UNIKASSEL
VERSITÄT

Fachbereich Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik

Prototypische Entwicklung eines Schedulers für Malleable-Jobs

Bachelorarbeit

Vorgelegt im Fachbereich 16 – Elektrotechnik/Informatik
der Universität Kassel

Eingereicht von: Janek Bürger
Matrikelnummer: 35540944

Vorgelegt im: Fachgebiet Programmiersprachen/-methodik

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

Eingereicht am: 26. September 2023

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den nach der Prüfungsordnung der Universität Kassel zulässigen Hilfsmitteln angefertigt habe. Die verwendete Literatur ist im Literaturverzeichnis angegeben. Wörtlich oder sinngemäß übernommene Inhalte habe ich als solche kenntlich gemacht.

Kassel, 26. September 2023

Janek Bürger

Inhaltsverzeichnis

1 Einleitung

Ein Supercomputer ist meist ein Verbund aus mehreren Rechnern (*Nodes*), die zu einem *Cluster* zusammengeschlossen sind. Hauptziel ist die Berechnung großer, komplexer Rechenaufgaben (*Jobs*), die von einem einzelnen Computer nicht oder nur sehr langsam bewältigt werden können. Jobs werden nicht direkt ausgeführt sondern in Form von Jobs via Batch-Skript an den Supercomputer übergeben. Diese Skripte enthalten nicht nur Anweisungen zur Programmausführung, sondern auch Angaben zur erforderlichen Anzahl an Nodes sowie eine Zeitbeschränkung. Wann und mit welchen Nodes dieser Job auf dem Supercomputer gestartet wird, bestimmt der *Scheduler*. Bei Überschreitung der Zeitbeschränkung wird der Job vom Scheduler des Supercomputers beendet. In der Praxis werden die Jobs auf einem Supercomputer *statisch* ausgeführt und geschedulet. Dies bedeutet, dass statische Jobs (*Rigid-Jobs*) eine feste Anzahl von Nodes benötigen, um gestartet zu werden. Die Anzahl der Nodes darf während der Laufzeit des Jobs nicht geändert werden.
Dieser statische Ansatz kann jedoch dazu führen, dass Nodes eines Supercomputers ungenutzt bleiben. Angenommen, es gibt zwei Jobs auf einem Cluster mit 4 Nodes: Job 1 verwendet momentan 3 Nodes. Job 2 benötigt 2 Nodes zum Starten. Damit ist nur noch 1 Node auf dem Cluster frei, so dass Job 2 warten muss, bis Job 1 berechnet wurde.
Um die Ressourcen besser zu nutzen, könnte Job 1 um einen Node verkleinert werden, damit Job 2 früher starten kann, oder Job 1 könnte den freien Node erhalten, damit dieser seine Berechnung schneller erledigen kann. Jobs, die dazu in der Lage sind, werden als *elastische* Jobs bezeichnet.
Feitelson und Rudolph [1] haben elastischen Jobs wie folgt klassifiziert: Bei *Moldable-Jobs* legt der Scheduler die Anzahl der verwendeten Nodes beim Start des Jobs fest. *Malleable-Jobs* können vom Scheduler während der Laufzeit verkleinert (*shrink*) oder vergrößert (*expand*) werden. *Evolving-Jobs* können nach Absprache mit dem Scheduler selbstständig vergrößert oder verkleinert werden.
Um elastische Jobs zu verwenden, müssen mehrere Bedingungen erfüllt sein. Neben dem Job, der elastisch sein muss, muss auch der Scheduler elastische Jobs unterstützen. Außerdem muss der Scheduler über einen geeigneten Job-Scheduling-Algorithmus verfügen, der mit elastischen Jobs umgehen kann. Dies ist wichtig für Malleable- oder Evolving-Jobs, da diese mit dem Scheduler kommunizieren müssen.
Da aber die Umstellung auf Elastizität bei einem Standard-Scheduler wie Slurm, der Elastizität nur rudimentär unterstützt, auf einem Supercomputer extrem zeitaufwendig ist, beschäftigt sich diese Bachelorarbeit mit der Entwicklung einer modularen Experimentierplattform.

Diese Experimentierplattform soll auf einem Supercomputer ausführbar sein und mit den ihr zur Verfügung gestellten Nodes ihr eigenes Cluster bilden. Auf diesem Cluster läuft dann der entwickelte Scheduler der Experimentierplattform. Im weiteren Verlauf der Arbeit wird die modulare Experimentierplattform als *Scheduler* und der Scheduler des Supercomputers als *Super-Scheduler* bezeichnet. Hauptziel ist die Entwicklung eines funktionsfähigen modularen Schedulers, in den neue Job-Scheduling-Algorithmen mit geringem Aufwand integriert und evaluiert werden können. Ein weiteres Ziel ist die Implementierung und Evaluierung von existierenden Job-Scheduling-Algorithmen mit dem entwickelten Scheduler. Um diese Ziele angehen zu können, werden neben dem modularen Scheduler elastische Jobs und passende Job-Scheduling-Algorithmen benötigt. Für die Jobs werden die Bibliotheken von APGAS und GLB verwendet [2]. APGAS bietet eine Rigid-, Moldable- und Malleability-Unterstützung für den Job und GLB bietet ein Beispielprogramm für die Ausführung des Jobs (Rechenaufgabe). Aus diesem Grund unterstützt der Scheduler bereits die folgenden Job-Typen: Rigid, Moldable und Malleable. Zusätzlich wurden folgende aus der Literatur bekannten Job-Scheduling-Algorithmen integriert: *First-Come-First-Serve-*, *Backfilling-* und *Easy-Backfilling*. Diese unterstützen ausschließlich Rigid-Jobs und werden entsprechend angepasst, damit sie auch Moldable-Jobs unterstützen. Des Weiteren wird ein *Malleable-Basisalgorithmus* [3] integriert. Dieser Algorithmus bietet eine Malleability-Unterstützung. Somit kann dieser Algorithmus auf die folgenden Job-Typen angewendet werden: Rigid, Moldable und Malleable.
Mithilfe der vom Scheduler verwendeten Algorithmen, wurden Evaluationen auf dem Supercomputer der Universität Kassel durchgeführt. Dabei werden die Experimente mit einen unterschiedlichen Anteil von Rigid- zu Moldable-/Malleable-Jobs durchgeführt. Die Ergebnisse sind eine Bestätigung für die Funktionsfähigkeit des Schedulers und für die Möglichkeit, die implementierten Algorithmen mit echten Programmen testen zu können. Zusätzlich können die Ergebnisse der Experimente automatisch ausgewertet werden. Unter anderem können die Gesamtlaufzeit der Experimente oder die Systemauslastung der Experimente ermittelt werden. Dabei hatte der Malleable-Basisalgorithmus mit einem Malleable-Job-Anteil von 100% die schnellste Gesamtlaufzeit von 2169 Sekunden und damit einen Speedup von 22,39% (im Vergleich zu 100% Rigid-Job-Anteil). Im Vergleich dazu hatte der zweitschnellste Algorithmus (Backfilling) eine Gesamtlaufzeit von 2290 Sekunden, was einem Speedup von 12,91% entspricht.
Im zweiten Kapitel werden die genannten Job-Scheduling-Algorithmen und die Bibliotheken APGAS und GLB näher erläutert. Kapitel 3 geht auf die Konzepte der Arbeit ein. Im vierten Kapitel werden der Aufbau und die Funktionsweise des Schedulers beschrieben. Die Ergebnisse der praktischen Experimente sind in Kapitel 5 dargestellt. Kapitel 6 fasst die Ergebnisse der Arbeit zusammen.

2 Grundlagen

2.1 Job-Scheduling Algorithmen

Im Folgenden werden die Algorithmen First-Come-First-Serve, Backfilling, Easy-Backfilling und Malleable erläutert.

2.1.1 First-Come-First-Serve-Algorithmus (FCFS)

Der FCFS-Algorithmus ist ein aus der Literatur bekannter Algorithmus, welcher auch unter dem Namen First-in-First-out (FiFo) bekannt ist. Dabei werden Jobs nach der Reihenfolge bearbeitet, in der sie eintreffen [4, 5].
Für ein Beispiel siehe die Abbildung 2.1. Für diese Abbildung wurde der Algorithmus mit 5 Nodes und 5 abzuarbeitenden Jobs ausgeführt. Die Jobs wurden kurz nacheinander an den FCFS-Algorithmus übergeben. Wie der Name schon sagt, wird der Job ausgeführt, der zuerst im Algorithmus abgegeben wurde. Somit wird Job 1 mit 2 Nodes gestartet. Damit sind noch 3 Nodes frei auf den ein Job gestartet werden kann. Da Job 2 jedoch 4 Nodes benötigt, muss gewartet werden, bis Job 1 fertig ist. Damit ist Job 2 das erste Element in der Liste der zu wartenden Jobs (*q_head*). Sobald Job 1 fertig ist, wird Job 2 gestartet und wieder geprüft, ob ein weiterer Job gestartet werden kann oder ob der nächste Job aus *q_head* warten muss.

2.1.2 Backfilling-Algorithmus

Der Backfilling-Algorithmus [6] startet wie der FCFS-Algorithmus. Sobald jedoch *q_head* nicht mehr gestartet werden kann, wird in der Liste offener Jobs nach passenden Jobs gesucht, die gestartet werden können. Wird ein geeigneter Kandidat gefunden, wird dieser gestartet.
Gut zu sehen in der Abbildung 2.2 in der die Jobs 3, 4 und 5 vorgezogen wurden. Wie zu sehen ist, ist dies jedoch nicht fair gegenüber Job 2, da dieser als zweiter abgegeben und als letzter berechnet wurde. Es kann also vorkommen, dass Jobs bei diesem Algorithmus immer wieder nach hinten verschoben werden und somit länger auf ihren Start warten müssen.

