfiles . . . 17
3.3.6 Build-Prozess . . . 18

4 Benchmarks **23**
4.1 Ausgewählte Benchmarks . . . 23
4.2 Konfiguration des Hosts und des Containers . . . 24
4.3 Ergebnisse der einzelnen Benchmarks . . . 26
4.4 Gesamtergebnisse der Benchmarks . . . 31

5 Zusammenfassung und Ausblick **33**
5.1 Zusammenfassung . . . 33
5.2 Ausblick . . . 33

1 Einleitung

1.1 Motivation

Die Forschungszentrum Jülich GmbH (FZJ) ist eine der 19 Forschungseinrichtungen der Helmholtz Gemeinschaft und mit seinen knapp 6000 Mitarbeitern eine der größten Forschungseinrichtungen in Deutschland und Europa. Am Jülich Supercomputing Centre (JSC), einem Institut des FZJ, widmet man sich dem Forschungsbereich “Schlüsseltechnologien”, konkreter den Themen “Supercomputing und Big Data”. In der Abteilung “High-Performance Computing Systems” liegt der Schwerpunkt dabei auf der Beschaffung, Installation und dem Betrieb von High-Performance Computing (HPC) Systemen. Die durch den Betrieb zur Verfügung gestellte Rechenzeit wird dabei auf nationaler und internationaler Ebene Wissenschaftlern zur Verfügung gestellt, damit diese mit der Hilfe von Simulationen wissenschaftliche Fragestellungen lösen können.

Ein HPC-System ist ein auf mehrere dutzend Schränke verteilter Zusammenschluss von mehreren tausend Computer Systemen, die verbunden durch ein entsprechendes Hochgeschwindigkeitsnetzwerk gemeinsam und parallel an Problemstellungen arbeiten können. Zugänglich gemacht werden die Ressourcen durch ein Batch-System, das als Schnittstelle für den Nutzer dient und dessen Aufgaben, sogenannte Batch-Jobs, asynchron abarbeitet. Im JSC wird das Batch-System Slurm[Sch19] benutzt.

Die Software Umgebung wird durch die Wahl des Betriebssystems und die Wahl der Ressourcen-Verwaltung bestimmt, am JSC wird dabei auf eine Kombination aus CentOS (Betriebssystem), Slurm (Scheduling) und Easybuild (HPC Software wie z.B. MPI) gesetzt.

Da nicht jeder Software Wunsch von Nutzern im Rahmen der oben erwähnten Umgebung bereitgestellt werden kann, lag es in der Vergangenheit in der Verantwortung von Nutzern, die eigene Software und deren etwaige Abhängigkeiten herunterzuladen und zu kompilieren beziehungsweise auf den HPC-Systemen lauffähig zu bekommen.

Dabei entstehen häufig komplexe Abhängigkeitsgraphen, die eine spezielle Umgebung benötigen, um reibungsfrei zu kompilieren beziehungsweise zu funktionieren.

Bisher stehen diese beiden Möglichkeiten, um Software auf HPC-Systeme zu bringen, bereit:

1. Modifikation der eigenen Software, damit diese Softwarepakete nutzen kann, die durch das Betriebssystem oder durch Easybuild zur Verfügung gestellt werden.

2. Nutzung eines komplexen Build-Prozesses, welcher statt den unpassenden Abhängigkeiten des Betriebssystems andere Versionen dieser Abhängigkeiten selbst kompiliert und nutzt.

Mit dem kommerziellen Erfolg von Docker, beziehungsweise Containertechnologien generell, wurde von Nutzern vermehrt der Wunsch der Unterstützung von Containertechno-

logien auf den HPC-Systemen des JSC geäußert. Durch entsprechend extern bereitgestellte Images, die in der Container Umgebung gestartet werden, können Nutzer ihr eigenes Betriebssystem samt Software Abhängigkeiten "mitbringen". Das HPC-System muss das Image nur noch ausführen und die darin enthaltene Software funktioniert in der Regel unmittelbar.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Implementierung einer Image-Verwaltung, welche Container-Images zu einem bestimmten Zeitpunkt auf Basis von Definitionsdateien reproduzierbar erstellt und darüber hinaus Nutzern zur Verfügung stellt. Diese Definitionsdatei wird von dem Vertreter einer spezifischen Nutzergruppe zur Verfügung gestellt und durch die Administratoren der HPC-Systeme validiert.

Anschließend werden die Images gebaut und im System verfügbar gemacht. Auf diesem Weg soll eine Auswahl an bestmöglich ausführbaren Images für den Betrieb auf den HPC-Systemen angeboten werden. Prinzipiell besteht die Möglichkeit, dass jeder Nutzer sein eigenes Image erstellen und laden kann. Dies würde jedoch den Supportrahmen sprengen, da der Administrations- und Supportaufwand hier zu groß wird. Deshalb ist es nicht Ziel dieser Arbeit, dass jeder Nutzer sein eigenes Image in die Image Verwaltung laden kann.

Um sicherzustellen, dass nicht nur die Container Images bestmöglich funktionieren, sondern gleichzeitig auch Indikatoren für die Leistungsfähigkeit von Containern im HPC-Umfeld zu sammeln, wird auf Basis einer Auswahl üblicher HPC-Benchmarks die Überprüfung bestimmter Leistungsindikatoren vorgenommen. Dabei wird die Ausführung innerhalb eines Containers mit der Ausführung außerhalb verglichen.

2 Grundlagen

2.1 Von physikalischen Servern zu Containern

Im Bereich Hosting gibt es schon seit geraumer Zeit die Möglichkeit, Bare-Metal Server für einen begrenzten Zeitraum zu mieten. Dies hat für Nutzer den Vorteil, dass diese nicht für die physikalische Administration und die damit verbundene Infrastruktur zuständig sind, sondern einen fertig verbauten Server zur Verfügung gestellt bekommen. Damit geben sie einen Teil der Verantwortung für das Hosting an den entsprechenden Anbieter ab und erhalten zeitnah ein nutzbares System. Betriebssystem und Software liegen dabei in der Verantwortung des Nutzers.

Da ein einzelner Server eine Mindestgröße in einem Serverschrank belegt, ist er gerade für einen kurzen Nutzungszeitraum mit unverhältnismäßig hohem zeitlichem Aufwand für die Installation verbunden. Gleichzeitig ist bei zu kleinen Problemstellungen die Differenz zwischen verfügbarer und angefragter Rechenleistung sehr groß, man verschenkt Ressourcen und somit auch Geld. Aus diesem Grund haben Anbieter angefangen, komplette Server zu virtualisieren. Als prominentes Beispiel ist hier Amazon mit seinen AWS-Services wie etwa dem EC2 zu nennen[Ama19]. Auf der gleichen Hardware können so mehrere virtuelle Server parallel zur Verfügung gestellt werden. Das hat den Vorteil, dass die verfügbaren Ressourcen effizienter genutzt werden. Im Gegenzug steht dem einzelnen virtuellem Server nicht die komplette Leistung zur Verfügung, was aber in vielen Fällen auch nicht benötigt wird.

Das Problem bei der Komplettvirtualisierung ist jedoch, dass diese im Vergleich zu direkter Hardware deutlich weniger effizient ist und einen gewissen Overhead benötigt[Fel+15]. Hier läuft typischerweise ein Betriebssystem im Betriebssystem. Damit können in der gleichen Zeit, auf der gleichen Hardware, weniger Anfragen beantwortet beziehungsweise weniger Berechnungen durchgeführt werden.

Der Linux-Kernel bietet die Möglichkeit, Ressourcen ähnlich wie bei einer virtuellen Maschine zu kapseln. Bei einer klassischen virtuellen Maschine würde der gesamte Linux Kernel mithilfe eines sogenannten Hypervisors virtualisiert werden. Diesen Overhead können sich die Container Lösungen ersparen[KSV17], indem sie das im Linux Kernel integrierte Feature "cgroups" verwenden[Ros13][Ros14]. Damit müssen nur noch Ressourcen wie CPU, Netzwerk und Filesystem isoliert werden.

Da inzwischen auch große Cloudanbieter wie Amazon mit AWS oder Microsoft mit Azure die Möglichkeit bieten, neben virtuellen Maschinen auch Container zu nutzen, findet deren Anwendung zunehmend Verbreitung.

Inzwischen wird auch im HPC-Umfeld ein sogenanntes "Bring your own Image" gefordert, um starre Betriebssystem- (beispielsweise alte Versionen von Paketen in CentOS) und Software-Umgebungen (easybuild, spack) zu umgehen.

2.2 Virtualisierung mit Containern

2.2.1 Docker

Docker ist eine Software, die sich um den kompletten Lebenszyklus eines Containers kümmert. Dies umfasst das Bauen und Verwalten von Images, das Ausführen von Containern sowie deren Netzwerkanbindung. Dabei besteht Docker aus einem dauerhaft laufenden Service, welcher alle Container verwaltet, und einem Kommandozeilen-Interface, mit dem der Nutzer interagieren kann.

Die Nutzung von Docker hat sich im Umfeld der Anwendungsentwicklung und -bereitstellung etabliert, da es den Entwicklungsprozess beschleunigt hat und einige Hürden für das Entwickeln, Installieren und Betreiben von Software nimmt. Dies erklärt sich durch die Nutzung einiger Konzepte, die zu diesem Zeitpunkt in dieser Kombination noch nicht etabliert waren[Ber14]:

- Deklarative Konfigurationsdatei: Eine einfache Konfigurationsdatei, das sogenannte Dockerfile, beschreibt, wie genau ein Container-Image erstellt wird. Diese Dateien sind dabei häufig 10-40 Zeilen lang, sodass diese einfach zu erstellen sind. Diese Konfigurationsdatei bietet die Möglichkeit, Images reproduzierbar zu bauen und entweder lokal oder über eine zentrale Datenbank, eine sogenannte Registry, zur Verfügung zu stellen.

- Schnelle Buildgeschwindigkeit: Im Gegensatz zum Imaging von virtuellen Maschinen, wo neben der Installation des Betriebssystem üblicherweise noch weitere Schritte notwendig sind, um ein lauffähiges Image zu erstellen, sind Container schnell gebaut. Ein Container kann so teilweise in weniger als einer Minute startbereit sein. Damit ist ein sauberer Buildprozess mit kurzer Wartezeit möglich, bei dem für jede Änderung ein neues Image zur Verfügung gestellt wird.

- Schnelle Startgeschwindigkeit: Container benötigen keinen eigenen Kernel, der gestartet werden muss. Damit entfällt die typische Bootzeit, die bei einer VM benötigt wird. Ein Docker-Container kann so in kürzester Zeit, teils unter einer Sekunde, gestartet werden. Abhängigkeiten zu anderen Containern, um beispielsweise ein Datenbanksystem zur Verfügung zu haben, lassen sich in Sekunden abbilden.

- Universelle Einsetzbarkeit: Docker läuft auf allen etablierten Linux-Plattformen und mit Einschränkungen auf Windows und macOS. Damit ein bestimmtes Image auf dem Hostsystem funktioniert, muss dort lediglich der Docker Service lauffähig sein. Die benötigten Abhängigkeiten für eine Software müssen nur noch im Image selbst sichergestellt werden. Damit muss der Entwickler nur innerhalb seines Images sicherstellen, dass die richtigen Bibliotheken für seine Software auf dem System, auf dem diese Software laufen soll, in der richtigen Version vorhanden sind.

- Single Responsibility: Ein bestimmtes Docker-Image hat genau eine Aufgabe. Es muss sichergestellt werden, dass diese Aufgabe erfüllt wird. Dies erleichtert die Fehlersuche und hat dazu geführt, dass sich Microservice Architekturen leichter etablieren. Außerdem kann eine Software innerhalb eines Containers nicht (ungewollt) das

Betriebssystem außerhalb des Containers so modifizieren, dass dieses nicht mehr wie gewünscht funktioniert.

- Einfache horizontale Skalierbarkeit: Wenn ein einzelner Docker-Container nicht genug Ressourcen für eine Anwendung zur Verfügung stellt, kann ein weiterer, gleichartiger Container gestartet werden. Dies ist zum Beispiel bei Webanwendungen häufige Praxis, sodass hier eine beliebige Anzahl an Docker-Containern gestartet werden können, je nachdem, wie oft die Webanwendung aktuell aufgerufen wird.

Aus diesen Gründen hat Docker innerhalb der letzten Jahre an Popularität gewonnen[And15]. Gleichzeitig haben sich andere Technologien wie Podman[Inc19b] etabliert, bei denen Docker als Blaupause fungiert hat.

2.2.2 Singularity

Singularity ist eine Software, die, ähnlich wie Docker, Container erzeugen und verwalten kann. Der Fokus liegt dabei, anders als bei Docker, auf der Unterstützung im HPC-Umfeld. Dies schließt die Unterstützung von typischen HPC-Workflows ein, zu denen vor allem die Unterstützung von Hochgeschwindigkeitsnetzwerken, GPUs und Batch-Systemen gehören[KSB17].

