UNIKASSEL
VERSITÄT

Universität Kassel

Bachelorarbeit

Weiterentwicklung und Evaluation von Scheduling-Algorithmen für elastische Jobs im High-Performance-Computing

vorgelegt vor
Fachbereich Elektrotechnik/Informatik
Fachgebiet Programmiersprachen/-methodik

Fabian Hupfeld
35523329

Kassel, 6. Juni 2023

Gutachter:
Prof. Dr. Claudia Fohry
Prof. Dr. Gerd Stumme

Selbstständigkeitserklärung

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

_____

Kassel, 6. Juni 2023

Fabian Hupfeld

Inhaltsverzeichnis

1. Einleitung

Moderne Supercomputer ermöglichen das Ausführen komplexer Berechnungen in kurzer Zeit. Diese *HPC-Systeme* bestehen aus vielen leistungsstarken Rechnern, auch *Nodes* genannt, die zu Clustern zusammengefasst und über ein Netzwerk miteinander verbunden sind. Um große Berechnungsaufgaben (*Jobs*) zu verarbeiten, teilt der *Scheduler* des HPC-Systems die Berechnungen jedes Jobs auf ein oder mehrere Nodes auf. Nach aktuellem Stand der Technik werden *Rigid-Jobs* verwendet, denen Nodes für die gesamte Rechenzeit fest zugewiesen werden.

Diese statische Art der Zuweisung durch den Scheduler kann allerdings zu einer schlechten Systemauslastung führen. Beispielsweise bleiben nicht zugewiesene Nodes unbenutzt, bis genügend Nodes für den Start eines neuen Jobs vorhanden sind, anstatt bei den Berechnungen laufender Jobs zur Verfügung zu stehen. Außerdem benötigen Jobs möglicherweise nicht ständig alle zugewiesenen Nodes. Diese nicht benötigten Nodes könnten ebenfalls anderen Jobs zur Verfügung gestellt oder zum Starten weiterer Jobs verwendet werden.

Durch den Einsatz von *elastischen* Jobs, welchen Nodes dynamisch zugewiesen werden, kann die Ressourcenauslastung eines HPC-Systems verbessert werden. Elastische Jobs werden dabei in zwei Typen unterteilt [1]. *Evolving-Jobs* fordern während ihrer Ausführung eine Änderung der zugewiesenen Nodes vom Scheduler an, der dann weitere Nodes zuweist oder zugewiesene Nodes entfernt. Im

Gegensatz dazu wird eine Änderung der Nodezuweisung bei *Malleable-Jobs* nicht durch den Job, sondern durch den Scheduler ausgelöst.

In dieser Bachelorarbeit wird die Leistungssteigerung von HPC-Systemen durch den Einsatz von Malleable-Jobs untersucht. Ein verwendeter Malleable-Job besteht dabei aus einer oder mehreren *Berechnungsphasen* und erlaubt eine Veränderung der zugewiesenen Nodes nur zwischen zwei Berechnungsphasen. Der Schwerpunkt der Arbeit liegt auf dem Algorithmus des Schedulers. Dazu wurden zunächst zwei existierende Algorithmen untersucht: der aus der Literatur bekannte *Easy-Backfilling-Algorithmus* [2, 3], der jedoch keine Umverteilung der Nodes von Malleable-Jobs unterstützt und ein neuerer *Malleable-Basisalgorithmus*, welcher bereits Malleable-Jobs unterstützt [4]. Basierend auf dem Malleable-Basisalgorithmus wurden verschiedene Anpassungen vorgenommen, insbesondere wurden für zwei Aspekte des Algorithmus jeweils drei verschiedene Strategien entwickelt. Zur Evaluation wurde das Verhalten der Algorithmus-Varianten mit dem Programm *ElastiSim* auf künstlich generierten Jobmengen und einem künstlichen HPC-System simuliert.