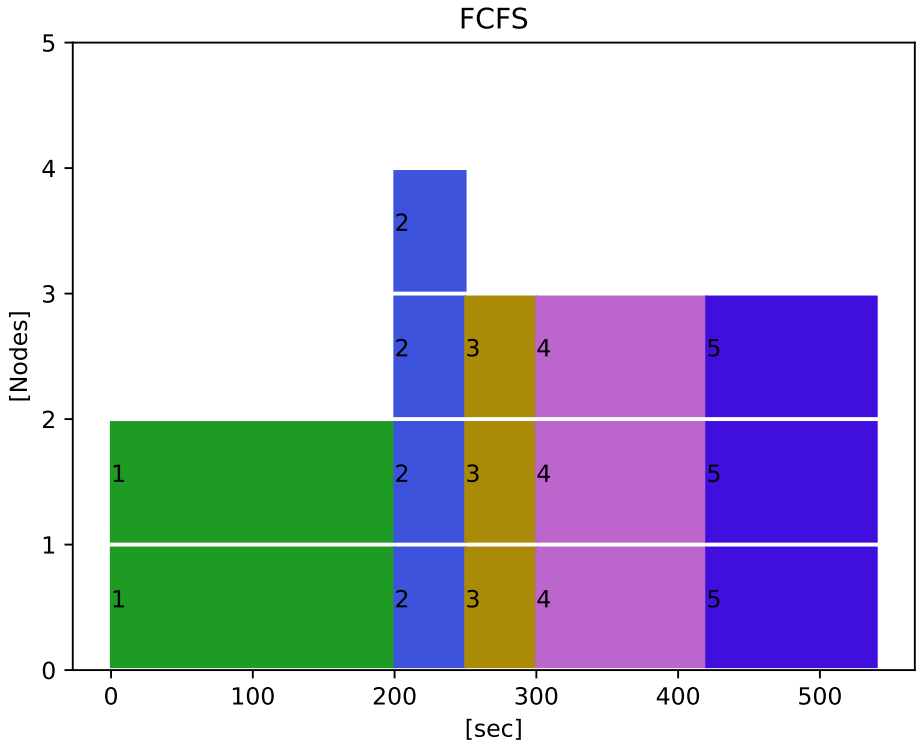

Abbildung 2.1: FCFS-Algorithmus

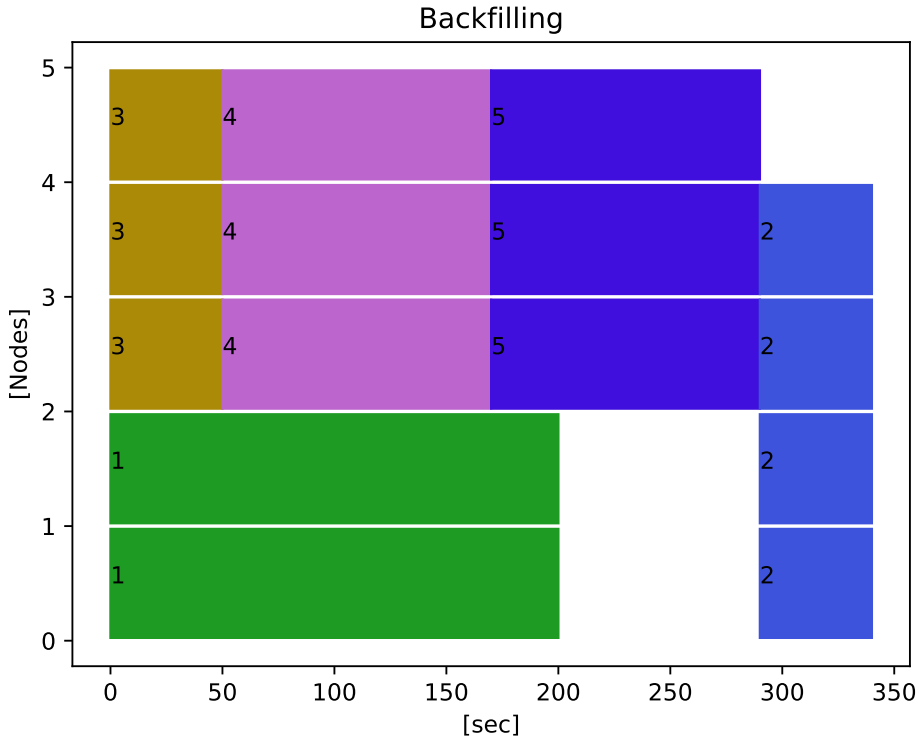

Abbildung 2.2: Backfilling-Algorithmus

2.1.3 Easy-Backfilling-Algorithmus

Der Easy-Backfilling-Algorithmus ist eine faire Version des Backfilling-Algorithmus [4, 5]. Voraussetzung für die Verwendung dieses Algorithmus ist die Angabe eines Zeitlimits im Job. D.h. es muss eine Zeit angegeben werden, wie lange der Job maximal für die Berechnung benötigt, wird diese Zeit überschritten, wird der Job beendet. Um sicherzustellen, dass *q_head* nicht nach hinten verschoben wird, wird für diesen ein möglicher Startzeitpunkt berechnet, der nicht nach hinten verschoben werden darf. Die Startzeit errechnet sich aus den Zeitlimits der auszuführenden Jobs. Für ein Beispiel siehe Abbildung 2.3. Job 1 wurde gestartet und Job 2 ist *q_head*. Job 1 hat ein Zeitlimit von 200 Sekunden, daher ist die Startzeit von Job 2 200 Sekunden. Der Algorithmus kann nun prüfen, ob er andere Jobs vorziehen kann, die Job 2 nicht verschieben. Job 3 mit einem Zeitlimit von 50 Sekunden und Job 4 mit einem Zeitlimit von 120 Sekunden passen in die Lücke, ohne dass sich die Startzeit von Job 2 verschiebt. Da aber Job 5 mit einem Zeitlimit von 120 Sekunden den Start von Job 2 verzögern würde, muss Job 5 warten.

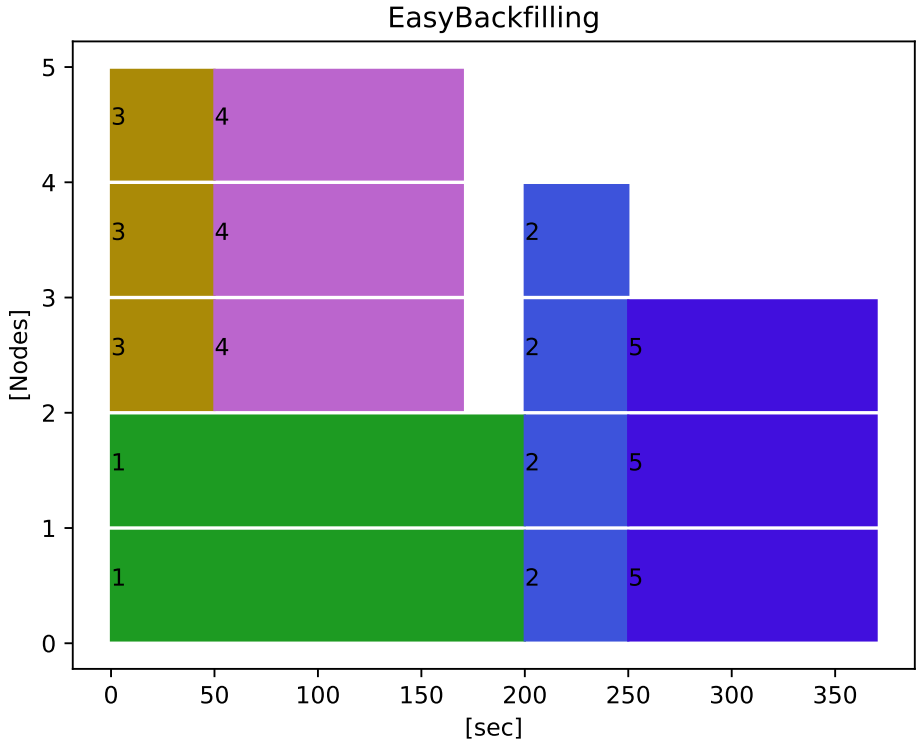

Abbildung 2.3: Easy-Backfilling-Algorithmus

2.1.4 Malleable-Algorithmus

Der vierte Algorithmus ist eine Abwandlung des Minimale Nodeanzahl (*min_agree*) Malleable-Basisalgorithmus aus der Bachelorarbeit von Fabian Hupfeld [3, 7]. Der Basisalgorithmus ist wie folgt aufgebaut:

1. **Easy-Backfilling:** siehe Kapitel 2.1.3.

2. **Shrink:** Nodes von laufenden Malleable-Jobs freigeben und zum Start neuer Jobs verwenden.

3. **Expand:** Freie Nodes dem laufenden Malleable-Jobs zuweisen.

In einem ersten Schritt werden die Jobs mit Hilfe des Easy-Backfilling-Algorithmus auf dem Cluster gestartet. Wenn nach Schritt 1 keine neuen Jobs gestartet werden können, wird Schritt 2 ausgeführt. Damit Schritt 2 ausgeführt wird, muss *q_head* existieren, sonst weiter mit Schritt 3. Im zweiten Schritt erhalten alle laufenden Jobs eine Rangfolge ($ranking\_shrink_{Job}$), die sich aus der aktuellen Node-Anzahl ($current\_nodes_{Job}$) minus der minimalen Node-Anzahl (min_nodes_{Job}) des Jobs errechnet.

$$ranking\_shrink_{Job} = current\_nodes_{Job} - min\_nodes_{Job}$$

Die laufenden Jobs werden anhand $ranking\_shrink_{job}$ von groß nach klein sortiert (*q_ranking_shrink*). Gleichzeitig wird die Liste *q_ranking_shrink* aufsummiert (*max_Nodes*), um zu sehen, wieviele Nodes durch shrink maximal freigeben werden können. Dann wird geprüft, ob *q_head* mit minimaler Anzahl von Nodes größer oder gleich *max_Nodes* ist. Wenn dies der Fall ist, wird die *q_ranking_shrink*, beginnend mit dem größtem $ranking\_shrink_{Job}$, so lange laufende Nodes des jeweiligen Jobs reduzieren, bis die minimale Node-Anzahl von *q_head* erreicht wurde. Sobald dies erreicht wird, wird *q_head* gestartet und es geht weiter mit Schritt 3. Schritt 3 prüft zunächst, ob noch freie Nodes vorhanden sind. Sind keine Nodes vorhanden, wird Schritt 3 übersprungen und es startet wieder mit Schritt 1. Wenn Nodes vorhanden sind, werden diese laufenden Jobs über eine Rangfolge zugewiesen. Diese Rangfolge ($ranking\_grow_{Job}$) berechnet sich aus den laufenden Jobs wie folgt. Die maximale Anzahl von Nodes, die ein Job haben kann, minus der aktuellen Anzahl von Nodes, die ein Job gerade hat.

$$ranking\_grow_{Job} = max\_nodes_{Job} - current\_nodes_{Job}$$

Die laufenden Jobs werden anhand $ranking\_grow_{Job}$ von groß nach klein sortiert ($q\_ranking\_grow$). Mit Hilfe der Liste $q\_ranking\_grow$ werden dann, beginnend mit dem größten $ranking\_grow_{Job}$, jedem Job so lange Nodes zugewiesen, bis der Job keine Nodes mehr aufnehmen kann und somit der nächste Job mit Nodes aufgefüllt wird bis entweder alle Jobs keine Nodes mehr aufnehmen können oder keine freien Nodes mehr vorhanden sind.

2.2 Anwendungen für die Jobs

2.2.1 APGAS

APGAS (Asynchronous Partitioned Global Adress Space) ist ein Modell und gibt es für Java als Bibliothek. APGAS unterstützt das Ausführen von Programmen auf mehreren Rechnern (*Nodes*) und fasst die Daten der einzelnen Rechner in einem virtuellen Speicher zusammen. APGAS ist eine Erweiterung, die auf dem Programmiermodell PGAS (Partitioned Global Address) basiert [8]. PGAS ermöglicht die Parallelisierung eines Programms zwischen den Nodes eines Clusters. Jeder Node hat seinen eigenen lokalen Speicher, welcher zu einem virtuellen Speicher zusammengefasst wird. Dieser ermöglicht den direkten Zugriff auf entfernte Speicherorte anderer Nodes.
APGAS [9, 10] erweitert das Programmiermodell PGAS durch asynchrone Task-Funktionen. Diese Tasks können dann an Nodes gesendet werden, die sie dann bearbeiten. Die Asynchronität von APGAS ermöglicht die Erstellung von asynchronen Tasks. Diese können dann von den Workern auf den einzelnen Nodes bearbeitet werden.
In dieser Arbeit wurde eine neue Implementierung von APGAS verwendet, welche eine Malleability-Unterstützung hat [2, 11]. Diese Version ermöglicht es, während der Ausführung zu schrumpfen (*shrink*) oder zu wachsen (*expand*). Die Kommunikation erfolgt über eine von APGAS bereitgestellte Schnittstelle.

shrink Die shrink-Funktion reduziert APGAS um die angegebene Anzahl an Nodes, die über die Schnittstelle empfangen worden sind. In diesem Fall muss die Nachricht „`shrink <Anzahl>`“ an die APGAS-Schnittstelle übergeben werden. Als Antwort erhält man die von APGAS freigegebenen Nodes als Textnachricht.

expand Beim Wachsen muss APGAS mitgeteilt werden, auf welche Nodes es sich ausdehnen soll. Dies ist identisch mit der shrink-Funktion über die Schnittstelle mit der Nachricht „`grow <Node_IP_1>,<Node_IP_2>,...`“. In diesem Fall bekommt man von APGAS keine Antwort.