Unterschiede zu Docker

Während der Grundgedanke von Singularity und Docker, nämlich Container zu erstellen und auszuführen, identisch ist, steht hinter beiden Programmen eine unterschiedliche Philosophie.

Der größte Unterschied ist dabei das Starten von Containern: Bei Docker werden Container durch einen sogenannten Docker-Service gestartet, welcher als Root-Prozess läuft. Singularity verzichtet auf ein solches Konzept und startet die Container direkt durch den Aufruf des Programms, zum Beispiel mit `singularity run`, unter Verwendung des setUID-Bits.

Die Herangehensweise von Docker ist im HPC-Umfeld aus diesen Gründen problematisch:

- Sicherheitslücken in Docker können zur Eskalation von Privilegien führen, da Container prinzipiell mit Root-Rechten gestartet werden. Solche Sicherheitslücken sind in der Vergangenheit mehrfach bekannt geworden, beispielsweise im Januar 2019. [Mah19] Die Nutzung von Hochgeschwindigkeitsnetzwerken und GPUs funktioniert nur in privilegierten Containern mit Root-Berechtigungen, sodass hier eine zusätzliche Angriffsfläche geschaffen wird.

  Da viele Nutzer eigenständig auf die HPC-Systeme zugreifen und auch innerhalb von Containern eigene Software starten können, ist es nicht möglich, diese Container und ihre Nutzung manuell auf Sicherheit zu überprüfen.

- Das Accounting wird erschwert. Der Job, der von einem Nutzer im Batchsystem eingereicht wird, wird durch den Root-Account ausgeführt, da der Docker Service

> Vaterprozess des Containers ist und nicht der Service des Batch-Systems, der üblicherweise für das Starten und Beenden von Nutzerapplikationen auf den Knoten eines HPC-Systems verantwortlich ist. Die Rechenzeit kann so nicht mehr dem Nutzer zugeordnet werden, der diese Rechenzeit verbraucht hat. Diese Zuordnung ist jedoch für den Betrieb und die faire Nutzung eines HPC-Systems essentiell.
>
> Zusätzlich zu diesem Problem wird es dem Batch System erschwert zu erkennen, wann ein Prozess fertiggestellt ist. Erst dann kann ein Knoten für die Verwendung durch andere Jobs freigegeben werden.

Hinzu kommt, dass die Unterstützung von typischen HPC-Workflows bei Docker, anders als bei Singularity, bisher nicht gegeben ist. Um diese vernünftig nutzen zu können, ist eine spezielle Anpassung der Container notwendig, was entweder in höherem Aufwand für die Nutzer oder die Systemadministratoren resultiert.

```
1  #!/bin/bash
2  #SBATCH --job-name=HPL
3  #SBATCH --nodes=1
4  #SBATCH --time=01:00:00
5  #SBATCH --output=/p/project/chpsadm/singularity/benchmarking/hpl/logs/hpl-%j.out
6
7  cd /p/project/chpsadm/singularity/benchmarking/hpl
8
9  echo "BAREMETAL RUN"
10 srun /p/project/chpsadm/singularity/benchmarking/hpl/xhpl
11 echo "SINGULARITY RUN"
12 # Load some modules to run singularity
13 srun singularity exec /p/fastdata/singularity/centos.sif
    ↪ /p/project/chpsadm/singularity/benchmarking/hpl/xhpl
```

Listing 2.1: Slurm Batchfile für die Nutzung des High Performance Linpack-Benchmarks mit und ohne Singularity

Außerdem ist Singularity leicht in das Batch-System Slurm integrierbar, sodass dieses für die Nutzung nicht zusätzlich angepasst werden muss. Nach dem Laden von Singularity ist die Submittierung von Batch-Jobs ohne weiteres möglich. In Listing 2.1 ist ein Beispiel für die Nutzung des Benchmarks High Performance Linpack (HPL) zu sehen. Die Nutzung wird später in Kapitel 4 näher erläutert. In Zeile 10 wird der HPL direkt auf der Maschine gestartet, während der Benchmark in Zeile 13 mit Singularity und einem CentOS-Image gestartet wird. Dies unterscheidet sich im Vergleich zur direkten Ausführung dadurch, dass Singularity zuerst den Container startet und in diesem dann der Benchmark ausgeführt wird.

Aus genannten Gründen wird die Image-Verwaltung, die im Rahmen dieser Arbeit entsteht, speziell für die Nutzung mit Singularity entwickelt.

3 Realisierung

3.1 Anforderungsanalyse

Für eine geeignete Realisierung ist eine Analyse der Anforderungen erforderlich. Diese werden im Folgenden näher erläutert:

3.1.1 Anforderungen

Unterstützung von Dockerfiles als Definitionsdatei für Images Die Verwaltung der Images soll Dockerfiles unterstützen, da sich dieses Format für viele Container Technologien etabliert hat. Als Beispiel ist hier Podman anzuführen, welches Red Hat 2019 veröffentlicht und in das Betriebssystem Red Hat Enterprise Linux 8 integriert hat[Inc19b]. Das heißt, dass Singularity Images erzeugt werden sollen, die durch ein Dockerfile definiert werden.

Das Dockerfile ist dabei ein definiertes Format, welches von Docker benutzt wird, um Container-Images zu erzeugen. Docker-Images sind jedoch nicht ohne Weiteres mit Singularity kompatibel, sodass die Unterstützung von Dockerfiles mit Hürden verbunden ist.

Overlay im Build-Prozess nutzen Die Verwaltung soll es ermöglichen, im Build-Prozess Overlays zu benutzen. Als Overlays werden hier zusätzliche Anweisungen für den Build-Prozess bezeichnet, die ausgeführt werden müssen und für jeden Build automatisch hinzugefügt werden, wie zum Beispiel die Installation von bestimmten Treibern. Dies ist notwendig, damit die individuelle Softwareumgebung des Host-Systems reibungslos im Container genutzt werden kann. Insbesondere sind Anpassungen notwendig, um Bibliotheken für das Hochgeschwindigkeitsnetzwerk im Container verfügbar zu machen.

Paralleler Build von Images Die Images, die mit einer Definitionsdatei definiert werden, sollen zu einem vollständigen Singularity Image gebaut werden. Dabei sollen alle Images gleichzeitig neu gebaut werden können, zum Beispiel nach einer Systemwartung, bei der Cluster Software aktualisiert wurde. Damit dieser Build auch bei einer großen Anzahl von Images in annehmbarer Zeit geschehen kann, soll der Build von mehreren Images parallel geschehen können.

Persistente Datenhaltung Alle Daten, die verwaltet werden, müssen persistent gespeichert werden. Das ist notwendig, damit die Daten auch bei mehrmaligem Starten der Anwendung stets in der aktuellsten Version verfügbar sind. Zu den Daten, die hier gespeichert werden müssen, gehören unter anderem:

- Alle Definitionen von Images in allen bisherigen Versionen

- Alle Overlays, die für bestimmte Images benutzt werden können
- Alle bisher gebauten Images sowie die Definition, mit der diese erstellt wurden
- Alle Logs zu bisherigen Builds von Images

Gleichzeitig ermöglicht die Persistenz Reproduzierbarkeit, sodass auftretende Fehler besser nachvollzogen werden können.

Kommandozeileninterface Das Programm soll sich sowohl für Administratoren als auch für Anwender über die Kommandozeile bedienen lassen. Daher wird ein sinnvolles Kommandozeileninterface benötigt, mit dem alle Funktionen nutzbar sind.

Übersicht über alle verfügbaren Images Das Programm soll eine Übersicht über alle verfügbaren Images geben können. Diese Übersicht soll nicht nur für Administratoren, sondern auch für Benutzer zur Verfügung stehen.

Rechtemanagement Damit Nutzer kein separates Tool benutzen müssen, um eine Übersicht über die Images zu erhalten, wird diese Funktionalität direkt in das Tool zur Verwaltung der Images integriert. Damit Nutzer jedoch nur bestimmte Funktionen nutzen können und z. B. keine Images bauen können, dürfen bestimmte Funktionen nur bestimmten Nutzern zur Verfügung gestellt werden. Dies wird über ein Rechtemanagement realisiert.

Konfigurierbarkeit Damit das Programm flexibel bleibt, ist die Konfiguration über eine Datei erforderlich. Hier soll beispielsweise der Dateipfad eingestellt werden können, unter dem alle Daten gespeichert werden, die von der Imageverwaltung erzeugt werden. Auch das Rechtemanagement wird über diese Konfiguration geschehen.

3.1.2 Ansätze für die Umsetzung bestimmter Problemstellungen

Konvertierung von Dockerfiles zu Singularity-Recipes Die Verwaltung der Container soll Dockerfiles unterstützen, da dies im Augenblick ein weit verbreiteter Standard ist. Da Singularity selbst jedoch keine Dockerfiles unterstützt, muss dafür eine Lösung in der Image Verwaltung geschaffen werden. Hierfür gibt es grundsätzlich zwei Vorgehensweisen:

1. Mithilfe von Docker werden Images erstellt. Innerhalb der Image Verwaltung muss eine Komponente entwickelt werden, welche diese Images in ein kompatibles Format für Singularity überführt.
2. Konvertieren der Dockerfiles in Singularity-Recipes. Anschließend werden die Images mit Singularity gebaut.

Die erste Möglichkeit hat den Nachteil, dass diese zwar bei einem definierten Build-Prozess reproduzierbar ist, allerdings muss der Build-Prozess hier aufwändiger gestaltet werden. Das liegt daran, dass das gebaute Image wieder entpackt und in ein neues Format überführt werden muss. Außerdem muss der Inhalt des Containers geändert werden, damit Singularity Umgebungsvariablen und das Startskript findet. Dies müsste auch innerhalb

des Buildprozesses geschehen. Zusätzlich gibt es eine Abhängigkeit zu Docker, welches zum Bauen von Images notwendig ist.

Aus diesen Gründen wird der zweite Lösungensansatz umgesetzt. Damit werden die Dockerfiles vor dem Build zu Singularity-Recipes konvertiert. Da der Konvertierungsprozess nicht von externen Tools abhängig ist und nur eine Definitionsdatei erstellt wird, sind Ursachen für eventuelle Fehler im Imaging Prozess leichter zu lokalisieren und zu beheben.

Ein Singularity-Recipe ist mit einem Dockerfile vergleichbar. Dieses beschreibt, mit welchen Schritten ein Image erstellt werden soll. Es stellt jedoch ein eigenes Format dar und ist damit nicht mit Dockerfiles kompatibel.

Nutzung einer relationalen Datenbank Um eine persistente Datenhaltung, wie in Abschnitt 3.1.1 gefordert, zu gewährleisten, bietet sich eine relationale Datenbank an. In dieser Datenbank können alle benötigten Daten sowie deren Relationen zueinander gespeichert werden. Damit kann für jeden Build eines Containers auf die entsprechende Definition verwiesen werden.

Für dieses Softwareprojekt wird SQLite als Datenbank genutzt, da diese Datenbank ohne Server auskommt, die Anzahl der verwalteten Images und daraus resultierend die Anzahl der Einträge überschaubar ist, und die Administration deutlich einfacher als bei einem Relational DataBase Management System (RDBMS) nach dem Client-Server-Prinzip ist. Da keine parallelen, schreibenden Datenbankzugriffe von mehreren Aktoren geplant sind, sollte an dieser Stelle kein Engpass entstehen.

Weil Datenbankzugriffe nicht direkt gemacht werden, sondern durch das Object Relational Mapping-Framework (ORM) SQLAlchemy geschehen, ist eine spätere Migration zu einer anderen Datenbank ohne größere Anpassungen möglich. Dies wird in Abschnitt 3.3.3 weitergehend erläutert.

Um sicherzustellen, dass bestehende, funktionierende Recipes nicht durch eine Modifikation verloren gehen, werden diese versioniert gespeichert. Deshalb muss bei der Realisierung beachtet werden, dass keine Definitionen von Recipes überschrieben werden, sondern lediglich neue Versionen erzeugt werden, sodass die früheren Versionen eines Container-Recipes immer noch in der Datenbank verfügbar sind.

3.2 Datenbankstruktur

Alle Daten, die persistent gespeichert werden sollen, werden mithilfe des ORM-Frameworks SQLAlchemy erstellt[Bay19]. Dies ermöglicht eine Beschreibung der Tabellen als Python-Klassen, sodass diese Klassen direkt im Code genutzt werden können. Das Framework stellt dabei eine Verbindung zwischen Tabellen in der Datenbank und Klassen im Python-Quellcode her. Jeder Eintrag einer Tabelle entspricht hier der Instanz einer Klasse. Gleichzeitig bietet SQLAlchemy die Anbindungen an unterschiedliche SQL-Datenbanken wie zum Beispiel sqlite3, MySQL, Microsoft SQL Server oder Oracle Database.

Die Datenbankstruktur, die nun näher erläutert wird, ist in Abb. 3.1 visualisiert.