Die Simulationsergebnisse zeigten keine signifikanten Unterschiede zwischen den entwickelten Algorithmus-Varianten. Im Vergleich zum Easy-Backfilling-Algorithmus führten die Algorithmus-Varianten bereits ab einem geringen Anteil an Malleable-Jobs zu einer Verbesserung der durchschnittlichen Durchlaufzeit der Jobs und der Systemauslastung. Für einen Malleable-Anteil von 100% konnte die durchschnittliche Durchlaufzeit der Jobs um 93,6% reduziert,

die Auslastung des Systems um 12,2% gesteigert und die Gesamtdurchlaufzeit aller Jobs um 13,4% verkürzt werden.

Im folgenden zweiten Kapitel wird der Aufbau und die Funktionsweise von HPC-Systemen dargestellt und die aus der Literatur bekannten Scheduling-Algorithmen beschrieben. Darauf aufbauend stellt Kapitel 3 die verschiedenen Anpassungen des Malleable-Basisalgorithmus vor. Kapitel 4 beschreibt den Simulationsaufbau und gibt einen Überblick über die Eingabekonfiguration und die Generierung der Jobmengen für die Simulationen. Die Ergebnisse der Simulationen werden in Kapitel 5 dargestellt und ausgewertet. Weiterführend gibt Kapitel 6 einen Überblick über verwandte Arbeiten und in Kapitel 7 werden die wichtigsten Erkenntnisse zusammengefasst.

2. Grundlagen

2.1. High Performance Computing

Ein HPC-System besteht aus vielen Rechnern (Nodes), die über ein Netzwerk verbunden sind. Die Jobs werden vom Benutzer in das HPC-System eingereicht und die Berechnungen eines Jobs werden vom Scheduler auf ein oder mehrere Nodes aufgeteilt. Im Rahmen dieser Bachelorarbeit wird angenommen, dass ein HPC-System aus identischen und miteinander verbundenen Nodes besteht. Jede Node ist entweder frei (nicht zugewiesen) oder genau einem Job zugewiesen.

Der Scheduler versucht die Nodes möglichst effizient auf die einzelnen Jobs zu verteilen. Dabei versuchen die meisten Scheduler die durchschnittliche Durchlaufzeit, d.h. die Zeit vom Einreichen bis zum Ende der Berechnung des Jobs, die Wartezeit der Jobs vom Einreichen bis zum Start des Jobs und die Systemauslastung, gemessen anhand der durchschnittlichen Auslastung aller Nodes, zu optimieren. Zusätzlich versuchen die meisten Scheduler Fairness zwischen den Jobs zu erreichen. Dabei soll insbesondere das endlose Warten eines Jobs durch das Vorziehen von anderen, meist kleineren, Jobs vermieden werden.

Zur Berechnung einer möglichst effizienten Zuordnung der Nodes verwendet der Scheduler einen Scheduling-Algorithmus. Ein Durchlauf des Algorithmus wird in regelmäßigen Intervallen und durch Events, wie dem Start oder Ende eines Jobs, gestartet und berechnet auf Grundlage des aktuellen Systemzustands

die Zuordnung der Nodes. Der aktuelle Systemzustand umfasst die Menge der laufenden Jobs (*r_jobs*) mit den jeweils zugewiesenen Nodes und eine Warteschlange von Jobs (*q_jobs*), die auf ihre Ausführung warten. Die Warteschlange kann jederzeit neue Jobs erhalten. Innerhalb der Warteschlange sind die Jobs nach dem Eingangs-Zeitpunkt sortiert, beginnend mit dem ältesten Job.

Beim Einreichen neuer Jobs werden zusätzliche Job-Daten vom Benutzer übergeben, um die Ressourcennutzung des Jobs besser abschätzen zu können. Dazu gehören grundlegende technische Daten, wie die zum Starten erforderliche Anzahl an Nodes oder der für die Ausführung benötigte Speicherplatz, aber auch die geschätzte Rechenzeit oder die Verzeichnisse für die Ein- und Ausgabedaten. Bei Rigid-Jobs wird zusätzlich eine exakte Anzahl der erforderlichen Nodes übergeben, während Malleable-Jobs eine Ober- und Untergrenze für die Anzahl der Nodes haben. Mit diesen zusätzlichen Daten kann der Scheduling-Algorithmus eine fundierte Entscheidung über die Zuweisungen der Nodes zu den Jobs treffen.