2.2.2 GLB

Das lifeline-based Global Load Balancing (GLB) ist ein vollständig verteiltes System zum Stehlen von Arbeit. Es wurde zunächst als Bibliothek in X10 implementiert und dann nach Java portiert. GLB richtet feste Kanäle für den Diebstahl ein, über die die Arbeit von anderen Workern gestohlen werden kann. In dieser Arbeit wird eine angepasste Version [2] von GLB verwendet, die durch die Verwendung von APGAS Malleability unterstützt. Zusätzlich wird in dieser Arbeit das Benchmark UTS Programm von GLB verwendet, was von den Jobs in dieser Arbeit ausgeführt wird. Der Parameter d von UTS kann durch eine Zahl festgelegt werden.

3 Konzepte

Das Hauptziel dieser Arbeit ist die Entwicklung eines modularen *Schedulers*. Mit Hilfe des Schedulers soll es möglich sein, schnell und einfach neue Job-Scheduling-Algorithmen zu implementieren und zu evaluieren sowie neue Job-Typen (z.B. Moldable und Malleable) zu integrieren, ohne den Super-Scheduler anpassen zu müssen. Um dies zu erreichen, müssen verschiedene Kriterien erfüllt sein:

Scheduler: Die Hauptaufgabe des Schedulers besteht darin, Jobs entgegenzunehmen, zu starten und zu beenden. Daher muss der Scheduler über eine Kommunikationsschnittstelle verfügen, an die die Jobs übergeben werden können. Ein an den Scheduler übergebener Job wird dann im Scheduler zur weiteren Verarbeitung hinterlegt. Der Job wird vom Scheduler gestartet, wenn genügend freie Nodes zur Verfügung stehen. Der Job wird auf einem Node gestartet und ihm wird eine Liste von Nodes übergeben, auf die er sich ausbreiten darf. In diesem Fall ist es die Aufgabe des Jobs, mit Hilfe der APGAS-Bibliothek auf dem angegebenen Nodes zu starten, und nicht die des Schedulers. In diesem Zusammenhang ist der Scheduler auch für die Überwachung der laufenden Jobs verantwortlich und muss registrieren, wenn diese ihre Arbeit beendet haben. Neben dem Empfangen und Starten von Jobs muss der Scheduler auch die Verwaltung der Nodes übernehmen, d.h. es darf nicht vorkommen, dass 2 Jobs gleichzeitig auf dem selben Node laufen. Jeder Job muss ein Zeitlimit angeben, damit der laufende Job nicht für immer im Scheduler verbleibt. Bei Zeitüberschreitung wird der laufende Job vom Scheduler beendet.
Zusätzlich werden alle Ausgaben vom Scheduler protokolliert und in einer txt-Datei gespeichert.

Modularität: Der Scheduler ist so konzipiert, dass er leicht erweiterbar und konfigurierbar ist. Ein Schwerpunkt der Entwicklung ist das Ermöglichen von neuen Job-Scheduling-Algorithmen und deren Evaluierung. Um dies zu zeigen, wurden die 4 Job-Scheduling-Algorithmen aus den Grundlagen (Kapitel 2) implementiert. Außerdem sollen die Job-Typen leicht erweiterbar sein, d.h. eine mögliche Erweiterung auf Moldable, Malleable oder Evolving. Das Ziel ist auch, dass der Scheduler in der Lage ist, während der Ausführung mit verschiedenen Job-Typen umzugehen.

Eigenes Cluster im Cluster bilden: Damit der Scheduler seine eigenen Nodes selbständig verwalten kann und vom Super-Scheduler unabhängig ist, bildet er auf einem Supercomputer ein eigenes Cluster. Um dies zu ermöglichen, muss eine beliebige Anzahl von Nodes auf einem Supercomputer reserviert werden. Der Scheduler wird

auf einem Node gestartet und die anderen reservierten Nodes werden nach dem Start des Schedulers über dessen Schnittstelle übergeben. Auf diesen Nodes kann der Scheduler nun eigenständig Jobs starten und stoppen, ohne den Super-Scheduler zu verwenden.

Jobs: Folgende Job-Typen werden unterstützt: Rigid, Moldable und Malleable. Damit der Scheduler weiß, von welchem Typ ein Job ist, muss dies im Job angeben werden. Wenn der Job nicht als malleable gekennzeichnet ist, wird dieser als rigid betrachtet. In dieser Arbeit werden Jobs mit der Kennung malleable immer als Malleable-Jobs gestartet. Da es jedoch Job-Scheduling-Algorithmen gibt, die keine Malleability-Unterstützung haben, betrachten diese Algorithmen Malleable-Jobs als Moldable-Jobs.

Evolving-Jobs werden noch nicht unterstützt, da APGAS diese Möglichkeit noch nicht bietet und im Rahmen dieser Arbeit keine Zeit war, neben dem Scheduler auch APGAS dafür anzupassen. Am Ende der Arbeit wird eine kurze Zusammenfassung gegeben, wie diese Erweiterung im Scheduler realisiert werden könnte.
Das zweite Ziel der Arbeit war die Evaluierung der implementierten Job-Scheduling-Algorithmen mit Hilfe des Schedulers. Die Algorithmen mussten unterschiedliche Anteile von Rigid- und Moldable/Malleable-Jobs verarbeiten. Die Experimente wurden mit Hilfe einem Python-Skript nach dem Zufallsprinzip generiert und danach mit einem weiterem Python-Skript ausgewertet.

4 Scheduler

Um einen besseren Überblick über die Funktionsweise des Schedulers zu erhalten, zeigt Abbildung 4.1 schematisch die Arbeitsweise des Schedulers. Die Ellipsen stellen einzelne Threads dar, die vom Scheduler gestartet wurden. Die Karos geben an, wann sich diese Threads wiederholen. Die Rechtecke sind Klassen, wobei die Klassenvariablen mit einem `V` und die Methoden mit einem `M` gekennzeichnet sind. Soll mit

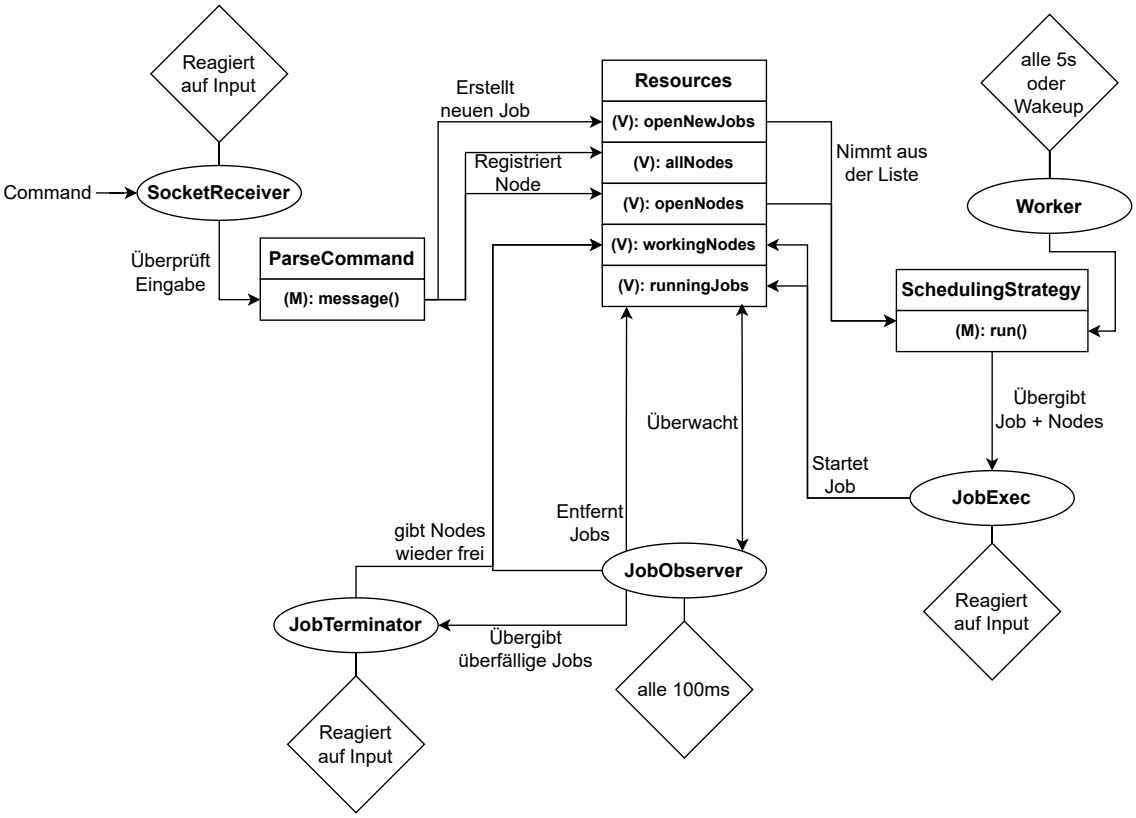

Abbildung 4.1: Schematischer Ablauf im Scheduler

dem Scheduler kommuniziert werden, so geschieht dies über die Schnittstelle, die vom Thread `SocketReceiver` (Kapitel 4.1.3) gestartet und überwacht wird. Dieser Thread empfängt die Befehle und wertet sie mit der Methode `message()` in der Klasse `ParseCommand` aus. Ein Befehl kann beispielsweise die Registrierung eines neuen Node im Scheduler sein. Der neue Node wird in den Listen `allNodes` und `openNodes` der Klasse `Ressources` gespeichert. Jobs werden ebenfalls über die gleiche Schnittstelle übergeben, müssen aber als Pfad auf das auszuführende Batch-Skript zeigen. Der `ParseCommand` erzeugt dann im Scheduler ein Job-Objekt mit dem angegebenen Pfad und legt es in der Liste `openNewJobs` ab.
Die Methode `run()` der Klasse `SchedulingStrategy` wird vom Thread `Worker` (Kapitel 4.1.1) nach einer Wartezeit von 5 Sekunden (diese ist konfigurierbar) oder per