Rechts ist hier die Tabelle `base_image` zu sehen, welche die Basis für ein oder mehrere Recipes darstellt. Dabei kann es sich um ein lokale Imagedatei oder um ein Image von einer Docker Registry handeln (z. B. `fedora:latest`). Dies wird in der Spalte `type`

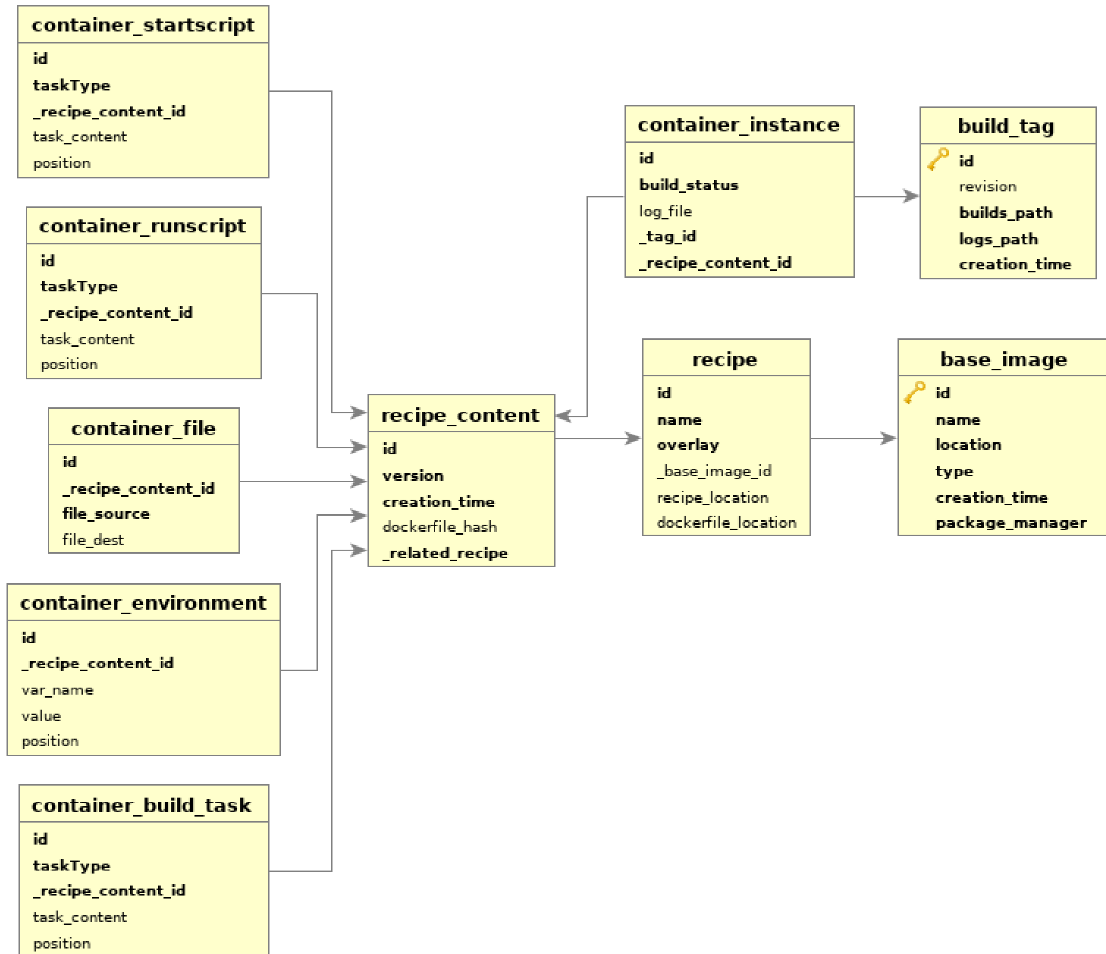

Abbildung 3.1: Datenbankstruktur

gespeichert. Weiterhin wird die Zeit, zu der das Base-Image erzeugt wird, sowie der Paketmanager der Distribution gespeichert. Der Paketmanager ist dabei für Operationen wie die Paketinstallation nützlich, da eine Paketinstallation dann universell für alle Images zur Verfügung gestellt werden kann. Außerdem wird ein Name gespeichert, um das Image identifizieren zu können.

In der Tabelle `recipe` werden alle verfügbaren Singularity-Recipes beschrieben, wobei die Spalte `_base_image_id` auf ein Basisimage aus der Tabelle `base_image` verweist. Dabei werden nur Metadaten beschrieben, da der tatsächliche Inhalt in der Tabelle `recipe_content` beschrieben wird. Dies wird benötigt um eine Versionierung zu realisieren, da sich der Inhalt eines Recipes ändern kann. Die älteren Versionen sollen in diesem Fall trotzdem noch verfügbar sein.

In der Spalte `overlay` der Tabelle `recipe` wird gespeichert, ob es sich um ein Overlay handelt. In diesem Fall kann dieses `recipe` als Basis für die Erstellung von weiteren Recipes dienen. Weitere Spalten beschreiben den Namen des Recipe, damit dieses identifiziert werden kann und noch zwei Dateipfade, für

- das Dockerfiles, aus dem das Recipe generiert wird
- den Speicherort, unter dem das Recipe bei Erzeugung exportiert werden soll

Der tatsächliche Inhalt wird durch `recipe_content` und die fünf Tabellen `container_runscript`, `container_build_task`, `container_file`, `container_environment` und `container_startscript` beschrieben.

Diese fünf Tabellen mit `container`-Präfix beschreiben Aufgaben, die in unterschiedlichen Schritten erledigt werden müssen. Bis auf die Tabelle `container_file` besitzen alle Tabellen das Attribut `position`, welches die Position der aktuellen Aufgabe angibt. Dieses wird benötigt, um bei Schritten, die abhängig voneinander sind, die Reihenfolge festzulegen. Beim Bauen eines Images ist das beispielsweise der Fall, da ein Paket erst installiert werden muss, um danach benutzt werden zu können. Außerdem enthalten diese Tabellen alle notwendigen Informationen, um alle Informationen zu einer Aufgabe zur Verfügung zu stellen, darunter den Aufgabentyp und den Inhalt.

Die Tabelle `recipe_content` enthält eine Versionsnummer, mit der die aktuelle Version identifiziert werden kann. Außerdem wird ein Hash für das Dockerfile gespeichert, aus dem dieses `recipe_content` erzeugt wurde. Damit kann bei erneutem Import des gleichen Dockerfiles die redundante Speicherung in der Datenbank verhindert werden.

Die Tabelle `container_instance` verweist sowohl auf die Tabelle `recipe_content` als auch auf die Tabelle `build_tag`. Hier wird ein tatsächlich zu bauendes Image dargestellt. Es wird der aktuelle Status festgehalten, also ob das Image später noch gebaut wird, aktuell gebaut wird, oder der Build fertiggestellt wurde, entweder erfolgreich oder fehlerhaft.

Alle zu einem Zeitpunkt gebauten Container werden im `build_tag` zusammengefasst. Hier wird auch der Erstellungszeitpunkt (`creation_time`) festgehalten, zu der dieses Tag erstellt wurde. Außerdem werden für jedes Tag jeweils zwei separate Ordner erstellt, in denen alle gebauten Images beziehungsweise die Logs zu deren Builds gespeichert werden. Anhand dieser Daten kann ein eindeutiger Name für das Buildtag erstellt werden, welches auch für die Images benutzt wird.

3.3 Implementierung

Die zuvor genannten Anforderungen wurden mithilfe der Programmiersprache Python in der Version 3.7 implementiert. Diese Wahl wurde aus persönlicher Präferenz, einer großen Auswahl an qualitativ hochwertigen Paketen und einer entsprechenden Verfügbarkeit für CentOS getroffen.

Die Implementierung wird im Folgenden nach Funktionen gegliedert vorgestellt.

3.3.1 Rechtemanagement

Da die Container Image Verwaltung sowohl für die Benutzer als auch für die Administratoren der HPC-Systeme zur Verfügung stehen soll, gibt es ein grundlegendes Rechtemanagement.

Die Rechteverwaltung sieht dabei drei unterschiedliche Arten von Nutzern vor, die hierarchisch aufgebaut sind:

1. User: Der normale, unprivilegierte Nutzer darf lediglich Kommandos ausführen, die keine Änderungen an der Datenbank vornehmen. Dazu gehört das Auflisten aller verfügbaren Container-Images. Wenn der Aufrufer des Programms zu keiner anderen Nutzergruppe gehört, ist dieser automatisch ein User.

2. Administrator: Ein Administrator darf alle Kommandos ausführen, die ein User ausführen kann. Zusätzlich kann der Administrator Kommandos ausführen, die die Datenbank ändern. Dazu gehört der Import und das automatische Updaten von Dockerfiles. Die Liste der Administratoren ist dabei konfigurierbar.

3. Root: Ein Root-Nutzer darf alle Kommandos ausführen, die verfügbar sind. Zusätzlich zu den Administrator-Kommandos ist hier das Bauen der Container zu sehen. Aus technischen Gründen ist das Bauen von Containern mit Singularity nur für Root-Nutzer möglich, weshalb Administratoren diese Funktion nicht nutzen können. Root-Nutzer ist ein Nutzer, wenn dieser die User-ID 0 hat, da dies auf UNIX-Systemen dem Superuser mit allen Privilegien entspricht.

Das Rechtemanagement fängt dabei bereits durch die Nutzung des Kommandozeileninterfaces ab, welche Kommandos zur Verfügung stehen. Kommandos, die für diesen nicht zur Verfügung stehen, können nicht gefunden werden. Das hat den Vorteil, dass diese Kommandos auch nicht in der Hilfe für das Programm auftauchen. In Listing 3.1 kann man sehen, dass das Kommando `build_all()` nur für Root-Nutzer zugänglich ist. Dies wird erreicht, indem die Definition des Kommandos nur erfolgt, wenn das `RightLevel` des Nutzers die Root-Berechtigung einschließt. Da es sich bei Python um eine dynamische Programmiersprache handelt und Funktionen hier zur Laufzeit definiert werden können, ist dieser Code zulässig.

Wie in Abschnitt 3.3.2 noch gezeigt wird, kann durch den Dekorator `@cli.command()` unter Anderem eine Hilfe erzeugt werden. Auch diese Hilfe wird bei unzureichenden Berechtigungen nicht erzeugt und damit auch nicht angezeigt.

```
1 if right_management.has_sufficient_permission(RightLevel.ROOT):
2     @cli.command()
3     def build_all():
4         image_build.build_all()
```

Listing 3.1: Rechtemanagement

Das Rechtemanagement stellt nicht sicher, dass die Datenbank vor unbefugten Zugriffen geschützt ist. Da es sich bei der Datenbank um eine SQLite-Datenbank handelt, ist dies nicht über eine Benutzerverwaltung im Datenbanksystem möglich. Die Daten in der Datenbank sind nicht vertraulich, sodass ein Lesezugriff für Nutzer unproblematisch ist. Da es sich bei der Datenbank im Fall von SQLite um eine einzelne Datei handelt, kann der Schreibzugriff durch unbefugte Nutzern verhindert werden, indem nur Administratoren Schreibrechte für diese Datei erhalten.

3.3.2 Kommandozeileninterface

Das Kommandozeileninterface wurde mit click erstellt[Pro19a]. Dies ist eine Python Bibliothek, welche für die einfache Erstellung von Argumentenparsern zuständig ist. Die Wahl ist hier nicht auf das in der Python-Standardbibliothek vorhandene argparse gefallen, da click einige Vorteile gegenüber argparse bietet.

Der größte Vorteil ist hier die einfache Umsetzung über ein deklaratives Modell gegenüber dem imperativen Modell von argparse.

Bei argparse wird zuerst ein Argumentenparser erstellt, bei dem unterschiedliche Optionen und Subkommandos definiert werden. Nachdem diese Optionen und Subkommandos definiert wurden, müssen diese zusätzlich noch manuell ausgewertet und entsprechend verschiedene Funktionen genutzt werden.

Bei click hingegen geschieht dies in einem Schritt, indem einzelne Methoden einen oder mehrere Dekoratoren erhalten.

```
1  @click.group()
2  @click.option('--debug', is_flag=True)
3  def cli(debug):
4      """Diese Anwendung ist zur automatisierten Verwaltung und zum
5      automatisierten Build von Singularity-Images zuständig"""
6      # Do setup stuff
7
8
9  @cli.command(name="import")
10 @click.argument("path")
11 @click.option('--recipe-name')
12 @click.option('--image-name')
13 def import_dockerfile(path, recipe_name, image_name):
14     """Importiert ein Dockerfile in die Datenbank und konvertiert dieses direkt in ein
       ↪ Singularity Recipe"""
15     # Do import stuff
16
17
18 if __name__ == '__main__':
19     cli()
```

Listing 3.2: Kommandozeileninterface mit click

In Listing 3.2 ist anhand eines vereinfachtes Beispiels aus dem Code dargestellt, wie click funktioniert. In Zeile 3 wird eine Methode definiert, welche beim Starten dieser Datei ausgeführt wird. Diese wird in Zeile 19 aufgerufen, falls die Datei direkt mit Python gestartet wurde.