2.2. Easy-Backfilling-Algorithmus

Zum Vergleich mit den in dieser Arbeit entwickelten Scheduling-Algorithmen wird der aus der Literatur bekannte *Easy-Backfilling-Algorithmus* implementiert. Der Easy-Backfilling-Algorithmus basiert auf dem bekannten *First-Come-First-Serve-Algorithmus* [3], der alle Jobs genau in der Reihenfolge

ihres Eintreffens abarbeitet. Dafür wird jeweils der älteste Job (*q_head*) aus *q_jobs* gestartet, bis nicht mehr genügend freie Nodes vorhanden sind. Falls *q_head* nicht gestartet werden kann, wird auf das Ende weiterer laufender Jobs gewartet, bis genügend Nodes frei verfügbar sind.

Zusätzlich erlaubt *Easy-Backfilling* [2, 3] das vorgezogene Starten von wartenden Jobs, während *q_head* auf freie Nodes wartet. Der erwartete Startzeitpunkt von *q_head* darf durch das Vorziehen jedoch nicht verzögert werden. Um die Startzeit von *q_head* zu bestimmen, muss der Benutzer eine geschätzte Laufzeit für jeden Job angeben. Beginnend mit dem zweitältesten Job werden die Jobs aus *q_jobs* vorgezogen, solange bis zum erwarteten Start von *q_head* genügend Nodes verfügbar sind.

Dieser Algorithmus nutzt die Vorteile von Malleable-Jobs nicht, sondern behandelt Malleable-Jobs wie Rigid-Jobs. Zum Starten eines Malleable-Jobs wird, anstatt der erforderlichen Nodeanzahl der Rigid-Jobs, eine bevorzugte Nodeanzahl verwendet, die vom Benutzer bei der Eingabe zusätzlich übergeben werden muss. Ein Durchlauf des Scheduling-Algorithmus wird gestartet, wenn ein neuer Job zu *q_jobs* hinzugefügt oder ein laufender Job beendet wird.

2.3. Malleable-Basisalgorithmus

Als Grundlage für die weiterentwickelten Scheduling-Algorithmen in dieser Arbeit wird ein kürzlich veröffentlichter Scheduling-Algorithmus für Malleable-Jobs [4]

verwendet. Jeder Durchlauf dieses *Malleable-Basisalgorithmus* besteht aus drei Schritten:

1. **Backfilling:** *q_jobs* mit freien Nodes durch Backfilling starten.
2. **Shrink:** Nodes von laufenden Malleable-Jobs entfernen und zum Starten neuer Jobs verwenden.
3. **Expand:** Freie Nodes den laufenden Malleable-Jobs zuweisen.

Die initiale Nodeanzahl beim Start eines Malleable-Jobs ist von der gewählten Strategie (Siehe Abschnitt 3.1) abhängig. Rigid-Jobs werden immer mit ihrer erforderlichen Nodeanzahl gestartet. Die Reihenfolge, in der laufende Malleable-Jobs (*rm_jobs*) geschrumpft (Schritt 2) oder erweitert (Schritt 3) werden, wird durch den *Stellenwert* der Jobs festgelegt. In Abschnitt 3.1 werden dafür drei verschiedene Formeln zur Berechnung des Stellenwertes angegeben, allgemein haben Jobs mit zunehmender Nodeanzahl einen höheren Stellenwert.

Im ersten Schritt werden die Jobs aus *q_jobs* durch Backfilling gestartet. In der Veröffentlichung [4] wird eine andere Backfilling-Variante verwendet, die Jobs aus *q_jobs* vorzieht, ohne *q_head* zu berücksichtigen. Um die Fairness zu verbessern wird in dieser Arbeit stattdessen Easy-Backfilling (2.2) verwendet. Falls nach Schritt 1 noch *q_jobs* vorhanden sind, werden im zweiten Schritt Nodes aus *rm_jobs* entfernt (Shrink), um mit diesen Nodes weitere Jobs aus *q_jobs* zu starten. Jeder Job aus *q_jobs* wird der Reihe nach betrachtet und gestartet, wenn die benötigte Nodeanzahl aus *rm_jobs* entfernt werden kann. Sortiert nach ihren

Stellenwerten werden die Jobs aus *rm_jobs* geschrumpft. Dabei wird der Job mit dem höchsten Stellenwert zuerst geschrumpft. Erst wenn dieser Job nicht weiter geschrumpft werden kann, wird der nächste Job geschrumpft.