Wakeup von anderen Threads aufgerufen. Diese Klasse kann für eine der vier implementierten Job-Scheduling-Strategies stehen, die festgelegt werden, wenn der Scheduler gestartet wird. Zuerst prüft die Klasse, ob neue Jobs in der Liste `openNewJobs` enthalten sind. Ist dies der Fall, werden die Jobs aus der Liste entfernt und deren Header (Kapitel 4.3) überprüft. Im Job-Header müssen folgende Parameter angegeben werden: Anzahl der Nodes, Name des Jobs, Zeitlimit, Job-Typ und mit welcher Bibliothek der Job läuft. Bei unvollständigen Angaben werden die entsprechenden Jobs aussortiert. Wenn die Angaben des Jobs vollständig sind, wird er in einer internen Datenstruktur der Klasse `SchedulingStrategy` gespeichert und verwaltet. Diese Datenstruktur hängt vom verwendeten Job-Scheduling-Algorithmus ab. Im zweiten Schritt wird der Job-Scheduling-Algorithmus ausgeführt. Dazu werden die internen Jobs der Klasse und die Liste der freien Nodes (`openNodes`) verwendet. Soll nun ein Job gestartet werden, so wird dieser an den Thread `JobExec` (Kapitel 4.1.4) mit den entsprechenden Nodes aus der `openNodes` Liste übergeben. Die entsprechenden Nodes werden ebenfalls aus der Liste der `openNodes` entfernt.
Der `JobExec` verbindet sich mit dem Ziel-Node, um dort den Job zu starten. Nachdem der Job gestartet wurde, wird im Job-Objekt ein Prozess hinterlegt, mit dem überprüft werden kann, ob der Job noch läuft oder bereits beendet ist. Zusätzlich wird der Job in die Liste `runningJobs` und die Nodes in die Liste `workingNodes` aufgenommen.
Der Thread `JobObserver` (Kapitel 4.1.2) überprüft in regelmäßigen Abständen die Liste der `runningJobs`. Wenn ein Job beendet ist, wird er aus der Liste `runningJobs` entfernt und die Nodes des Jobs werden aus der Liste `workingNodes` in die Liste `openNodes` verschoben. Überschreitet ein Job sein angegebenes Zeitlimit, wird er an den `JobTerminator` (Kapitel 4.1.5) übergeben und aus der Liste der `runningJobs` entfernt. Der `JobTerminator` beendet den ihm übergebenen laufenden Job. Sobald der Job beendet ist, werden die Nodes (wie beim `JobObserver`) von den `workingNodes` auf die `openNodes` übertragen.
Im Unterkapitel Funktionsweise (4.1) werden die einzelnen Threads näher erklärt. Es folgt eine Beschreibung, wie der Scheduler gestartet wird (4.2). Wie Jobs aufgebaut seien müssen wird in Kapitel 4.3 erklärt. Anschließend wird erläutert, wie die Algorithmen angepasst (4.4) wurden und wie man neue Algorithmen implementieren (4.5) kann. Abschließend werden die Änderungen in APGAS erläutert (4.6).

4.1 Funktionsweise

4.1.1 Worker

Der `Worker` ist das Herzstück des Schedulers. Nachdem der `Worker` gestartet wurde, startet dieser die Threads `JobObserver`, `JobExec` und `JobTerminator`. Anschließend beginnt der `Worker` eine Loop-Schleife, die erst dann beendet wird, wenn der Scheduler den Shutdown-Befehl erhält und alle Jobs im Scheduler abgearbeitet wurden. Nach dem Abschalten werden auch die zuvor gestarteten Funktionen und der `SocketReceiver` (siehe 4.1.3) beendet. Die Loop-Schleife wird alle 5 Sekunden wiederholt oder kann von außen durch einen Wakeup-Call vorzeitig gestartet werden. Wakeup-Calls können durch folgende Events ausgelöst werden: ein neuer Job wurde übergeben, ein laufender Job wurde beendet oder ein laufender Job wurde vom `JobTerminator` beendet. Die Idee hinter Wakeup-Calls ist, dass der `Worker` dynamisch auf Events reagieren soll, um Wartezeiten zu vermeiden. In dieser Schleife wird nun die Methode `run()` des Job-Scheduling-Algorithmus aufgerufen. Der zu verwendende Job-Scheduling-Algorithmus wird beim Start des Schedulers festgelegt. Das Kapitel 4.5 erläutert die Methode `run()` und beschreibt, wie beim Start des Schedulers festgelegt wird, welcher Job-Scheduling-Algorithmus verwendet werden soll.

4.1.2 JobObserver

Der `JobObserver` prüft alle 100ms, ob einer der laufenden Jobs fertig ist. Grundsätzlich ist es einfacher, wenn der Job selbst Bescheid gibt, dass er fertig ist. Allerdings führt dies zu einem Mehraufwand für den Entwickler, der seinen Job bei dem Scheduler abgeben will. Da jeder Job so angepasst werden muss, dass er dem Scheduler antworten muss, wenn dieser fertig ist. Hat ein Job diese Anpassung nicht, läuft dieser endlos und wird vom Scheduler nicht entfernt. Daher übernimmt der Scheduler diese Aufgabe, um dies zu vermeiden.
Die Wartezeit kann über die Konstantenklasse mit Hilfe der Variablen `jobObserverSleepInMillis` eingestellt werden. Wenn ein Job normal beendet wird, erkennt der `JobObserver` dies und entfernt den Job aus der Liste der laufenden Jobs (`runningJobs`). Darüber hinaus werden die vom Job belegten Nodes wieder zurück in den Pool freier Nodes (`openNodes`) übergeben. Falls ein Job länger als angegeben benötigt, wird er als veraltet erkannt, aus der Liste der laufenden Jobs ent-

fernt und an den `JobTerminator` weitergeleitet, der dann den laufenden Job beendet.

4.1.3 SocketReceiver

Der `SocketReceiver` ist die Schnittstelle, über die die Kommunikation mit dem Scheduler möglich ist. Dabei wird in diesem ein serverSocket mit einer Portnummer erzeugt, die man in der Konstantenklasse (SCHEDULER_PORT) setzen kann, bevor das Programm startet. In dieser Arbeit handelt es sich um die Portnummer 8081.
Es gibt verschiedene Möglichkeiten, Nachrichten an den Scheduler zu senden. Am einfachsten ist es, die selbst entwickelte Klasse `sfbatch` zu verwenden, die beim Start verschiedene Argumente entgegennehmen kann und diese dann per Socket an den Scheduler sendet. Die Ausführung der Klasse muss jedoch auf demselben Node gestartet werden, auf dem auch der Scheduler läuft. Um den Scheduler von außen zu erreichen, muss in einer Socket-Nachricht neben der Portnummer auch die IP-Adresse des Node angegeben werden.
Der `SocketReceiver` kann die folgenden vier Meldungen verarbeiten, bei anderen oder falschen Angaben wird die Meldung vom `SocketReceiver` ignoriert.

Neue Nodes hinzufügen:

Um dem Scheduler neue Nodes hinzuzufügen, setzt man das Schlüsselwort „`Add Node`“ an den Anfang der Nachricht. Anschließend gibt man die IP-Adressen der Nodes an, die man hinzufügen möchte. Möchte man mehr als einen Node gleichzeitig hinzufügen, müssen diese durch Kommas getrennt werden. Im Folgenden ist eine Beispielnachricht zu sehen.

```
„Add Node node1,node2,node3,...,nodeN"
```

Zusätzlich wird jeder Node vom Scheduler angepingt, bevor er hinzugefügt wird. Dies soll verhindern, dass Nodes hinzugefügt werden, die nicht erreichbar sind oder gar nicht existieren. Schlägt der erste Ping fehl, wird er 4 mal wiederholt, danach gilt der Node als nicht erreichbar und wird nicht in den Scheduler übernommen. Gleichzeitig wird der Benutzer informiert, dass der Node nicht hinzugefügt wurde.

Einen Job einreichen:

Wenn man möchte, dass der Scheduler einen bestimmten Job ausführt, muss man ihm den Pfad zum Batch-Skript als String übergeben. Momentan erkennt der Scheduler nur Batch-Skripts als Jobs. Die abgegebenen Jobs werden in der Liste `openNewJobs` als Objekte der Klasse `Job` gespeichert und enthalten nur den Pfad zum Batch-Skript. Im Folgenden ist eine Beispielnachricht zu sehen.
`„apgas-jobs/malleable/14/malleable-UTS14-1_2.sh"`

Den Scheduler beenden:

Bei Verwendung des Schlüsselworts `„shutdown"` wird der Scheduler gestoppt, sobald sich im Scheduler keine weiteren Jobs mehr befinden.

APGAS ist Malleable-fähig:

Der Scheduler betrachtet jeden Malleable-Job beim Start als Moldable-Job. Erst wenn der Job dem Scheduler mitteilt, dass er den `MalleableHandler` registriert hat, wird er als Malleable-Job anerkannt. Während des Starts von APGAS können keine Shrink- und Expand-Anfragen an den Job gesendet werden, d.h. APGAS ist erst nach dem Start Malleable-fähig. Details zur Umsetzung finden sich im Kapitel 4.6.

4.1.4 JobExec

Der `JobExec` ist für den Start der Jobs auf seinem Ziel-Node verantwortlich. Dazu wird eine Verbindung zum Ziel-Node aufgebaut. Dies geschieht entweder über eine ssh- oder srun-Verbindung, die beim Start festgelegt wird. Beim Start des Jobs erzeugt der `JobExec` einen Prozess, der auf dem Job-Objekt des Jobs gespeichert wird. Zusätzlich werden die Out- und Error-Streams des Jobs auf die entsprechenden Job-Dateien umgeleitet. Im Gegensatz zum `JobObserver` oder `Worker` läuft der `JobExec` nur, wenn ihm Jobs übergeben werden. Die übertragenen Jobs werden in eine Warteliste aufgenommen und abgearbeitet. Wenn die Liste leer ist, wartet der `JobExec`, bis entweder ein Job eintrifft oder der Scheduler heruntergefahren wird.

4.1.5 JobTerminator

Der `JobTerminator` ist verantwortlich für das Beenden von Jobs, die ihr Zeitlimit überschritten haben. Dazu wird die Methode `kill()` des Job-Objekts aufgerufen. Diese Methode führt einen Bereinigungsprozess auf allen Nodes aus, die von dem Job verwendet werden. Da in dieser Arbeit nur Jobs verwendet werden, die Java-Programme ausführen, wird auf den entsprechenden Nodes der Befehl „`killall java`“ ausgeführt. Sobald alle Nodes bereinigt sind, werden diese in die Liste der `openNodes` übernommen und der Job wird verworfen.

4.1.6 Logger

Der `Logger` ist ein Thread, der mit dem allgemeinen Ablauf im Scheduler nicht viel zu tun hat und daher auch nicht in der Abbildung zu sehen ist. Wie der Name schon sagt, werden im `Logger` alle Ausgaben des Schedulers protokolliert. Zusätzlich zur Ausgabe auf der Konsole werden die Ausgaben in eine txt-Datei geschrieben. Jede Ausgabe erhält einen Zeitstempel im ISO-Format, einen Zeitstempel in Millisekunden, von welcher Methode die Ausgabe stammt und den individuellen Inhalt, in dieser Reihenfolge.