Der Dekorator `@click.group()` sorgt dafür, dass die Methode `cli()` eine Gruppe ist, die weitere Subkommandos definieren kann. Damit kann dieser Methodenname auch als Dekorator verwendet werden, wie in Zeile 9 zu sehen.

Der Dekorator `@click.option('--debug', is_flag=True)` ist dafür zuständig, dass ein optionales Flag `--debug` beim Aufruf der Kommandozeile angegeben werden kann. Dieses Flag wird automatisch der Methode übergeben. Deshalb funktioniert auch der Aufruf von `cli()` in Zeile 19 ohne die Angabe des Parameters `debug`.

Der Docstring in Zeile 4-5 wird von Click für den Hilfetext des automatisch generierten `--help`-Befehls verwendet.

Die Methode `import_dockerfile()` ist ein Subkommando der Funktion `cli()`. Dies wird in Zeile 9 definiert. Hier wird zusätzlich noch ein Name angegeben, der für das Subkommando benutzt werden soll. Standardmäßig wird dafür der Name der Funktion genutzt, wobei Unterstriche durch Bindestriche ersetzt werden. In diesem Fall war es jedoch nicht möglich, die Funktion `import` zu nennen, da dies ein reserviertes Schlüsselwort in Python ist.

Die Funktion `import` ist, wie in Zeile 10-12 zu sehen, mit einem Argument und zwei Optionen versehen. Der Unterschied zwischen Argument und Option ist, dass ein Argument in jedem Fall benötigt wird, während eine Option optional angegeben werden kann, indem der Optionsname und dahinter der Wert dieser Option angegeben wird. Bei einem Argument wird lediglich ein Wert angegeben, ohne zusätzlich den Namen des Argumentes anzugeben. Da das Argument nicht optional ist, geschieht die Zuordnung des Wertes zu diesem Argument über die Position des Parameters. Argumente und Option werden der Funktion `import_dockerfile` übergeben.

Der Aufruf dieses Unterprogramms ist insgesamt durch

```
python dateiname.py import <dateipfad> [--image-name <image name>]
 ↪  [--recipe-name <recipe name>]}
```

möglich, wobei die eckigen Klammern optionale Angaben beinhalten.

3.3.3 Datenbankanbindung

Die Anbindung und Nutzung der Datenbank erfolgt durch das ORM-Framework SQLAlchemy. Das hat den Vorteil, dass alle Daten bei Datenbankzugriffen automatisch zu passenden Klassen umgewandelt werden und umgekehrt. Damit können Klassen geschrieben werden, die eine direkte Anbindung an die Datenbank haben. Außerdem ist SQLAlchemy für die Erstellung der Tabellen zuständig, sodass diese nicht extern vorbereitet werden müssen.

In Listing 3.3 ist ein Beispiel der Anbindung an die Datenbank zu sehen. Die Basisklasse `Base`, die in Zeile 1 erstellt wird, ist eine deklarative Basisklasse, die von SQLAlchemy erstellt wird. Jede Tabelle wird als Klasse, die von `Base` erbt, modelliert. Damit ist eine deklarative Konstruktion einer Klasse möglich. Dabei können Klassenattribute definiert werden, welche Spalten oder Beziehungen in der Tabelle darstellen. Die Struktur, die eine Klasse haben muss, wird dabei in der Dokumentation von SQLAlchemy vorgegeben und muss beachtet werden, damit Tabellen korrekt erstellt werden können.

In diesem Beispiel wird die Klasse `Recipe` erstellt. In der Datenbank wird, wie in Zeile 7 zu sehen, eine Tabelle mit dem Namen `recipe` erstellt.

Die Zeilen 9-15 definieren alle Spalten der Tabelle. Der Datentyp wird dabei als erstes Argument des Aufrufs von `Column` angegeben. Die Anmerkungen des Datentyps, wie beispielsweise bei `name: str` sind Type Hints, die in Python optional verfügbar sind. Da es sich bei Python um eine dynamische Programmiersprache handelt, muss man sich nicht an diese Datentypen halten. Allerdings helfen diese Type-Annotationen beim Programmieren, da die Entwicklungsumgebung durch diese Informationen den Typen der Parameter

```python
Base = sqlalchemy.ext.declarative.declarative_base()

class Recipe(Base):
    """
    This class represents the content of a recipe.
    """
    __tablename__ = "recipe"

    # columns
    id: int = Column(Integer, primary_key=True)
    name: str = Column(String, nullable=False, unique=True)
    overlay: bool = Column(Boolean, nullable=False, default=False)
    _base_image_id: int = Column(Integer, ForeignKey("base_image.id"))
    recipe_location: str = Column(String)
    dockerfile_location: str = Column(String)

    # relationships
    base_image: BaseImage = relationship("BaseImage")
    versions: List[RecipeContent] = relationship("RecipeContent", back_populates="recipe")

# Create all tables when something is imported from here
logging.debug("Creating Metadata for tables")
Base.metadata.create_all(engine)
```

Listing 3.3: Datenbankanbindung mit SQLAlchemy

bestimmen kann und der Programmierer weiß, welchen Typ dieser Parameter besitzt. Zusätzlich kann eine Spalte Attribute besitzen, wie zum Beispiel dass es sich um den Primary Key der Tabelle handelt, dass ein Argument kein `NULL` zulässt oder dass diese Spalte ein Verweis auf einen Fremdschlüssel darstellt.

Die Zeilen 18-19 definieren Beziehungen zu anderen Tabellen. Damit eine Beziehung zwischen zwei Tabellen möglich ist, muss eine dieser Tabellen einen Fremdschlüssel zu dieser Tabelle definieren. Das trifft in diesem Fall bei der zur Tabelle `BaseImage` zu, da `_base_image_id` einen Fremdschlüssel zu dieser Tabelle definiert. Für die Beziehung `versions` ist der Fremdschlüsselverweis in der Tabelle `RecipeContent` vorhanden. Durch die Rückbeziehung, die mit `back_populates="recipe"` definiert wird, kann diese Relation auch in der umgekehrten Richtung von der Klasse `RecipeContent` aus aufgerufen werden. Da ein Recipe mehrere Versionen haben kann, wird diese Beziehung als Liste dargestellt, während die Rückbeziehung nur auf dieses eine Recipe verweist.

In Zeile 23 wird eine Verbindung zur Datenbank hergestellt. Mit dieser Verbindung werden die gerade definierten Tabellen angelegt, falls diese noch nicht vorhanden sind. Realisiert wird dies über das Objekt `engine`, welche die allgemeinen Befehle von SQLAlchemy für ein spezifisches Datenbanksystem umsetzt, in diesem Fall für SQLite.

3.3.4 Templating

Die Anwendung nutzt Templating, um die Daten aus der Datenbank in tatsächliche Recipes umzuwandeln. Es existiert eine Schablone für ein Recipe, ein sogenanntes Template,

```
1  Bootstrap: {{ type }}
2  From: {{ location }}
3  {% if tasks|length > 0 %}
4  %post
5  {% for task in tasks %}    {{ task }}
6  {% endfor %}
7  {%- endif %}
8  {%- if envs|length > 0 %}
9  %environment
10 {% for env in envs %}    {{ env }}
11 {% endfor %}
12 {%- endif %}
13 {%- if files|length > 0 %}
14 %files
15 {% for file in files %}    {{ file }}
16 {% endfor %}
17 {%- endif %}
18 {%- if runscript|length > 0 %}
19 %runscript
20 {% for cmd in runscript %}    {{ cmd }}
21 {% endfor %}
22 %startscript
23 {% for cmd in startscript %}    {{ cmd }}
24 {% endfor %}
25 {%- endif %}
```

Listing 3.4: Jinja2-Template zur Erstellung eines Recipe

welches die Form dieses Recipes darstellt. Das Template ist mit Platzhaltern versehen, die durch Einträge aus der Datenbank ersetzt werden. Dabei wird das Python-Paktet Jinja2 für das Templating benutzt[Pro19b]. Jinja-Templates sind einzelne Textdateien, bei denen innerhalb von 2 geschweiften Klammern der Name eines Platzhalters steht, der dort eingefügt werden soll. Dabei unterstützt Jinja2 auch Schleifen oder Kontrollstrukturen. Diese stehen jeweils in einzelnen geschweiften Klammern. In Listing 3.4 ist beispielsweise zu sehen, dass in den ersten zwei Zeilen Platzhalter für den type und die location vorhanden sind.

Zwischen den Klammern, die mit `{%` und `%}` markiert werden, können Kommandos eingefügt werden, wie zum Beispiel ein if oder eine for-Loop. Dabei nutzt jinja2 eine eigene Syntax für erlaubte Kommandos. In dem Template sind mehrere Sektionen zu sehen, die mit `%` beginnen. Eine Sektion startet beispielsweise mit `%post` und ist vor dem `%environment` beendet. Diese Sektionen markieren einen bestimmten Schritt für den Build eines Singularity-Images, die `%post`-Sektion beschreibt beispielsweise Aufgaben, die mit dem Basisimage während des Buildprozesses durchgeführt werden müssen. In einer Schleife werden alle Objekte der unterschiedlichen Listen dargestellt, im Fall der `%post`-Sektion alle Elemente aus der Liste `tasks`, welche, wie nachfolgend zu sehen, dem Aufruf der Rendermethode mitgegeben wird. Wenn in der Liste `tasks` kein einziges Element vorhanden ist, wird die komplette Sektion inklusive `%post` nicht dargestellt. Dies wird durch das if in Zeile 3 erreicht, welches in Zeile 7 wieder geschlossen wird.

Um dieses Template mit Werten zu füllen, wird die Rendermethode "`def recipe_from_container(recipe_content: RecipeContent)`", die in Listing 3.5 gezeigt ist, aufgerufen. Der Parameter `recipe_content` beinhaltet dabei alle Referenzen zu den unterschiedlichen Parametern, die im Template benötigt werden. Alle Listen, die benötigt werden, werden aufsteigend nach ihrer Position sortiert.

Zum Rendern selbst wurde eine Instanz der Klasse `Renderer` erzeugt, welche das benötigte Template vom Filesystem beim Initialisieren lädt. Bei der Methode `def render_recipe(...)` werden nun die Parameter erwartet, die im Template benötigt werden. Als Standardwerte sind leere Tupel statt der Listen vorhanden, da durch diese iteriert werden kann und diese nicht veränderbar sind. Veränderbare Argumente als Standardwerte für Parameter können bei mehrfachem Aufruf einer Methode zu unerwartetem Verhalten führen.

3.3.5 Import von Dockerfiles

Um Singularity-Recipes zu erstellen, wird zunächst als Quelle ein Dockerfile benötigt, welches das Image beschreibt. Um dieses Image-Recipe importieren zu können, wird das Paket spython (Singularity Python) benutzt, welches auch die Umwandlung von Singularity- zu Dockerfiles beherrscht.

Wie in Abschnitt 3.3.2 erläutert wurde, kann über die Kommandozeile mit dem Kommando `import` ein Dockerfile importiert werden. Dies wird innerhalb der Image Verwaltung durch das Paket spython eingelesen. Alle Informationen, die in den unterschiedlichen Sektionen stehen, werden als Listen bereitgestellt. Exemplarisch ist in Listing 3.6 zu sehen, wie die Liste für die Build-Tasks erstellt wird.

In Zeile 10 ist zuerst zu sehen, wie ein DockerParser erstellt wird, welcher das Dockerfile einliest. Darauf werden alle Aufgaben, die in allen Overlays der Datenbank vorhanden sind, bereits eingeholt. Dies sind Aufgaben, die bereits vorher definiert wurden und für alle Recipes wichtig sind.

Das Basisimage wird anhand des Ursprungs in der Datenbank gesucht. Ist dieses Image bereits vorhanden, verweist das neue Recipe auf dieses vorhandene Basisimage, ansonsten wird ein neues Basisimage erstellt. Wurde bei Aufruf kein Name für das Basisimage als Option mitgegeben, wird der Nutzer in diesem Schritt nach einem Namen gefragt.

Danach wird überprüft, ob das Recipe mit dem entsprechenden Dockerfile bereits existiert. Wenn dies nicht der Fall ist, wird es neu erstellt.

Im nächsten Schritt wird ein Hash des Dockerfiles erzeugt. Damit kann sichergestellt werden, ob genau dieses Dockerfile bereits im letzten Schritt existiert hat. In diesem Fall haben sich die Daten nicht geändert und es wird kein neue Version als RecipeContent erzeugt, sondern die aktuelle Version zurückgegeben. Ansonsten werden alle Daten aus dem Dockerfile zuerst extrahiert und dann an die entsprechende Stelle im RecipeContent eingefügt, wobei hier erst die BuildTasks für das Overlay eingefügt werden. Zum Ende erhalten die BuildTasks, die nun in der richtigen Reihenfolge sind, die korrekte Position. Damit können diese später, in dieser Reihenfolge sortiert, abgerufen werden.