Sind nach Schritt 1 und 2 noch freie Nodes verfügbar, werden diese in Schritt 3 den Jobs aus *rm_jobs* zugewiesen. Dafür werden die Jobs aus *rm_jobs* nach ihrem Stellenwert sortiert. Der Malleable-Job mit dem niedrigsten Stellenwert aus *rm_jobs* erhält zuerst die freien Nodes. Erst wenn dieser Job nicht weiter erweitert werden kann, wird der nächste Job erweitert.

3. Anpassungen des Malleable-Basisalgorithmus

Im Rahmen dieser Bachelorarbeit wurden verschiedene Scheduling-Algorithmen auf Basis des Malleable-Basisalgorithmus (2.3) entwickelt.

3.1. Nodeanzahl & Stellenwert

Diese Anpassung beeinflusst die Anzahl der Nodes, die den Jobs zugewiesen werden. Es wurden drei Varianten entwickelt, um die Anzahl der Nodes, die einem Job hinzugefügt, entfernt oder beim Start zugewiesen werden, anzupassen. Dabei wird auch die Formel für die Berechnung des Stellenwertes verändert, so dass die Algorithmus-Variante in den Schritten 2 und 3 die angepasste Nodeanzahl besser berücksichtigt.

Minimale Nodeanzahl

Beim Start eines Jobs wird die minimale Nodeanzahl des Jobs zugewiesen. Der Stellenwert eines Jobs wird durch die Anzahl an Nodes bestimmt, die zusätzlich zur minimalen Nodeanzahl zugewiesen wurden.

$$ranking_{Job} = current\_nodes_{Job} - min\_nodes_{Job}$$

Diese Variante zur Berechnung des Stellenwertes wurde auch in der Veröffentlichung des Malleable-Basisalgorithmus [4] untersucht.

Bevorzugte Nodeanzahl

Durch die Verwendung einer bevorzugten Anzahl von Nodes für Malleable-Jobs konnte bereits in anderen Arbeiten eine Verbesserung des Scheduling-Algorithmus erreicht werden [5]. Ziel dieser Variante ist es, möglichst vielen Jobs ihre bevorzugte Nodeanzahl zuzuweisen.

Wartende Jobs werden mit ihrer bevorzugten Nodeanzahl gestartet. Sind dafür nicht genügend Nodes verfügbar, wird der Job mit allen verfügbaren Nodes gestartet, wenn mindestens die minimale Nodeanzahl verfügbar ist.

Im Gegensatz zum Malleable-Basisalgorithmus wird das Schrumpfen zum Starten eines Jobs in zwei Durchläufe unterteilt. In jedem Durchlauf werden die Jobs aus *rm_jobs* nach ihrem Stellenwert sortiert und der Job mit dem höchsten Stellenwert wird zuerst geschrumpft. Erst wenn dieser Job nicht weiter geschrumpft werden kann, wird der nächste Job geschrumpft. Im ersten Durchlauf werden zum Starten zuerst die Nodes entfernt, die den Jobs aus *rm_jobs* zusätzlich zur bevorzugten Nodeanzahl zugewiesenen sind. Dadurch werden alle Malleable-Jobs zuerst nur auf die bevorzugte Nodeanzahl geschrumpft. Wenn die überschüssigen Nodes nicht zum Starten ausreichen, werden im zweiten Durchlauf die Malleable-Jobs auch auf weniger als die bevorzugte Nodeanzahl geschrumpft.

Auch in Schritt 3 werden zunächst alle Jobs aus *rm_jobs* auf die bevorzugte Nodeanzahl erweitert. Erst wenn alle Jobs die bevorzugte Node zugewiesen haben, werden die Jobs in einem zweiten Durchlauf bis zur maximale Nodeanzahl erweitert.