4.2 Starten des Schedulers

Der Scheduler wurde in Java entwickelt und benötigt zum Kompilieren ein JDK größer/gleich 11. Beim Start können dem Scheduler verschiedene Parameter übergeben werden und wenn diese nicht angegeben werden, wird ein voreingestellter Standardwert verwendet:

```
java    -Dscheduler.strategies=<String>     \
        -Dscheduler.id=<String>             \
        -Dscheduler.min.nodes=<int>         \
        -Dscheduler.job.launcher=<String>   \
              scheduler.Scheduler
```

-Dscheduler.strategies=<String>: Mit diesem Parameter kann eingestellt werden, mit welchem Job-Scheduling-Algorithmus der Scheduler arbeiten soll. Dazu muss

der Klassenname des implementierten Job-Scheduling-Algorithmus im *<String>* angegeben werden. Als Beispiel: Wenn der Scheduler den Easy-Backfilling-Algorithmus verwenden soll, muss der Klassenname `EasyBackfilling` verwendet werden. Wenn keine Strategie angegeben ist, wird standardmäßig der FCFS-Algorithmus verwendet.

-Dscheduler.id=<String>: Mit Hilfe der Id wird ein Unterordner im Ordner out erstellt, in dem alle Dateien gespeichert werden, die während der Ausführung des Schedulers erzeugt werden. Wird keine Id angegeben, verwendet der Scheduler die aktuelle Zeit in Millisekunden.

-Dscheduler.min.nodes=<int>: Mit diesem Parameter kann eingestellt werden, dass der Scheduler eine Mindestanzahl von Nodes benötigt, bevor er Jobs bearbeiten darf. Wird die Nodeanzahl unterschritten, schaltet sich der Scheduler ab, sobald der erste Job eintrifft. Der Grund für die Einführung dieses Parameters ist die Tatsache, dass es sein kann, dass bei einem Experiment weniger Nodes im Scheduler registriert werden, als festgelegt. Das Experiment startet dann mit einer falschen Anzahl an Nodes und verbraucht dann unnötig Ressourcen. Wenn keine Mindestanzahl von Nodes angegeben ist, gibt es diesen Abbruch nicht.

-Dscheduler.job.launcher=<String>: Die Jobs werden entweder über ssh oder srun gestartet. Mit diesen Parametern kann eingestellt werden, welcher der beiden Modi verwendet werden soll. Mit *job.launcher.SshLauncher* wird der ssh-Launcher und mit *job.launcher.SrunLauncher* der srun-Launcher verwendet. Der srun-Launcher funktioniert nur unter Slurm. Wenn dieser Parameter nicht angegeben ist, wird beim Start der ssh-Launcher verwendet.

4.3 Aufbau der Jobs

Damit ein Job vom Scheduler verarbeitet werden kann, muss dieser folgende Parameter im Header der Job-Datei haben:

```
#JOB_TYPE <String>
#JOB_CLASS <String>
#JOB_NAME <String>
#REQUIRED_TIME <int>
#NODES <int>
#MIN_NODES <int>
#MAX_NODES <int>
```

JOB_TYPE: Mit Hilfe des Job-Typs kann der Scheduler feststellen, um welche Art von Job es sich handelt. Da der Scheduler bisher nur APGAS unterstützt, gibt es auch nur einen Job-Typ, nämlich den *apgas*.

JOB_CLASS: Die Jobklasse gibt an, ob es sich um einen Rigid-, Moldable- oder Malleable-Job handelt. Rigid-Jobs werden mit *rigid* und Moldable-/Malleable-Jobs mit *malleable* gekennzeichnet.

JOB_NAME: Da jeder Job, der vom Scheduler ausgeführt wird, eine eigene out- und error-Datei erhält, wird hier der Name des Jobs verwendet. Dies erleichtert das Auffinden bestimmter Jobs im Ordner out des Schedulers.

REQUIRED_TIME: Damit ein Job während seiner Ausführung im Scheduler nicht dauerhaft Ressourcen blockiert, muss ein Zeitlimit angegeben werden, wie lange ein Job während seiner Ausführung laufen darf. Wird diese Zeit überschritten, beendet der Scheduler vorzeitig den Job. Die Angabe ist in Millisekunden.

NODES: Das Schlüsselwort NODES gibt an, wie viele Nodes ein Job zur Ausführung benötigt. Es darf nicht mit den Schlüsselwörtern MIN_NODES und MAX_NODES in einer Job-Datei vermischt werden.

MIN_NODES/MAX_NODES: Müssen beide zusammen angeben werden. MIN_NODES gibt die minimale Anzahl von Nodes an, mit denen ein Job noch ausgeführt werden darf. MAX_NODES gibt die maximale Anzahl von Nodes an, die ein Job haben darf.

4.4 Anpassungen von vorhandenen Scheduler-Algorithmen

Da die Algorithmen FCFS, Backfilling- und Easy-Backfiling nur Rigid-Jobs unterstützen, wurden sie so angepasst, dass sie auch Moldable-Jobs unterstützen. Bei dieser Anpassung sind die Algorithmen in der Lage, mit den MIN_NODES und MAX_NODES eines Jobs umzugehen. Das bedeutet, dass die Algorithmen beim Start entscheiden können, mit welchen Nodes ein Job starten kann, solange dies noch innerhalb des Min-/Max-Nodes-Bereichs des Jobs liegt. Dabei streben die Algorithmen immer danach, die maximale Anzahl von Nodes für einen Job beim Start zu

erreichen. Ansonsten arbeiten die Algorithmen wie aus der Literatur bekannt. Der Malleable-Algorithmus wurde ebenfalls geringfügig angepasst. In diesem Fall werden die Jobs als Moldebale-Jobs gestartet und erst dann als Malleable-Jobs betrachtet, wenn der Job das Signal gegeben hat, dass dieser hochgefahren ist (siehe Kapitel 4.6).

4.5 Implementierung neuer Scheduler-Algorithmen

Um einen neuen Job-Scheduling-Algorithmus zu implementieren, müssen verschiedene Bedingungen erfüllt sein, damit der Algorithmus auch im Scheduler läuft. Das Wichtigste ist, dass der Algorithmus eine eigene Klasse hat, die sich im gleichen Ordner wie die Klasse `SchedulingStrategy` befindet und von dieser erbt. Die Klasse `SchedulingStrategy` enthält zwei abstrakte Methoden, die von der neuen Klasse implementiert werden müssen. Die Methode `run()` ist die Methode, die vom `Worker` regelmäßig ausgeführt wird, und muss daher den gewünschten Job-Scheduling-Algorithmus enthalten. Die Methode `checkForNewJobs()` sollte in der Methode `run()` des Algorithmus am Anfang implementiert werden. In der Methode `checkForNewJobs()` werden die Jobs aus der Liste `openNewJobs` der Klasse `Resources` entnommen und in eine eigene Datenstruktur (Liste, Map, etc.) des Algorithmus überführt. Der Grund dafür ist, dass verschiedene Job-Scheduling-Algorithmen unterschiedliche Datenstrukturen haben können, um die Jobs zu verwalten. Dadurch muss der Scheduler nicht an verschiedenen Stellen angepasst werden und es bleibt die Flexibilität erhalten, dass unterschiedliche Job-Scheduling-Algorithmen implementiert werden können.
Um den neuen Algorithmus zu verwenden, muss beim Start nur der Klassenname angegeben werden, siehe 4.2.

4.6 Anpassungen von APGAS

APGAS bietet zwar eine Malleability-Unterstützung, aber diese auch erst wenn APGAS hochgefahren ist. Das heißt, erst wenn APGAS gestartet ist, wird auch die Kommunikationsschnittstelle geöffnet, an die der Scheduler eine Nachricht senden kann. Nachrichten, die vor diesem Zeitpunkt an die Jobs gesendet werden, erreichen die Jobs nicht. Um zu verhindern, dass der Scheduler Shrink- oder Expand-Befehle an Jobs sendet, die dazu noch nicht bereit sind, sendet APGAS nach dem Start der

Kommunikationsschnittstelle eine Nachricht an den Scheduler. Daher wird der Job beim Start als Moldable-Job betrachtet und erst beim Empfang der Nachricht als Malleable-Job. Ein weiterer Grund für diese Art der Implementierung ist die Art und Weise, wie der Scheduler mit der Fehlerbehandlung in den Job-Anfragen umgeht. Jeder Job, bei dem eine Anfrage in einer Exception endet, wird vom Scheduler wieder als Moldable-Job eingestuft. Der Grund dafür ist, dass der Status des Jobs nicht mehr konsistent ist, wenn die Anfrage mit einer Exception endet. Um zu verhindern, dass Jobs Nodes erhalten, auf denen bereits ein Job läuft, werden die Jobs und ihre Nodes bei einer Exception gesperrt (zu Moldable-Jobs degradiert).
Wenn z.B. ein Job eine Shrink-Anfrage erhält und eine Exception auftritt, bleiben alle Nodes beim Job, auch die, die bereits freigegeben wurden. Bei einer Grow-Anfrage mit einer Exception werden alle Nodes plus die Nodes in der Grow-Anfrage dem Job zugeordnet, bis dieser aus der Liste der `runningJobs` entfernt wird.

5 Experimente

5.1 Aufbau

Um die Funktionalität des entwickelten Schedulers zu demonstrieren, wurden die darin implementierten Algorithmen mit Hilfe des Schedulers auf dem Uni-Cluster der Partition public2023 evaluiert. Im Folgenden werden die Konfigurationen der Evaluation beschrieben.

Scheduler-Konfiguration: Pro Experiment wurde der Scheduler mit 16 Nodes gestartet und musste 100 Jobs mit einem unterschiedlichen Anteil an Rigid- zu Moldable- und Malleable-Jobs verarbeiten. Für jedes Experiment wurde einer der vier implementierten Job-Scheduling-Algorithmen verwendet. Außerdem nutzten alle Scheduler den srun-Launcher.

Job-Konfiguration: Alle Jobs führen den GLB Benchmark UTS aus. Der Parameter d von UTS wurde zwischen 15 und 19 eingestellt. Die Anzahl der Nodes variiert zwischen 1 und 16. Jeder Moldable-Job wird als Malleable-Job gestartet. Wenn der Scheduling-Algorithmus Malleability nicht unterstützt, werden malleable Jobs vom Algorithmus als moldable betrachtet.

Experimente Erstellung: Die Experimente wurden mittels einem Zufallsgenerator mit eingestelltem Seed generiert und wurden dann mit jedem Job-Scheduling-Algorithmus ausgeführt. Dabei wurde der Anteil von Moldable- und Malleable-Jobs stetig erhöht und zwar immer um 20%. Dies führte dazu, dass ein Experiment pro Algorithmus 6 Experimente ausführen musste, was eine Anzahl von 24 Experimenten pro Seed ergibt. In dieser Arbeit wurden die Job-Scheduling-Algorithmen mit 5 Seeds evaluiert. Dies entspricht einer Anzahl von 120 Experimenten, die auf dem Cluster durchgeführt werden mussten.

5.2 Ergebnisse

Der Scheduler und die Algorithmen wurden mit folgenden Metriken evaluiert.