Nachdem alle Daten in die Datenbank importiert wurden, wird mithilfe des Templates in Abschnitt 3.3.4 eine Datei mit dem Singularity-Recipe erzeugt.

3.3.6 Build-Prozess

Um alle aktuellen Container neu zu bauen, kann von der Kommandozeile mit dem Befehl `build-all` der Build der Container angestoßen werden.

In Listing 3.7 wird der Ablauf eines Builds mit der Methode `def build_all()` gezeigt. Zuerst wird ein neuer Build-Tag, in Zeile 45 zu sehen, erstellt. Hier werden auch Ordner für den Build und für die Logs dieses Builds erstellt. Dieses Build-Tag wird direkt in die Datenbank übertragen, sodass der Build-Status später für jeden einzelnen Eintrag unabhängig geändert werden kann. Nun werden asynchron alle Container gebaut. Die Programmierung erfolgt hier asynchron und nicht parallel, weil eine echte Parallelisierung weder notwendig noch einfach mit Python zu realisieren ist[Bea10]. Der größte Aufwand beim Build ist nicht die Verwaltung, sondern das Ausführen des Singularity-Prozesses, der den eigentlichen Build durchführt. Dies kann nebenläufig mit mehreren Prozessen geschehen.

Für jedes Image, das in Zeile 36 mit dem aktuellen Buildtag gefunden wird, wird die Methode `build_simg()` asynchron aufgerufen. Damit kann bei jedem `await` die Imageverwaltung auf das Ergebnis warten und mit der Ausführung des restlichen Codes fortfahren.

In dieser Methode `build_simg()` wird der Ort des Recipes ermittelt, da dieser benötigt wird. Auch der Pfad des Images ist wichtig für den Build. Damit später noch nachvollziehbar ist, wie genau der Build abgelaufen ist, werden nun zwei Dateien zum Loggen geöffnet, von denen die eine Datei die Ausgabe nach stdout und die andere nach stderr beinhalten soll. Im Status in der Datenbank wird gespeichert, dass das Image gebaut wird, sodass dies später nachvollziehbar ist. Anschließend wird ein Prozess erstellt, welcher das aktuelle Singularity-Recipe baut und damit ein Imagefile erstellt.

Wenn der Prozess erstellt ist, kann es eine Weile dauern, bis der Build fertiggestellt wird. Auf die Fertigstellung des Prozesses wird gewartet.

Nachdem sich der Prozess beendet hat, werden die Logdateien gelöscht, falls sie leer sind. Dies passiert zum Beispiel, wenn ein Build fehlschlägt und nur ein Log nach stderr erfolgt. In der Datenbank wird auch gespeichert, ob der Build erfolgreich war oder ob dieser fehlgeschlagen ist. Bei erfolgreichem Build müssen zusätzlich noch die Unix-Dateirechte des neu gebauten Images angepasst werden, da diese Datei je nach Konfiguration von Singularity einem bestimmten Nutzer oder einer bestimmten Gruppe gehören müssen, um gestartet werden zu können.

```
class Renderer:
    def __init__(self):
        self._env = Environment(
            loader=FileSystemLoader(["../templates", "templates"])
        )
        self._recipe_template = self._env.get_template("recipe.tpl")

    def render_recipe(self, image_type: str, location: str, tasks: Iterable[str] = (),
                      envs: Iterable[str] = (), files: Iterable[str] = (),
                      startscript: Iterable[str] = (), runscript: Iterable[str] = ()):
        return self._recipe_template.render(
            type=image_type,
            tasks=tasks,
            location=location,
            envs=envs,
            files=files,
            startscript=startscript,
            runscript=runscript
        )

renderer = Renderer()

def recipe_from_container(recipe_content: RecipeContent):
    base_image = recipe_content.recipe.base_image
    if recipe_content.recipe.overlay:
        raise InvalidActionException("Overlay cannot be a recipe")

    # Get all needed lists and sort them
    tasks = sorted(recipe_content.container_build_tasks, key=_position)
    envs = sorted(recipe_content.container_environments, key=_position)
    files = recipe_content.container_files
    startscript = sorted(recipe_content.startscript, key=_position)
    runscript = sorted(recipe_content.runscript, key=_position)

    return renderer.render_recipe(
        image_type=_get_image_string(base_image.type),
        tasks=[_get_task_string(x) for x in tasks],
        location=recipe_content.recipe.base_image.location,
        envs=[_get_env_string(x) for x in envs],
        files=[_get_file_string(x) for x in files],
        startscript=[_get_task_string(x) for x in startscript],
        runscript=[_get_task_string(x) for x in runscript],
    )
```

Listing 3.5: Templating eines Recipe

```
from spython.main.parse.parsers import DockerParser

def install_to_list(install_list: List[str]) -> List[ContainerBuildTask]:
    """Create a list of ContainerBuildTask from the list of commands as string"""
    # Truncated

def recipe_from_dockerfile(dockerfile_location, session, recipe_name, base_image_name):
    current_task_position = 1

    recipe_from_parser = DockerParser(dockerfile_location).recipe
    overlay_tasks = create_overlay.get_all_overlay_tasks(session)
    build_tasks = install_to_list(recipe_from_parser.install)
    base_img_location = recipe_from_parser.fromHeader

    base_image = session.query(BaseImage).filter_by(location=base_img_location).first()
    if not base_image:
        if not base_image_name:
            base_image_name = input("Base Image Name: ")
        base_image = BaseImage(name=base_image_name,
                               location=base_img_location,
                               type=ImageType.DOCKER)
        session.add(base_image)
    recipe = create_or_get_recipe(session, recipe_name, base_image=base_image,
                                  dockerfile_location=dockerfile_location)

    # Get SHA256 from dockerfile, we do not want a new revision if nothing changed
    file_hash = get_file_hash(dockerfile_location)

    current_recipe = recipe.current_version(session)
    # Do not create a new dockerfile if SHA256 matches (so no change to dockerfile)
    if current_recipe and current_recipe.dockerfile_hash == file_hash:
        logging.debug("No changes to dockerfile detected, not creating a new version")
        return current_recipe

    # Add Everything to a new recipe content (truncated)
    current_recipe = recipe.new_recipe_content(session)
    current_recipe.dockerfile_hash = file_hash
    current_recipe.container_build_tasks.extend(overlay_tasks)
    current_recipe.container_build_tasks.extend(build_tasks)

    # Order the overlay_tasks and build_tasks that accumulated in current_recipe
    for task in current_recipe.container_build_tasks:
        task.position = current_task_position
        current_task_position += 1

    session.add(current_recipe)
    write_recipe_to_file(session, current_recipe.recipe)
    return recipe
```

Listing 3.6: Import eines Recipes (gekürzt)

```
async def build_simg(image_path, instance: ContainerInstance, session):
    session.add(instance)

    recipe_location = instance.related_content.recipe.recipe_location
    args = [
        "build",
        image_path,
        recipe_location
    ]

    logfilename = os.path.join(instance.built_with_tag.logs_path,
                               instance.get_name() + ".log")
    errfilename = os.path.join(instance.built_with_tag.logs_path,
                               instance.get_name() + ".err")
    with open(logfilename, "w") as logfile, open(errfilename, "w") as errfile:
        instance.build_status = BuildStatus.BUILDING
        session.commit()
        build_proc = await asyncio.create_subprocess_exec("singularity", *args,
         ↪  stdout=logfile, stderr=errfile)
        await build_proc.communicate()
    status = build_proc.returncode
    _delete_file_if_empty(logfilename)
    _delete_file_if_empty(errfilename)
    if status:
        instance.build_status = BuildStatus.FAILURE
        session.commit()
        return status
    # Do chown for image, to be owned by authorized person
    os.chown(image_path, pwd.getpwnam(Config.container_owner).pw_uid,
             grp.getgrnam(Config.container_group).gr_gid)
    instance.build_status = BuildStatus.SUCCESS
    session.commit()
    return status


async def build_for_buildtag(tag: BuildTag, session):
    images = queries.get_images_with_buildtag(tag)
    for image in images:
        instance = ContainerInstance(built_with_tag=tag,
         ↪  related_content=image.current_version(session))
        await build_simg(image_path=os.path.join(tag.builds_path, instance.get_name()),
                         instance=instance, session=session)


def build_all():
    session = Session()
    build_tag = create_build_tag(session)
    session.add(build_tag)
    session.commit()
    asyncio.run(build_for_buildtag(build_tag, session))
```

Listing 3.7: Build von Containern

4 Benchmarks

Damit die Container Image Verwaltung zu einem späteren Zeitpunkt in Produktion gehen kann, ist eine Evaluation der richtigen Funktionsweise und vor allem der Performance von Containern auf den HPC-Systemen notwendig. Durch eine Evaluierung soll festgestellt werden, ob der Einsatz von Containern sich negativ auf die Performance auswirkt. Da die HPC-Systeme, auf denen die Container laufen sollen, in der Anschaffung und im Betrieb sehr teuer sind, muss sichergestellt werden, dass die Hardware möglichst gut ausgenutzt wird.

Zur Evaluation wurden vier Benchmarks ausgewählt, die wichtige Komponenten von HPC-Systemen testen und damit einen guten Überblick darüber geben, wie gut die Performance innerhalb der Container ist. Dabei wurden diese vier Benchmarks jeweils 20 Mal ausgeführt, sodass einzelne Benchmarks mit Ausreißern die Auswertung nicht verzerren. Alle Benchmarks wurden auf dem HPC-System JURECA ausgeführt[Jül19].

In Listing 2.1 war bereits ein Beispielskript für die Submittierung des HPL Benchmarks zu sehen. Mithilfe des Kommandos `sbatch` wird der Benchmark an den Scheduler Slurm übergeben. Je nach Verfügbarkeit der Ressourcen wird der Job automatisch gestartet.

In Zeile 10 und Zeile 13 werden die Benchmarks einmal außerhalb und einmal innerhalb des Containers ausgeführt, beides im Rahmen desselben Jobs. Damit ist sichergestellt, dass der Benchmark jeweils auf dem oder den gleichen Knoten zu einer ähnlichen Zeit ausgeführt wird.

Die Ausführungsdauer eines Benchmarks beträgt jeweils ein paar Minuten. Da die Ausführung innerhalb des Containers unmittelbar nach der nativen Ausführung auf den gleichen Knoten durchgeführt wird, sind von eventuellen Ausreißern der Performance mit hoher Wahrscheinlichkeit beide Durchläufe betroffen. Diese Ausreißer können zum Beispiel auftreten, wenn zeitgleich andere Jobs auf das Filesystem zugegriffen wird, sodass hier nicht für jeden Knoten die vollständige IO-Bandbreite zur Verfügung steht, oder bei Temperaturschwankungen innerhalb eines Knotens. Auch Hardwaredefekte sind eine mögliche Ursache für ein unterschiedliches Ergebnis. Da nicht die absolute Performance gemessen werden soll, sondern lediglich die Unterschiede zwischen der Ausführung innerhalb und außerhalb eines Containers, ist eine konstant schlechtere Performance bei beiden Ausführungen hier nicht schlimm.

4.1 Ausgewählte Benchmarks

HPL Der High Performance Linpack-Benchmark (HPL) ist ein weit verbreiteter Benchmark zum Testen der reinen Rechenleistung von großen HPC-Systemen. Die Geschwindigkeit wird in FLOP/s (FLoating point OPerations per second) gemessen. Der HPL ist auch Grundlage für die Top500 Liste, in welcher die 500 schnellsten HPC-Systeme der Welt gelistet werden[pro19]. Die Problemstellung des HPL ist dabei das Lösen eines Li-

nearen Gleichungssystems mit dem Verfahren der LU-Zerlegung. Dieser Benchmark wird in diesem konkreten Testfall auf genau einem Knoten ausgeführt.

IOR Der IOR-Benchmark (Interleaved Or Random) ist ein Benchmark zum Test des Dateisystems. Dabei wird sowohl die Lese- als auch die Schreibgeschwindigkeit gemessen. Im JSC wird das parallele Dateisystem "General Parallel File System" (GPFS) von IBM benutzt, welches über ein Ethernet-Netzwerk angesprochen wird. Deshalb wird hier effektiv die Geschwindigkeit der Anbindung des Filesystems gemessen. Je nach aktueller Situation kann diese entweder durch das Filesystem oder das Netzwerk limitiert sein. Auch dieser Benchmark wird auf einem Knoten ausgeführt.

IMB Der Intel MPI Benchmark (IMB) ist ein Benchmark zum Testen des Hochgeschwindigkeitsnetzwerkes, welches alle Knoten miteninander verbindet. Dabei testet der IMB für unterschiedliche Nachrichtengrößen die Zeit, die für die Übertragung und den Nachrichtenaufbau benötigt wird. Dabei werden jeweils bestimmte Funktionen getestet, die der MPI-Standard zur Verfügung stellt. Dieser Benchmark wird auf 32 Knoten parallel ausgeführt.

LinkTest Der LinkTest ist ein Benchmark, der im JSC entwickelt wurde. Ähnlich wie der IMB testet dieser das Hochgeschwindigkeitsnetzwerk. Dabei werden PingPong-Tests zwischen allen möglichen Knotenpaaren durchgeführt. Es wird die Zeit gemessen, die eine Übertragung benötigt. Dieser Benchmark wird auf 32 Knoten parallel ausgeführt, wobei paarweise verschiedene Knoten miteinander kommunizieren. Ziel dieses Benchmarks ist es, defekte Komponenten im Hochgeschwindigkeitsnetzwerk zu identifizieren.

4.2 Konfiguration des Hosts und des Containers

Die Compute Knoten des Hosts JURECA besitzen jeweils zwei Mal die CPU Intel Xeon E5-2680 v3 mit jeweils 12 Kernen und 24 Threads. Als Hochgeschwindigkeitsnetzwerk wird Mellanox EDR InfiniBand eingesetzt[Jül19].

Die Tests innerhalb der Container wurden alle mit dem gleichen Image durchgeführt. Das Image wurde mit der Image Verwaltung gebaut. Das zugrundeliegende Dockerfile ist in Listing 4.1 zu sehen. Dieses Dockerfile wurde in die Datenbank der Imageverwaltung importiert, um dann ein Recipe für den Image Build Prozess zu erzeugen. Damit ist dieses Dockerfile Grundlage für das finale Image, mit dem die Tests durchgeführt wurden.

Das Dockerfile definiert in Zeile 1 CentOS als Basis-Image, genauer die CentOS-Version 7.6.1810. Da nicht anders angegeben, wird das Basis-Image über den Dockerhub bezogen[Inc19a].

Nach dem initialen Setup werden zwischen Zeile 3 und 8 lokale Dateien in den Container kopiert. Dies sind Daten, um die verfügbaren Repositories innerhalb des Container zu ändern, sodass der Container während des Setups Zugriff auf die internen Repositories des JSC hat.

Diese werden in Zeile 10 benötigt, da hier die standardmäßig genutzten Repositories deaktiviert werden, während die internen Repositories aktiviert bleiben. Danach werden

```
1  FROM centos:7.6.1810
2
3  COPY jsc-centos.repo /etc/yum.repos.d/jsc-centos.repo
4  COPY jsc-epel.repo /etc/yum.repos.d/jsc-epel.repo
5  COPY jsc-ofed.repo /etc/yum.repos.d/jsc-ofed.repo
6  COPY jsc.repo /etc/yum.repos.d/jsc.repo
7  COPY RPM-GPG-KEY-JURECA-JSC /etc/pki/rpm-gpg/RPM-GPG-KEY-JURECA-JSC
8  COPY RPM-GPG-KEY-JURECA-OFED /etc/pki/rpm-gpg/RPM-GPG-KEY-JURECA-OFED
9
10 RUN yum-config-manager --disable base,extras,updates,epel && sed -i
   ↪ 's/enabled=1/enabled=0/g' /etc/yum/pluginconf.d/fastestmirror.conf && yum makecache
   ↪ && yum install -y mlnx-ofa_kernel-4.5-OFED.4.5.1.0.1.1.gb4fdfac infiniband-diags
   ↪ libmlx5 ofed-scripts dapl libibcm ucx ibacm mft-4.11.0-103.x86_64 libibverbs-utils
   ↪ perftest && yum clean all && rm -rf /var/cache/yum
```

Listing 4.1: Dockerfile

einige Pakete installiert, die für die native Verwendung des Hochgeschwindigkeitsnetzwerkes zuständig sind. Ohne diese Pakete ist die Verwendung des Hochgeschwindigkeitsnetzwerkes nur unter Verwendung von Easybuild-Paketen möglich.

Für den tatsächlichen Build mit Singularity wurde das Image in ein Singularity-Recipe konvertiert, welches in Listing 4.2 zu sehen ist. Alle `COPY` Befehle aus dem Dockerfile wurden in diesem Recipe in die `%files` Sektion überführt. Der `RUN` Befehl ist nun in der `%post` Sektion in Zeile 7 vorzufinden. In dieser Sektion sind zwei zusätzliche Befehle zu finden, nämlich in Zeile 5 und 6. Hier werden innerhalb des Containers Pfade erstellt. Der Pfad `/usr/local/software` ist ein Pfad des Easybuild. Dieser Pfad ist notwendig, um Softwaremodule, die auf dem Hostsystem geladen wurden, auch den Containern zur Verfügung zu stellen. Dieser Pfad wird während der Ausführung mit einem Bindmount gemountet, sodass der Pfad innerhalb des Containers auf den gleichen Pfad gemapped wird. Die Nutzung dieser Softwarepakete ist optional, bietet Nutzern allerdings mehr Wahlmöglichkeiten für den Bezug benötigter Software.

Der zweite zusätzliche Pfad, der auch mit einem Bindmount in den Container gemapped wird, ist der Pfad `/p`. In diesem Pfad werden Dateisysteme aus dem GPFS verfügbar gemacht, auf denen die Nutzerdaten liegen.

Diese beiden zusätzlichen Pfade sind als Aufgaben des Standard-Overlay in der Datenbank eingespeichert. Dieses Overlay wird für alle Images in der Datenbank verwendet, da diese beiden Pfade in jedes Image gemountet werden sollen.

Das gewünschte Image wurde durch die Image Verwaltung mit dem Befehl `build-all` gebaut, nachdem es in die Datenbank aufgenommen wurde.

```
1  Bootstrap: docker
2  From: centos:7.6.1810
3
4  %post
5      mkdir -p /usr/local/software
6      mkdir -p /p
7      yum-config-manager --disable base,extras,updates,epel && sed -i
        ↪  's/enabled=1/enabled=0/g' /etc/yum/pluginconf.d/fastestmirror.conf && yum
        ↪  makecache && yum install -y mlnx-ofa_kernel-4.5-OFED.4.5.1.0.1.1.gb4fdfac
        ↪  infiniband-diags libmlx5 ofed-scripts dapl libibcm ucx ibacm
        ↪  mft-4.11.0-103.x86_64 libibverbs-utils perftest && yum clean all && rm -rf
        ↪  /var/cache/yum
8
9  %files
10     jsc-centos.repo /etc/yum.repos.d/jsc-centos.repo
11     jsc-epel.repo /etc/yum.repos.d/jsc-epel.repo
12     jsc-ofed.repo /etc/yum.repos.d/jsc-ofed.repo
13     jsc.repo /etc/yum.repos.d/jsc.repo
14     RPM-GPG-KEY-JURECA-JSC /etc/pki/rpm-gpg/RPM-GPG-KEY-JURECA-JSC
15     RPM-GPG-KEY-JURECA-OFED /etc/pki/rpm-gpg/RPM-GPG-KEY-JURECA-OFED
16
17 %runscript
18     exec /bin/sh -c "@$"
19
20 %startscript
21     exec /bin/sh -c "@$"
```

Listing 4.2: Recipe-File, welches durch die Image Verwaltung konvertiert wurde

4.3 Ergebnisse der einzelnen Benchmarks

HPL In Abb. 4.1 ist die erste Ausführung zu sehen, bei welcher der HPL-Benchmark innerhalb von Containern langsamer läuft als bei direkter Ausführung. Im Durchschnitt war die Ausführung hier etwa 1,47% langsamer. Auch wenn die Geschwindigkeit etwas schwankt, ist eine höhere Geschwindigkeit bei der Direktausführung des HPLs erkennbar und signifikant. Außerdem ist auch jeweils bei den individuellen Wertepaaren der erste Durchlauf ohne Container in jedem der 20 Fälle schneller als der zweite Durchlauf innerhalb des Containers.

Die naheliegende Vermutung hier ist eine geringere Geschwindigkeit aufgrund der Tatsache, dass die Ausführung innerhalb eines Containers geschieht. Da die Geschwindigkeit einer CPU jedoch auch durch die aktuellen Temperatur beeinflusst wird[1], könnte auch dies Auslöser für die geringere Geschwindigkeit der Ausführung innerhalb eines Containers sein.

Um dies zu überprüfen, wurden 20 weitere Jobs für den HPL submittiert. Im Unterschied zur letzten Ausführung wurde jedoch dieses Mal die Reihenfolge der Ausführung umgekehrt, sodass der HPL zuerst im Container und danach auf direkter Hardware ausgeführt wurde. Diese Ergebnisse sind in Abb. 4.2 zu sehen. Hier ist wieder die erste Ausführung schneller als die zweite Ausführung, obwohl die erste Ausführung innerhalb des Containers

[1] Bei Intel funktioniert der Turbo Boost, eine Technologie zum Hochtakten von Kernen, bei höheren Temperaturen nicht mehr

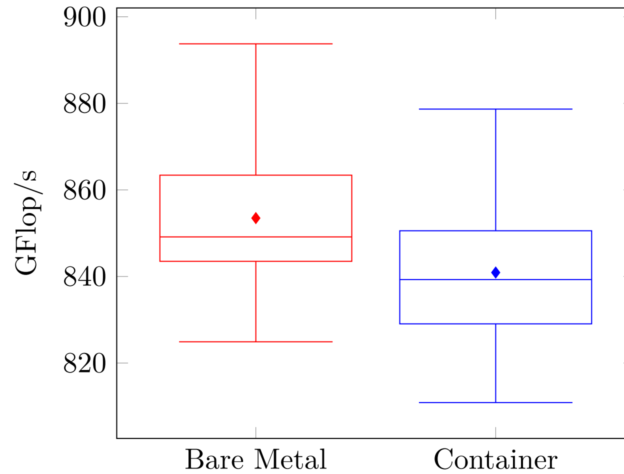

Abbildung 4.1: HPL Benchmark, direkte Ausführung zuerst

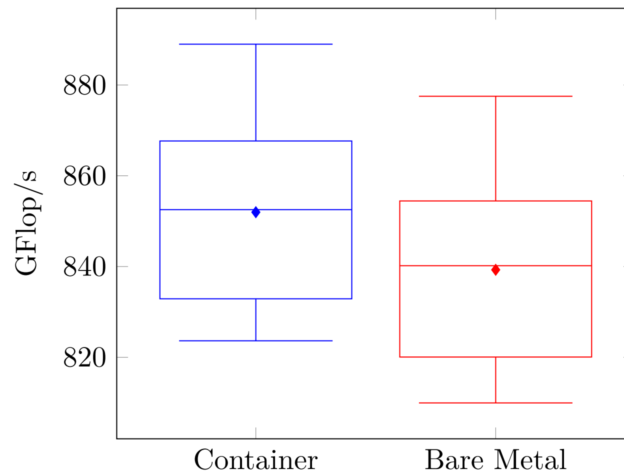

Abbildung 4.2: HPL Benchmark, Ausführung im Container zuerst

geschieht. Die Bare Metal Ausführung ist hier im Durchschnitt etwa 1,49% langsamer als die Ausführung im Container.

Der Unterschied lässt sich dadurch erklären, dass die Knoten zuerst darauf warten müssen, bis sie einen neuen Job erhalten. Dies ist aus logistischer Sicht mit Wartezeit für diesen Knoten verbunden, sodass die CPU in diesem Knoten in dieser Zeit nicht arbeitet und dadurch abkühlen kann. Sobald der Benchmark gestartet wird, kann die CPU mit einem erhöhten Takt arbeiten, der nicht dauerhaft gekühlt werden kann. Wenn eine bestimmte Temperatur erreicht wird, wird dieser Takt niedriger und die Geschwindigkeit damit auch geringer. Da die zweite Ausführung des Benchmarks direkt nach der ersten Ausführung stattfindet, beginnt der zweite Durchlauf mit einer höheren Temperatur innerhalb der CPU und entsprechend auch mit einem niedrigeren Takt.

Die 1,47% und 1,49% Verlangsamung sind in einer gleichen Größenordnung. Die 0,02% Differenz zwischen diesen beiden Werten sind durch statistisches Rauschen zu erklären.

Die Nutzung von Singularity Containern hat damit keine Einschränkungen bezüglich der Rechenleistung in Flop/s gegenüber einer direkten Ausführung von Programmen.