5.2.1 Gesamtlaufzeit

Die Gesamtlaufzeit ist die Zeit vom ersten an den Scheduler übergebenen Job bis zum Berechnungsende des letzten Jobs eines Experiments. Eine kürzere Gesamtlaufzeit deutet auf eine effizientere Verarbeitung der übertragenen Jobs an den Scheduler hin. Mit einer Gesamtlaufzeit von 2900 Sekunden bei 0% Moldable-Anteil ist der FCFS (2.1.1) der langsamste Algorithmus. Durch die Erhöhung des Anteils an Moldable-Jobs verbessert sich die Gesamtlaufzeit. Bei 100% Moldable-Anteil beträgt die Laufzeit 2621 Sekunden und hat damit einen Speedup von 9,62% (Im Vergleich zu 0% Moldable-Anteil). Es ist jedoch anzumerken, dass bei einem Anteil von 40% Moldable-Anteil der Speedup bereits bei 6,02% liegt. Backfilling (2.1.2) ist der schnellste Algorithmus mit einer Gesamtlaufzeit von 2630 Sekunden bei 0% Moldable-Anteil. Es sei noch einmal darauf hingewiesen, dass Backfilling der einzige Algorithmus ist, der nicht fair agiert und auch früh abgegebene Jobs als letzte berechnen kann. Dies ist auch bei einem Anteil von 20% und 40% Moldable-Jobs der Fall, ab 60% wird der Backfilling-Algorithmus vom Malleable-Algorithmus überholt. Bei einem 100% Moldable-Anteil hat der Backfilling eine Laufzeit von 2290 Sekunden, was einem Speedup von 12,91% entspricht.

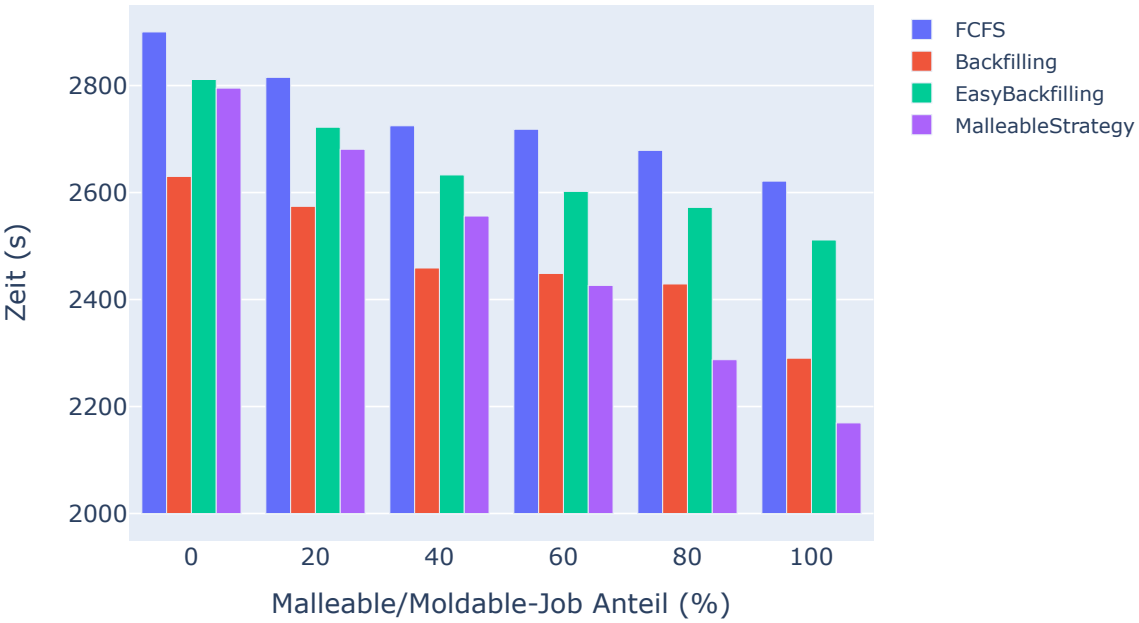

Abbildung 5.1: Gesamtlaufzeit

Der Easy-Backfilling (2.1.3) hat eine Laufzeit von 2811 Sekunden bei 0% Moldable-Anteil. Wie bei den anderen Algorithmen wird auch hier die Laufzeit durch die Erhöhung des Moldable-Anteils verkürzt. Bei einem Anteil von 100% Moldable-Jobs beträgt die Laufzeit 2511 Sekunden, was einem Speedup von 10,67% entspricht. Betrachtet man nun den Malleable-Algorithmus (2.1.4), so fällt auf, dass dieser bei

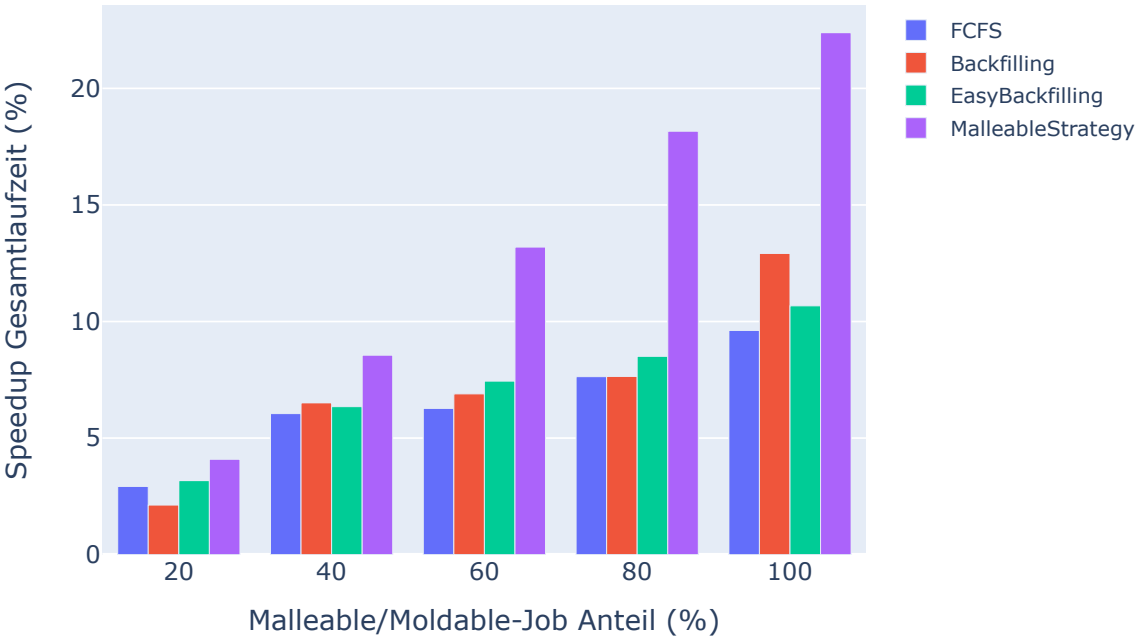

Abbildung 5.2: Gesamtlaufzeit [Speed Up]

einem Anteil von 0% Malleable-Jobs eine andere Gesamtlaufzeit hat als der Easy-Backfilling-Algorithmus, obwohl der Malleable-Algorithmus bei 0% Malleable-Jobs gleich arbeitet wie der Easy-Backfilling-Algorithmus. Der Grund dafür ist, dass Jobs keine konstante Rechenzeit haben. Jeder Job muss gestartet und berechnet werden, und das kann unterschiedlich lange dauern, was zu dieser Verschiebung von 16 Sekunden führt. Wie man sehen kann, hat der Malleable-Algorithmus bei einem Anteil von 100% Malleable-Jobs eine Laufzeit von 2169 Sekunden und damit einen Speedup von 22,39%. Es ist deutlich zu erkennen, dass mit jeder Erhöhung des Anteils an Malleable-Jobs der Speedup um 4-5% zunimmt.

5.2.2 Durchschnittliche Systemauslastung

Die Systemauslastung gibt den Wert in Prozent an, wie lange ein Node im Vergleich zur Gesamtlaufzeit genutzt wurde. Eine hohe Systemauslastung deutet auf eine bessere Nutzung der Nodes hin. Mit 82,03% bei 0% Moldable-Anteil hat Backfilling das beste Ergebnis, gefolgt von Easy-Backfilling und dem Malleable-Algorithmus mit 77%. FCFS hat eine Auslastung von 74%. Durch die Erhöhung des Moldable-/Malleable-Anteils steigt auch die Systemauslastung. Das beste Ergebnis hat der Backfilling mit 95,85%, hierbei ist aber zu erwähnen das 2 der 5 Experimente eine 100% Systemauslastung hatten. Dicht gefolgt vom Malleable-Algorithmus, der eine Systemauslastung von 94,78% hat.

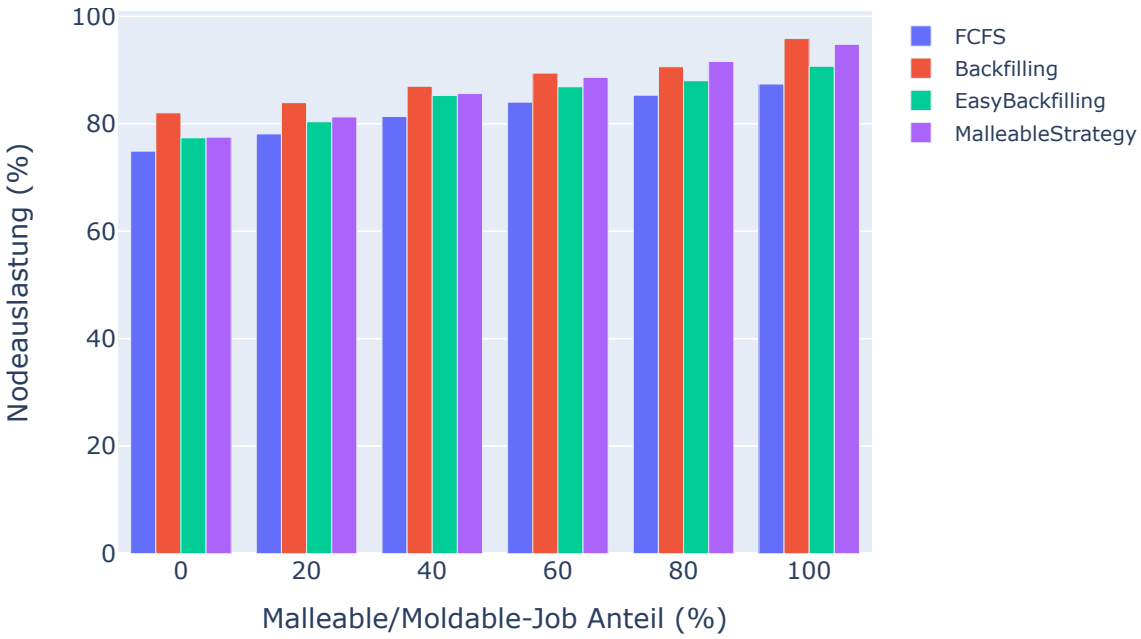

Abbildung 5.3: Systemauslastung

5.2.3 Durchschnittliche Wartezeit/Durchlaufzeit eines Jobs

Bei der Wartezeit handelt es sich um die Zeit, die zwischen der Übergabe des Jobs an den Scheduler und dem Start des Jobs vergeht. Die Durchlaufzeit ist die Zeitspanne zwischen der Abgabe an den Scheduler und dem Ende des Jobs. Je länger die Wartezeit und die Durchlaufzeit sind, desto länger muss der Benutzer, der seine Jobs abgibt, auf die Ergebnisse warten. Am längsten war der FCFS mit einer Wartezeit von 1262 Sekunden und einer Durchlaufzeit von 1300 Sekunden bei einem Moldable-Anteil von 0%. Selbst bei einem Anteil von 100% Moldable sinkt der Wert zwar auf 1097 Sekunden und die Durchlaufzeit auf 1136 Sekunden, ist aber immer noch höher als alle Werte der anderen Algorithmen. Der Grund dafür ist, dass der FCFS keine Jobs vorzieht, sondern sie nacheinander abarbeitet, wodurch sich die Wartezeit und die Durchlaufzeit erhöhen. Backfilling liefert die schnellsten Ergebnisse mit einer Wartezeit von 559 Sekunden und einer Durchlaufzeit von 597 Sekunden bei 0% Moldable-Anteil. Auch hier hat die Erhöhung des Moldabe-Anteils die Zeiten leicht verbessert, bei 100% Moldabe-Anteil liegt die Wartezeit bei 543 Sekunden und die Durchlaufzeit bei 587 Sekunden. Easy-Backfilling und der Malleable-Algorithmus haben beide ähnliche Werte. Außer bei einem 100% Malleable-Anteil, denn hier verschlechtert sich die Warte- und Durchlaufzeit des Malleable-Algorithmus schlagartig. Die Ursache konnte auch nicht durch nachträgliches Betrachten der Daten erklärt werden und steht auch im Widerspruch zu der Tatsache, dass der Malleable-Algorithmus die schnellste Gesamtlaufzeit hatte.