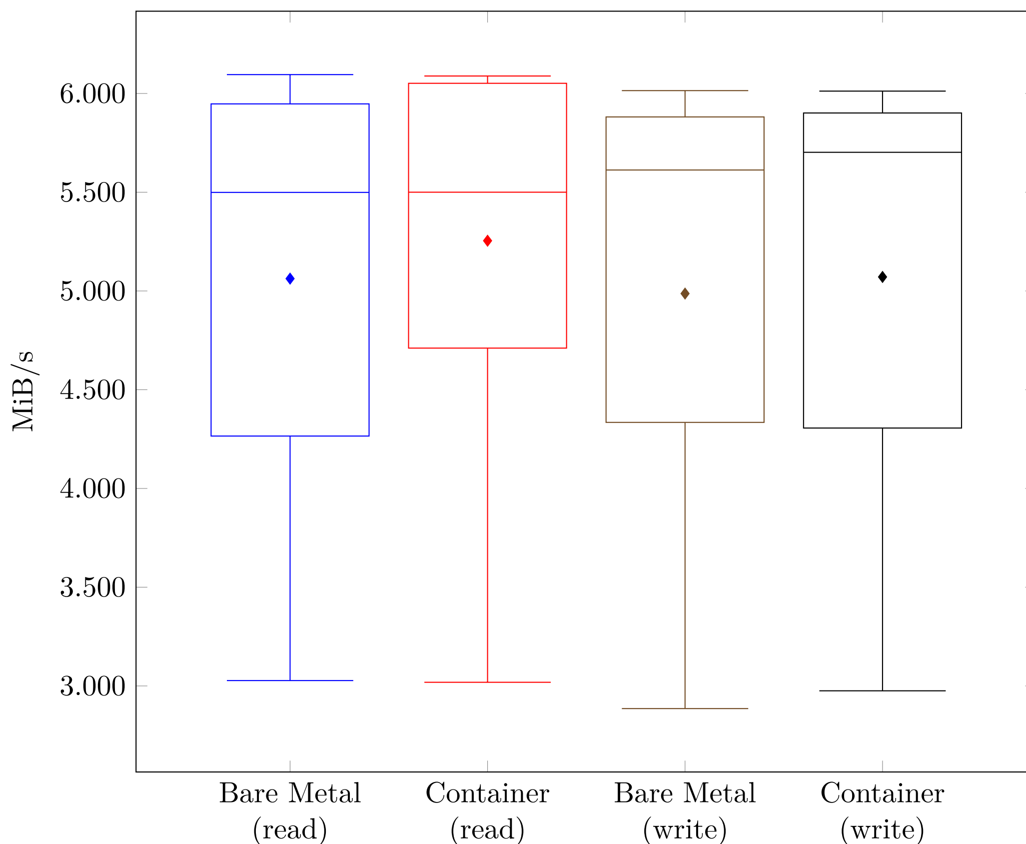

Abbildung 4.3: Ergebnisse des IOR Benchmark

IOR In Abb. 4.3 ist die Messung der Lese- und Schreibgeschwindigkeit auf dem Filesystem GPFS zu sehen. Hier werden sowohl beim Schreiben als auch beim Lesen Geschwindigkeiten zwischen 3GiB/s und 6GiB/s gemessen. Dabei sind beim Schreiben etwas niedrigere Maximal- und Minimalgeschwindigkeiten zu sehen als beim Lesen, allerdings sind beide Werte in der gleichen Größenordnung anzusiedeln.

Die Daten weisen eine sehr hohe Streuung auf und der Minimalwert liegt bei gerade mal 50% des Maximalwertes. Da das Filesystem auch von anderen Jobs benutzt wird, ist eine hohe Streuung zu erwarten. Bei der durchgeführten Messung ist die Geschwindigkeit sowohl beim Lesen als auch beim Schreiben innerhalb von Containern schneller. Diese Differenz ist auf die hohe Streuung zurückzuführen und statistisch nicht signifikant.

Insgesamt zeigen diese Daten, dass Lese- und Schreibvorgänge nicht durch Container limitiert werden. Hier liegt die Vermutung nahe, dass die Lese- und Schreibgeschwindigkeit vor allem durch andere Jobs limitiert wird und diese eine hohe Streuung in den Messergebnissen verursachen.

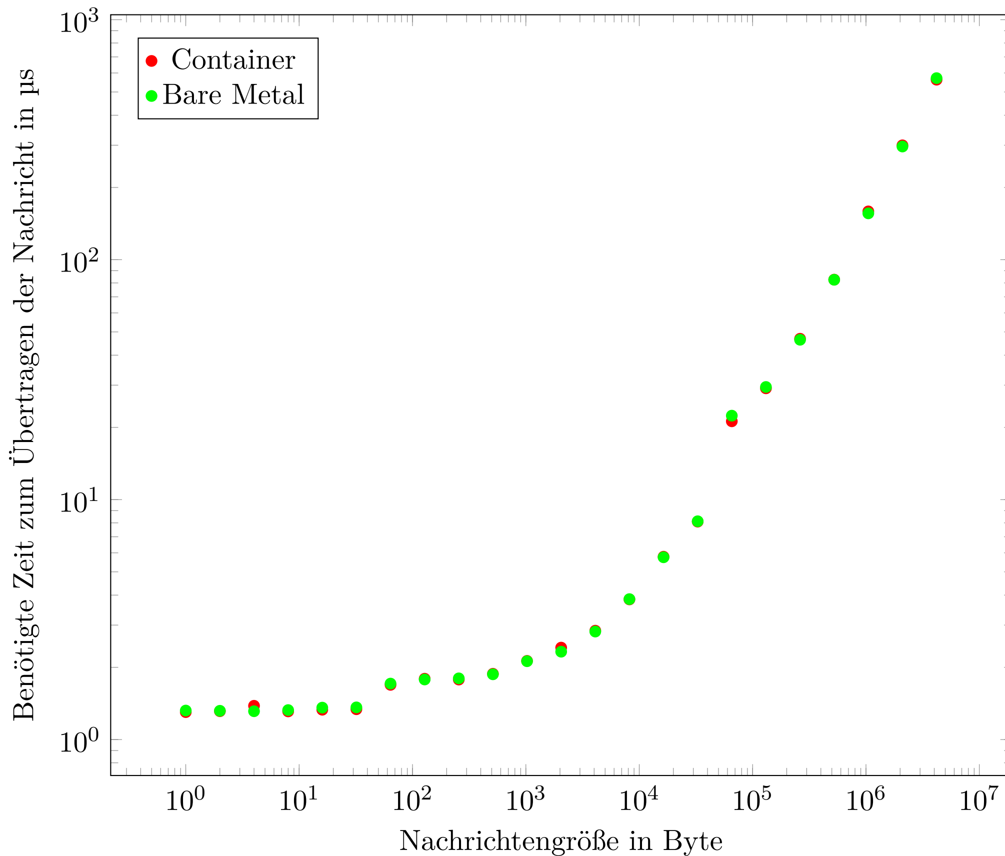

Abbildung 4.4: Durchschnittliche Zeit zum Übertragen einer Nachricht mit MPI

IMB Abb. 4.4 zeigt die Zeit, die für eine Übertragung einer Nachricht benötigt wird. Dabei sind beide Zeitskalen logarithmisch gewählt, damit die komplette Skala in einem guten Maßstab angesetzt werden kann. Die Nachrichtengrößen variieren dabei, wie auch in Abb. 4.5, zwischen 1 Byte und 4 MiB. Hier wurde die durchschnittlich benötigte Zeit aus 20 Durchläufen des Benchmarks ermittelt, wobei der IMB selbst bereits einige Iterationen an Messungen durchführt, bei den kleineren Nachrichtengrößen bis 32KiB jeweils 1.000 Iterationen.

Anhand dieses Benchmarks kann man sehen, dass sich alle Punkte mit den gleichen Nachrichtengrößen berühren, sodass die benötigte Zeit eines Sende- und Empfangsvorgangs innerhalb des Containers in etwa der benötigten Zeit ohne Nutzung von Containern entspricht.

Die Daten werden in Abb. 4.5 mit veränderter Metrik auf der y-Achse erneut dargestellt. Hier ist statt der Zeit die Übertragungsgeschwindigkeit zu sehen, die sich antiproportional zur Zeit verhält. Anhand dieser Grafik ist zu sehen, dass die Übertragungsgeschwindigkeit bei den ersten Verdoppelungen der Nachrichtengröße nahezu linear zunimmt. Dies ist zu erwarten, da hier die Latenz dominiert, also die Zeit, die für den Kommunikationsauf- und abbau benötigt wird, bevor überhaupt das erste Byte einer Nachricht übertragen werden kann. Bei großen Nachrichtengrößen spielt die Latenz nur noch eine untergeordnete Rolle.

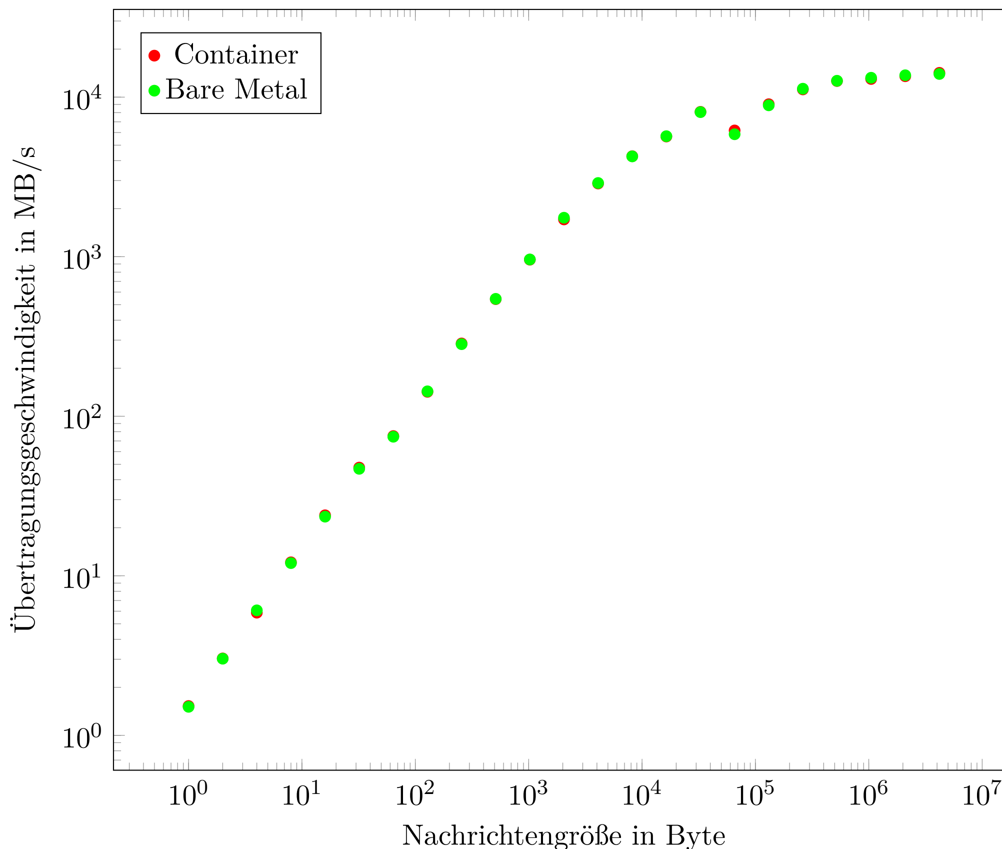

Abbildung 4.5: Durchschnittliche Übertragungsgeschwindigkeit mit MPI

Hier ist die maximale Übertragungsgeschwindigkeit, die über das Hochgeschwindigkeitsnetzwerk erreicht werden kann, der entscheidende Faktor, sodass die Kurve abflacht. Die Kurve approximiert etwa 14GB/s, die maximal erreichbar sind.

Bei einer Nachrichtengröße von 64 KiB ist die Übertragungsgeschwindigkeit niedriger als bei der Nachrichtengröße 32 KiB. Dafür ist vermutlich eine interne Änderung des Sende- und Empfangsalgorithmus, welchen MPI verwendet, verantwortlich. Diese Algorithmen werden je nach Nachrichtengröße ausgewählt.

Da in Abb. 4.5 die gleichen Daten wie in Abb. 4.4 visualisiert werden, liegen auch hier die Punktepaare sehr nah beieinander. Die Ausführung innerhalb des Containers ist teils schneller und teils langsamer als die Ausführung ohne Container, beide befinden sich aber in einer ähnlichen Größenordnung. Bis auf eine gewisse Streuung können die Ergebnisse aus diesen Benchmarks als gleich angesehen werden, sodass die Ausführung von MPI innerhalb der Container gleich schnell ist wie die Ausführung ohne Container.

LinkTest Aus dem Benchmark LinkTest wurden die zusammengefassten Ergebnisse extrahiert, wobei hier die durchschnittlich gemessene Latenz im gesamten Testdurchlauf als Metrik verwendet wurde. In Abb. 4.6 ist zu sehen, dass die durchschnittlichen Latenzen bei Ausführung innerhalb eines Containers und bei Ausführung ohne Container ähnlich

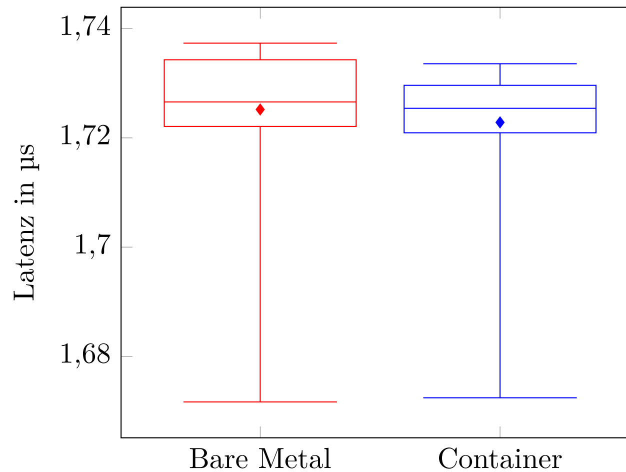

Abbildung 4.6: Durchschnittliche Latenz innerhalb eines einzelnen Durchlaufes des Benchmarks LinkTest

sind. Beide Boxplots stellen einen ähnlichen Bereich dar und sind vergleichbar. Die minimal bessere Latenz innerhalb des Containers ist durchschnittlich lediglich 0,13% niedriger. Diese niedrigere Latenz ist auf die Streuung der Werte zurückzuführen.