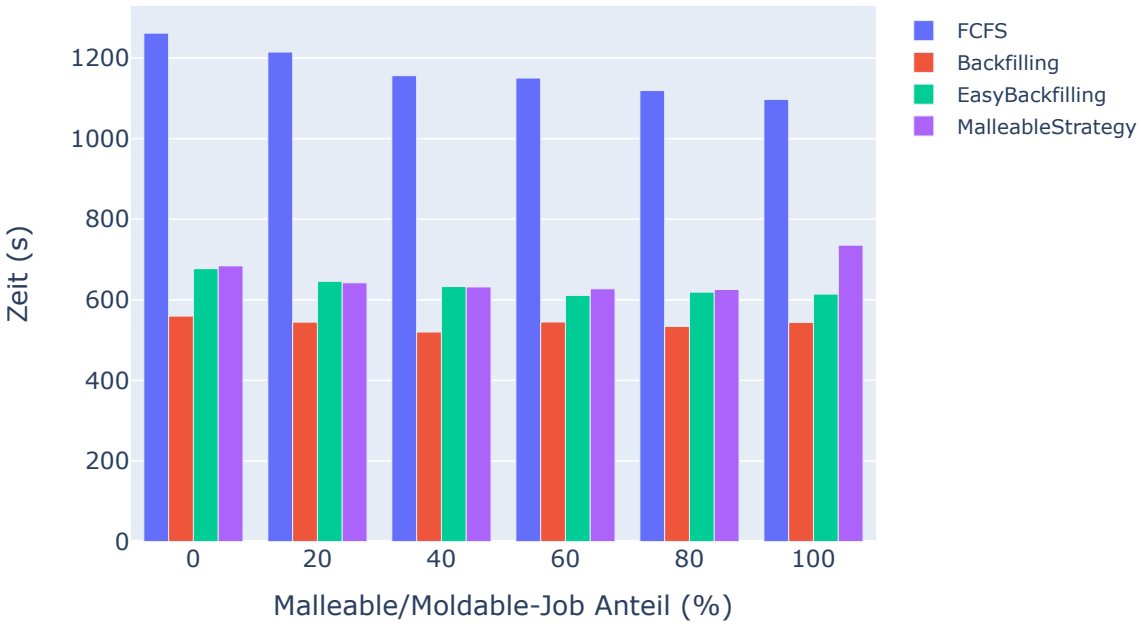

Abbildung 5.4: Durchschnittliche Wartezeit

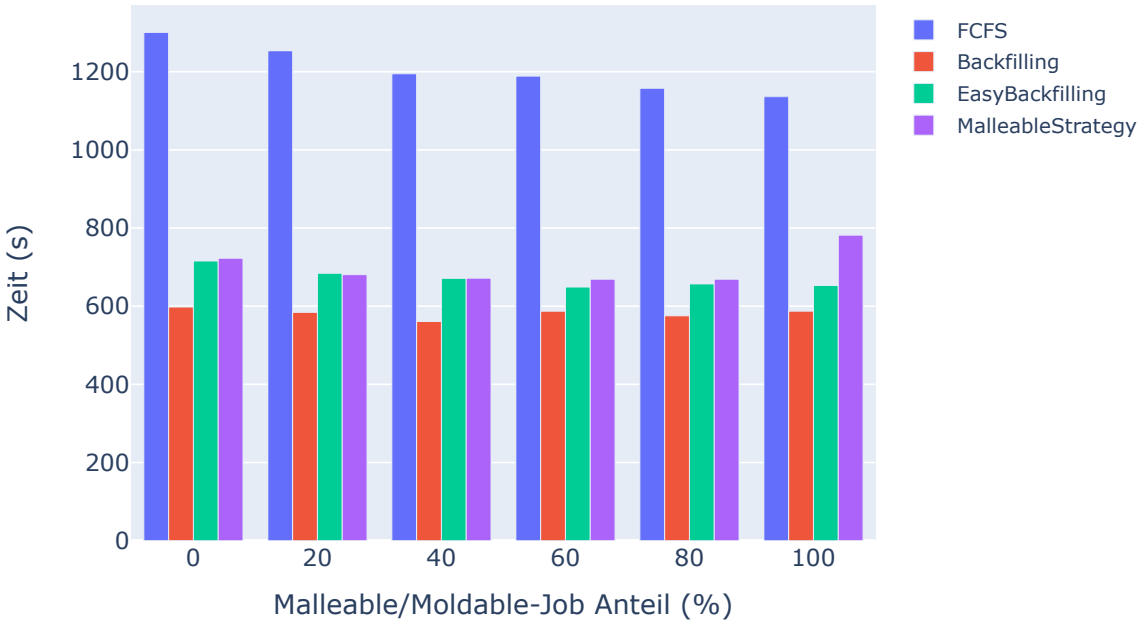

Abbildung 5.5: Durchschnittliche Durchlaufzeit

5.2.4 Durchschnittliche Rechenzeit eines Jobs

Die Rechenzeit ist die Zeit zwischen Start und Ende eines Jobs. Da jeder Job-Scheduling-Algorithmus die gleichen Experimente durchführen musste, beträgt die Rechenzeit bei einem Anteil von 0% Moldable/Malleable ca. 38,4 Sekunden. Die minimalen Abweichungen (ca. ±0,1 Sekunden) sind darauf zurückzuführen, dass gleiche Jobs bei mehrmaliger Ausführung nicht immer die gleiche Rechenzeit haben und sich geringfügig unterscheiden können. Wie aus dem Diagramm ersichtlich, erhöht sich die Rechenzeit nur beim Backfilling und beim Mallebale-Algorithmus. Beim Mallebale-Algorithmus liegt dies daran, dass die Jobs durch die Shrink- und Expand-Anfragen Zeit benötigen und somit die Jobs mehr Zeit für die Berechnung benötigen. Ein Job mit einem Malleable-Anteil von 100% benötigt somit 46 Sekunden, d. h. 8 Sekunden länger als ein Job mit einem Malleable-Anteil von 0%. Backfilling war hier eine Überraschung, da dort die Rechenzeit der Jobs mit steigendem Moldable-Anteil ebenfalls steigt. Der Grund dafür ist, dass der Algorithmus zwar die maximale Anzahl von Nodes eines Jobs anstrebt, aber durch das Vorziehen von Jobs in den meisten Fällen nur die minimale Anzahl von Nodes verwendet. Da die Jobs mit wenigen Nodes gestartet werden, erhöht sich auch hier die Rechenzeit. Diese betrug bei 100% Moldable-Anteil 44 Sekunden.

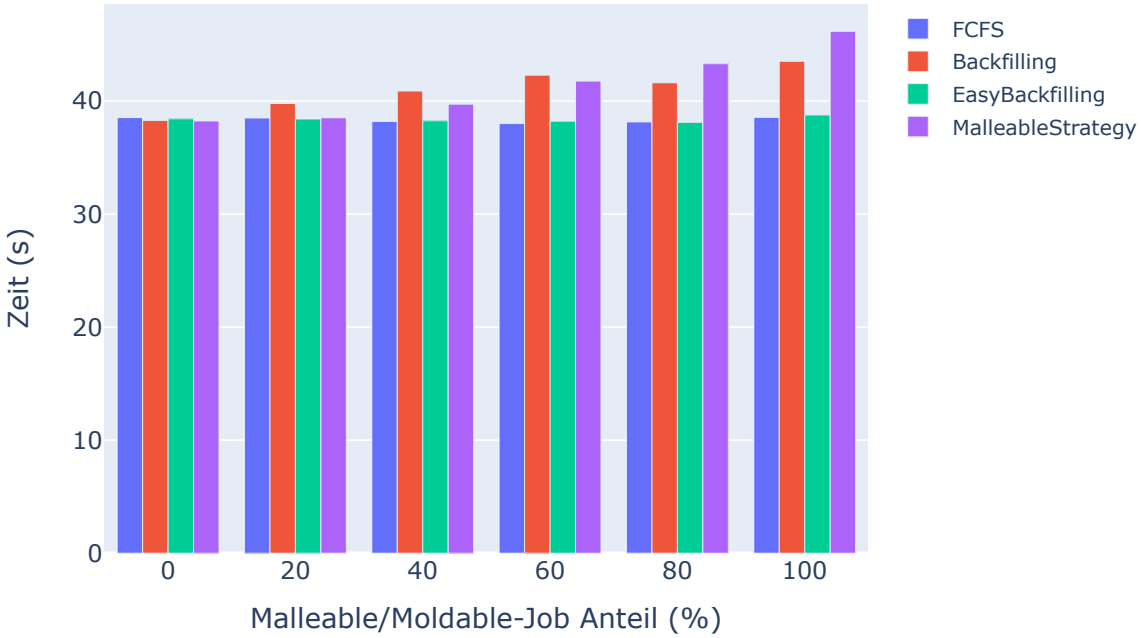

Abbildung 5.6: Durchschnittliche Rechenzeit

5.2.5 Durchschnittliche Expand/Shrink-Events pro Malleable-Job

Die Abbildung 5.7 beschreibt die durchschnittlichen Expand-/Shrink-Events pro Malleable-Job. Da nur der Malleable-Algorithmus Malleable-Jobs verwendet, wurden die Daten nur von diesem verwendet. Das Expand-Event gibt an, wie oft ein Job vom Scheduler aufgefordert wurde, um einen oder mehrere Nodes zu wachsen. Das Shrink-Event gibt an, wie oft ein Job vom Scheduler aufgefordert wurde, einen oder mehrere Nodes zu schrumpfen. Dabei ist zu beachten, dass der Scheduler erst