Damit ist auch bei diesem Benchmark keine niedrigere Geschwindigkeit für Container zu erkennen.

4.4 Gesamtergebnisse der Benchmarks

Die vier ausgewählten Benchmarks haben gezeigt, dass die Ausführung von Containern mit Singularity zu keiner Beeinträchtigung in der Ausführungsgeschwindigkeit zumindest dieser Benchmarks führt. Da diese Benchmarks wichtige Metriken für HPC-Systeme messen, sind diese auch für tatsächliche Anwendung aussagekräftig.

Um für eine spezifische Anwendung zu bestimmen, ob diese innerhalb eines Containers auch ähnlich schnell läuft wie in einer Umgebung ohne Container, muss ein spezifischer Benchmark mit Code dieser Anwendung durchgeführt werden, da Anwendungen individuelle Anforderungen an ihre Umgebung haben.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer Software zu der Problemstellung, Container Images in einer HPC-Umgebung automatisiert zu bauen und zu verwalten. Eine Software, die in der Lage ist, dies zu leisten, wurde im Rahmen dieser Arbeit entwickelt. Sie ermöglicht den automatisierten Build von Singularity Container Images, deren Build-Anweisungen durch ein Dockerfile zur Verfügung gestellt werden. Dadurch stellt die zusätzliche Pflege von Container Images nur einen geringen zusätzlichen Aufwand für die Administratoren von HPC-Systemen dar. Gleichzeitig wird zu jedem Zeitpunkt Reproduzierbarkeit gewährleistet, ein sehr wichtiger Aspekt im HPC. Basierend auf diesen Ergebnissen besitzt das Jülicher Supercomputing Centre mit dem hier entwickelten Softwaresystem nun die Grundelemente, um seinen HPC Benutzern die Möglichkeit von individualisierten Containern für ihre Computing Jobs anzubieten - ein Feature, das in der Vergangenheit schon häufig angefragt worden ist. Es werden aber noch kleinere Anpassungen notwendig sein, um die Image Verwaltung ins Jülicher HPC Ökosystem zu integrieren.

Die Arbeit beschäftigt sich auch damit, die Geschwindigkeit von Singularity-Containern auf dem HPC-System JURECA zu evaluieren und kommt zu dem Schluss, dass bei der Nutzung von Singularity Containern im HPC-Umfeld keine Leistungseinbußen festgestellt werden können, sodass diese im Produktionsbetrieb ohne Bedenken eingesetzt werden können. Damit können bei nativer Leistung die Einschränkungen, die durch die System Software geschaffen werden, umgangen werden.

5.2 Ausblick

Die Image Verwaltung, die im Rahmen dieser Arbeit erstellt wurde, ist zum aktuellen Zeitpunkt funktionsfähig. Es gibt jedoch an einigen Stellen noch Verbesserungsmöglichkeiten und neue Features, die in Zukunft implementiert werden können:

- Installationsmöglichkeit: Aktuell gibt es noch keine Möglichkeit, die Image Verwaltung mit einem standardisierten Verfahren zu installieren. Das ist jedoch für einen geordneten Produktionsbetrieb notwendig, damit die Verwaltung deterministisch installiert werden kann, sodass die Installation und das Update der Software schneller und vor allem kontrolliert geschieht.

- Unterstützung des Imports von Images: Aktuell werden noch keine Images unterstützt, die nicht durch die Image Verwaltung gebaut werden. In der Zukunft ist es wünschenswert, dass ein Teil der Images nicht durch die Imageverwaltung gebaut

wird, sondern bereits als fertige Images zur Verfügung stehen. Dieses soll in die Datenbank aufgenommen werden können. Die aktuelle Struktur der Datenbank kann dafür bereits benutzt werden, sodass lediglich eine Anpassung des Benutzerinterfaces notwendig ist.

- Tests: Das Testen der Software geschieht momentan manuell. Um die Software und neue Features schneller und zuverlässiger zu testen, ist es sinnvoll, diese automatisiert zu testen. Fehler können so schneller entdeckt werden.

- Flexiblere Nutzung von Overlays: Die Overlays, die eine spezielle Anpassung des Buildprozesses und damit an die benötigte Umgebung bieten, sind momentan noch statisch, sodass man diese nicht zur Laufzeit erzeugen kann. Außerdem werden aktuell alle Overlays, die in der Datenbank vorhanden sind, auf jedes neue Image angewandt. Eine sinnvolle Ergänzung hier wären beispielsweise Overlays, die nur für bestimmte Distributionen aktiv werden, sodass beispielsweise RPM-Pakete nur bei Images installiert werden, die das RPM-Format nativ unterstützen.

- Beantragung von neuen Images: Der Import von neuen Images ist aktuell nur durch Administratoren möglich. Dabei wird das Image durch den sogenannten Principle Investigator (PI) eines Projektes, der zwischen einer Nutzergruppe und dem JSC steht, submittiert. Wie genau die Submittierung abläuft, ist aktuell noch nicht geklärt. Eine Möglichkeit ist die Submittierung durch die Imageverwaltung, sodass die Administratoren diese Definition prüfen und gegebenenfalls zu einem späteren Zeitpunkt zu einem Image bauen können.

  Dieses Feature lässt sich auch gut mit den nächsten beiden Features verbinden.

- Verifizierungsdurchlauf: Sobald die Definition eines Images zum ersten Mal importiert oder geändert wurde, kann das Bauen des Images fehlschlagen. Hier wäre es interessant, zu überprüfen, ob der Build eines Images grundsätzlich möglich ist, ohne dafür alle Images neu bauen zu müssen und die neu gebauten Images in die Datenbank aufzunehmen. Da der Build ein komplexer Prozess ist, ist nicht mit sinnvollem Aufwand überprüfbar, ob sich dieses Image ohne Fehler bauen ließe, ohne diesen Build tatsächlich durchzuführen. Daher könnte ein Verifizierungsdurchlauf durchgeführt werden, der zurück gibt, ob das Image erfolgreich gebaut werden konnte. Nach dem Build wird dieses Image wieder gelöscht. Außerdem wird dieser Buildprozess nicht in die Datenbank aufgenommen, da er nur temporär ist.

  Wird dieses Feature mit der Integration einer Beantragung kombiniert, könnte so direkt bei der Beantragung dieser Verifizierungsdurchlauf geschehen, sodass die Beantragung bei einem Fehler in diesem Testdurchlauf abgebrochen werden kann.

- Änderung des Interfaces zu einer Weboberfläche: Aktuell wird die Image Verwaltung über die Kommandozeile gesteuert. Eine komfortablere Steuerung wäre über eine Weboberfläche möglich. Hier könnten alle verfügbaren Informationen auf einen Blick dargestellt werden. Auch eine Submittierung von neuen Images wäre über eine Weboberfläche einfacher zu bedienen.

- Weitere Schritte zur Bereitstellung der Images: Die Images werden aktuell gebaut und in einem Pfad zur Verfügung gestellt, in dem alle Images, die mit einem bestimmten Build-Tag gebaut werden, gefunden werden können. Da diese Strukturierung für Nutzer unübersichtlich sein könnte, sind weitere Schritte zur Bereitstellung sinnvoll. Dabei könnten die Images in einen neuen Pfad kopiert werden, in dem die jeweils aktuelle Version der Images zu finden ist. Dabei lassen sich auch unterschiedliche Pfade für verschiedene Images realisieren. Mit unterschiedlichen Pfaden lassen sich auch private Images realisieren, die nicht in einem öffentlichen Pfad zugreifbar sind, sondern nur für eine spezielle Nutzergruppe zur Verfügung steht. Dies vereinfacht zusätzlich das Accounting des genutzten Plattenspeichers, welches am JSC üblich ist.

- Versionshistorie darstellen: Die Versionshistorie eines Containers wird zwar in der Datenbank gespeichert, jedoch lässt sich diese nicht über die Image Verwaltung anzeigen. Sobald es zu Fehlern kommt, müssen die älteren Versionen manuell aus der Datenbank wiederhergestellt werden. Dies könnte verbessert werden, indem eine Implementierung für die Darstellung der Versionshistorie hinzugefügt wird.

Literaturverzeichnis

[Ama19] Inc. Amazon Web Services. *Amazon EC2*. 2019. URL: https://aws.amazon.com/ec2/ (besucht am 23. 08. 2019).

[And15] Charles Anderson. „Docker [software engineering]“. In: *IEEE Software* 32.3 (2015), S. 102–c3.

[Bay19] Michael Bayer. *SQLAlchemy - The Database Tookit for Python*. 2019. URL: https://sqlalchemy.org (besucht am 19. 08. 2019).

[Bea10] David Beazley. „Understanding the python gil“. In: *PyCON Python Conference. Atlanta, Georgia*. 2010.

[Ber14] David Bernstein. „Containers and cloud: From lxc to docker to kubernetes“. In: *IEEE Cloud Computing* 1.3 (2014), S. 81–84.

[Fel+15] W. Felter, A. Ferreira, R. Rajamony und J. Rubio. „An updated performance comparison of virtual machines and Linux containers“. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. März 2015, S. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[Inc19a] Docker Inc. *Docker Hub*. 2019. URL: https://hub.docker.com (besucht am 15. 08. 2019).

[Inc19b] Red Hat Inc. *8.0 release notes*. 2019. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/8.0_release_notes/index#notable_changes_to_containers (besucht am 19. 08. 2019).

[Jül19] FZ Jülich. *Konfiguration JURECA*. 2019. URL: https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html (besucht am 15. 08. 2019).

[KSB17] Gregory M. Kurtzer, Vanessa Sochat und Michael W. Bauer. „Singularity: Scientific containers for mobility of compute“. In: *PLOS ONE* 12.5 (Mai 2017), S. 1–20. DOI: 10.1371/journal.pone.0177459. URL: https://doi.org/10.1371/journal.pone.0177459.

[KSV17] Charalampos Gavriil Kominos, Nicolas Seyvet und Konstantinos Vandikas. „Bare-metal, virtual machines and containers in OpenStack“. In: *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE. 2017, S. 36–43.

[Mah19] Jan Mahn. *Sicherheitsforscher brechen aus Docker-Container aus*. 2019. URL: https://www.heise.de/security/meldung/Sicherheitsforscher-brechen-aus-Docker-Container-aus-4276108.html (besucht am 31. 07. 2019).

[Pro19a] The Pallets Project. *Click Documentation.* 2019. URL: `https://click.palletsprojects.com` (besucht am 01. 08. 2019).

[Pro19b] The Pallets Project. *Jinja2 Documentation.* 2019. URL: `https://jinja.palletsprojects.com` (besucht am 05. 08. 2019).

[pro19] The TOP500 project. *TOP500 Supercomputing Sites.* 2019. URL: `https://top500.org` (besucht am 19. 08. 2019).

[Ros13] Rami Rosen. „Resource management: Linux kernel namespaces and cgroups“. In: *Haifux, May* 186 (2013).

[Ros14] Rami Rosen. „Linux containers and the future cloud“. In: *Linux J* 240.4 (2014), S. 86–95.

[Sch19] SchedMD. *Slurm Documentation.* 2019. URL: `https://slurm.schedmd.com` (besucht am 07. 08. 2019).

Abbildungsverzeichnis

3.1 Datenbankstruktur 10

4.1 HPL Benchmark, direkte Ausführung zuerst 27
4.2 HPL Benchmark, Ausführung im Container zuerst 27
4.3 Ergebnisse des IOR Benchmark 28
4.4 Durchschnittliche Zeit zum Übertragen einer Nachricht mit MPI 29
4.5 Durchschnittliche Übertragungsgeschwindigkeit mit MPI 30
4.6 Durchschnittliche Latenz innerhalb eines einzelnen Durchlaufes des Benchmarks LinkTest 31

Listingsverzeichnis

2.1 Slurm Batchfile für die Nutzung des High Performance Linpack-Benchmarks mit und ohne Singularity 6

3.1 Rechtemanagement 12
3.2 Kommandozeileninterface mit click 13
3.3 Datenbankanbindung mit SQLAlchemy 15
3.4 Jinja2-Template zur Erstellung eines Recipe 16
3.5 Templating eines Recipe 19
3.6 Import eines Recipes (gekürzt) 20
3.7 Build von Containern 21

4.1 Dockerfile 25
4.2 Recipe-File, welches durch die Image Verwaltung konvertiert wurde 26