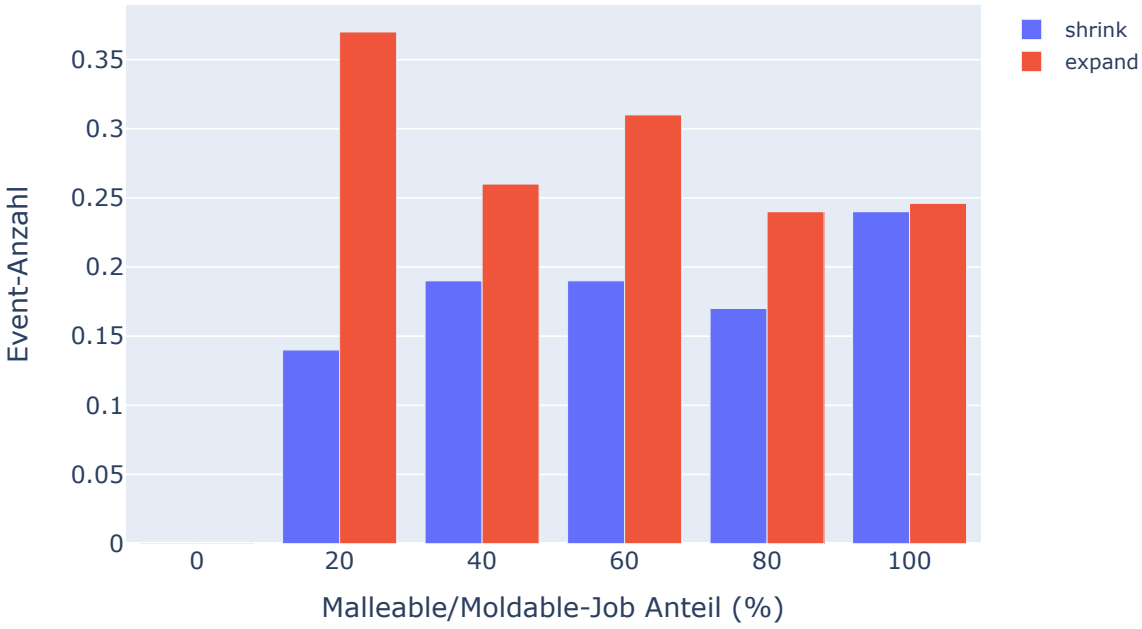

Abbildung 5.7: Durchschnittliche Shrink-/Grow-Events pro Malleable-Job

dann Anfragen an einen Malleable-Job sendet, wenn dieser gestartet wurde. Diese Antwortzeit betrug im Durchschnitt 6,3 Sekunden pro Malleable-Job. Im Allgemeinen sind die Expand-/Shrink-Events geringer als erwartet, da nicht einmal 1 Ereignis pro Job überschritten wurde. Es ist jedoch zu erkennen, dass die Shrink-Events mit steigendem Malleable-Anteil abnehmen, während die Expand-Events zunehmen.

5.3 Auswertung

Die Ergebnisse zeigen, dass der Scheduler wie geplant funktioniert. Gleichzeitig muss erwähnt werden, dass verschiedene Experimente wiederholt werden mussten, da APGAS ab und zu noch ein Problem damit hatte, herunterzufahren. Womit Jobs für mindesten 5 Minuten im Scheduler waren und dieser die Jobs dann mit Gewalt beenden musste. Da diese Fehler zufällig auftraten, konnten die Experimente

einfach wiederholt werden, ohne dass der Fehler an derselben Stelle nochmals auftrat. Mithilfe der gesammelten Daten aus den Experimenten ist zu sehen, dass die Erhöhung des Moldable-/Malleable-Anteils zu einer Verbesserung aller untersuchten Metriken führte. Ausnahme ist die Rechenzeit, die beim Backfilling und beim Malleable-Algorithmus zu einer Verschlechterung führte.
Selbst geringe Moldable-/Malleable-Anteile von 20-40% sorgen je nach Algorithmus für eine Gesamtlaufzeitverbesserung von 2-8,5%. Überraschend war die niedrige Anzahl an Shrink-/Expand-Events pro Malleable-Job. Eine Ursache dafür kann die niedrige Rechenzeit der einzelnen Jobs sein. Durch die Verwendung von Jobs mit einer längeren Rechenzeit könnte sich die Shrink-/Expand-Eventanzahl erhöhen.
Darüber hinaus konnten die Daten des Malleable-Algorithmus mit den Simulationsdaten aus der Arbeit von Hupfeld [3] verglichen werden. Diese zeigten ähnliche Ergebnisse. Ein Unterschied ist die Abflachung der Gesamtlaufzeit bei der Erhöhung des Malleable-Anteils, die in dieser Arbeit nicht gezeigt werden konnte. Gleichzeitig waren die Shrink-/Expand-Events um einiges geringer als in der Simulation. Dennoch ist hier zu erwähnen, dass die Simulation mit einer höheren Gesamtlaufzeit (720-840 Stunden) arbeitete und somit weit höher liegt als die in dieser Arbeit bestimmte Gesamtlaufzeit.

6 Fazit

Das Hauptziel dieser Arbeit war die Entwicklung eines modularen Schedulers, der auf einem Supercomputer ausführbar sein sollte und dort sein eigenes Cluster bildet. Die Hauptaufgabe des Schedulers ist die Annahme, Bearbeitung und Beendigung von Jobs. Dabei darf der Scheduler keine zwei Jobs auf demselben Node starten und muss die Verwaltung der Jobs und Nodes selbstständig übernehmen. Die Modularität kommt durch das flexible Integrieren neuer Job-Scheduling-Algorithmen. Aus diesem Grund wurden in dieser Arbeit 4 Job-Scheduling-Algorithmen implementiert und mit dem Scheduler evaluiert. Ein weitere Aufgabe des Schedulers war die Unterstützung der folgenden Job-Typen: Rigid, Moldable und Malleable. Diese Unterstützung kam durch die angepasste Bibliothek APGAS. Das zweite Ziel war die Überprüfung der Funktionalität des Schedulers mit echten Programmen. Dazu wurden mehrere Experimente durchgeführt, aus denen dann mehrere Metriken (z.B Gesamtlaufzeit) zu den einzelnen implementierten Algorithmen erstellt werden konnten. Aus diesen Metriken wurde ersichtlich, dass eine Erhöhung des Moldable-/Malleable-Anteils zu einer Verbesserung der Ergebnisse führte. Beispielsweise hatte der Malleable-Algorithmus die beste Gesamtlaufzeit bei 100% Malleable-Anteil.
Diese Arbeit dient als Grundstein für zukünftige Arbeiten, um einfacher neue Job-Scheduling-Algorithmen zu implementieren und evaluieren und diese dann mit echten Programmen zu testen, ohne den Super-Scheduler zu verändern. Gleichzeitig wurden auch schon Anpassungen gemacht, die es ermöglichen, den Scheduler auf Evolving-Jobs zu erweitern. Wenn Malleable- und Evolving-Jobs verwendet werden sollen, wird neben dem Job-Scheduling-Algorithmus auch eine Bibliothek benötigt, die den Job dazu befähigt.
Zukünftig könnten weitere Malleable-Job-Scheduling-Algorithmen mit dem Scheduler entwickelt und evaluiert werden. Zusätzlich kann eine Evolving-Unterstützung implementiert werden, sofern Jobs vorhanden sind, die Evolving-fähig sind. Somit könnten auch zukünftig neben neuen Malleable-Job-Scheduling-Algorithmen auch Evolving-Job-Scheduling-Algorithmen entwickelt und evaluiert werden.
Im Kapitel Ausblick folgt eine kurze Beschreibung, wie der Scheduler auf Evolving-Jobs angepasst werden kann.

7 Ausblick: Evolving-Unterstützung

Um den Scheduler mit Evolving-Jobs zu erweitern, müssen an mehreren Stellen kleine Erweiterungen vorgenommen werden. Zunächst muss der SocketReceiver erweitert werden, damit dieser auf eingehende Nachrichten von Evolving-Jobs reagieren kann. In diesem Zusammenhang ist auch eine Erweiterung der Klasse *Job* um einige Methoden erforderlich, die das gewünschte Shrink oder Expand des Evolving-Jobs definieren. Schließlich muss noch ein Job-Scheduling-Algorithmus implementiert werden, der mit den neuen Methoden der Job-Klasse arbeiten kann.
Ein möglicher Ablauf des Programms würde wie folgt aussehen. Ein Evolving-Job wurde vom Scheduler gestartet und möchte jetzt um 5 Nodes wachsen. Dazu kontaktiert der Job den Scheduler über den SocketReceiver. Der SocketReceiver legt nun den Wunsch des Jobs in das entsprechende Job-Objekt und weckt gleichzeitig den Worker, wenn dieser schläft. Der Worker führt nun den Job-Scheduling-Algorithmus aus, der wiederum die Running-Jobs auf Wünsche überprüfen muss. Wenn nun einer dieser Jobs den Wunsch hat, um 5 Nodes zu wachsen, kann der Job-Scheduling-Algorithmus entscheiden, ob er dies tun soll oder nicht.

Literaturverzeichnis

[1] D. G. Feitelson and L. Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. *Proceedings Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). Springer, doi:10.1007/BFb0022284*, 1996.

[2] P. Finnerty, R. Takaoka, T. Kanzaki, and J. Posner. Malleable APGAS Programs and their Support in Batch Job Schedulers. *Euro-Par Parallel Processing Workshops (AMTE), In Proceeding.*

[3] F. Hupfeld. *Weiterentwicklung und Evaluation von Scheduling-Algorithmen für elastische Jobs im High-Performance-Computing.* 2023. Bachelorarbeit an der Universität Kassel Fachgebiet Programmiersprachen/-methodik.

[4] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. *Proceedings. International Conference on Parallel Processing Workshop, DOI: 10.1109/ICPPW.2002.1039773*, 2002.

[5] S. Yi, Z.Wang, S. Ma, Z. Che, F. Liang, and Y. Huang. Combinational backfilling for parallel job scheduling. *2010 2nd International Conference on Education Technology and Computer, DOI: 10.1109/ICETC.2010.5529424*, 2010.

[6] D. Talby, D. Tsafrir, Z. Goldberg, and D. G. Feitelson. Session-Based, Estimationless, and Information-less Runtime Prediction Algorithms for Parallel and Grid Job Scheduling. 01 2006.

[7] P. Finnerty J. Posner, F. Hupfeld. Enhancing Supercomputer Performance with Malleable Job Scheduling Strategies. *Springer, to appear*, 2023.

[8] J. Jeffers, J. Reinders, and A. Sodani. Chapter 16 - PGAS programming models. *Intel Xeon Phi Processor High Performance Programming ,DOI: 10.1016/B978-0-12-809194-4.00016-8*, 2016.

[9] J. Posner, L. Reitz, and C. Fohry. Comparison of the HPC and Big Data Java Libraries Spark, PCJ and APGAS. *IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM), DOI:10.1109/PAW-ATM.2018.00007*, 2018.

[10] O. Tardieu. The APGAS library: resilient parallel and distributed programming in Java 8. *Association for Computing Machinery ,DOI: 10.1145/2771774.2771780*, 2015.

[11] J. Posner and C. Fohry. Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing. *Proc. Int. Conference on Parallel Processing (ICPP) Workshops (P2S2)*, 2021.

A Anhang

Inhalt der digitalen Abgabe:

- Digitale Fassung der Arbeit und der Grafiken
- Programmcode des Schedulers
- Alle Ergebnisse aus den Experimenten
- Programme zur Durchführung und Auswertung der Experimente