Zur Bestimmung des Stellenwerts wird die Differenz von den zugewiesenen Nodes zur bevorzugten Nodeanzahl verwendet.

$$ranking_{Job} = current\_nodes_{Job} - prefered\_nodes_{Job}$$

Durchschnittliche Nodeauslastung

Ziel dieser Variante ist die gleichmäßige Verteilung der Nodes auf alle laufenden Malleable-Jobs. Deshalb wird beim Schrumpfen/Erweitern der aus *rm_jobs* ausgewählte Job nach jeder einzelnen Nodeveränderung neu bestimmt. Dies soll verhindern, dass nur ein Job geschrumpft/erweitert wird, sondern dass alle Jobs aus *rm_jobs* gleichmäßig geschrumpft/erweitert werden.

Der Stellenwert eines Jobs wird durch seine prozentuale Node-Auslastung bestimmt. Sie gibt an, zu wie viel Prozent die Spannweite der Nodes eines Jobs belegt sind. Bei 0 ist die minimale, bei 1 ist die maximale Nodeanzahl zugeordnet. Diese prozentuale Angabe soll eine bessere Vergleichbarkeit von Jobs mit unterschiedlicher Nodeanzahl ermöglichen.

$$ranking_{Job} = \frac{current\_nodes_{Job} - min\_nodes_{Job}}{max\_nodes_{Job} - min\_nodes_{Job}}$$

3.2. Umverteilung von zugewiesenen Nodes

In der Veröffentlichung des Malleable-Basisalgorithmus [4] wird angenommen, dass eine sofortige Zuweisung von Nodes von einem Job zu einem anderen Job möglich ist. In der Praxis kann eine sofortige Umverteilung zu Inkonsistenzen in den Daten oder zum Verlust von Zwischenergebnissen von entfernten Nodes führen. Um dieses Problem zu lösen, werden Malleable-Jobs in mehrere Berechnungsphasen [6] unterteilt. Zwischen diesen Berechnungsphasen werden die Zwischenergebnisse der zu entfernenden Nodes umverteilt und die verbleibenden Berechnungen auf neue Nodes umverteilt.

Um dennoch eine Umverteilung zu ermöglichen, wurde die Umverteilung angepasst, um die Verzögerung bei der Freigabe der Nodes zu berücksichtigen. Es wurden drei verschiedene Varianten der Umverteilung entwickelt, die grundsätzlich gleich aufgebaut sind:

In Schritt 2 werden weiterhin die Jobs aus *rm_jobs* ermittelt, die geschrumpft werden, um einen Job aus *q_jobs* zu starten. Da jedoch eine direkte Umverteilung nicht möglich ist, werden stattdessen die zu schrumpfenden Jobs aus *rm_jobs* angewiesen, ihre Nodes zum Ende der Berechnungsphase freizugeben. Zusätzlich wird in einer Vereinbarung festgehalten, welche Nodes dem zu startenden Job aus *q_jobs* zugewiesen werden sollen. Bis zur Umsetzung der Vereinbarung werden die verwendeten Nodes *p_nodes* und die zu startenden Jobs mit geplanten Umverteilungen *p_jobs* in den Schritten 1-2 nicht mehr berücksichtigt.

Um die Jobs aus *p_jobs* zu starten, werden zu Beginn eines Algorithmus-Durchlaufs in einem Schritt 0 die Vereinbarungen betrachtet. Dabei wird versucht, die Jobs aus *p_jobs* mit den freien Nodes aus *p_nodes* zu starten. Die genaue Zuordnung der entfernten Nodes zu den Jobs unterscheidet die drei verschiedenen Varianten.

Umverteilung durch direkte Vereinbarungen

Um das ursprünglich beschriebene Verhalten des Malleable-Basisalgorithmus (2.3) nachzubilden, wurde eine Umverteilung von Nodes über direkte Vereinbarungen implementiert. Dazu wird in Schritt 0 ein Job aus *p_jobs* genau mit seinen vereinbarten Nodes gestartet, jedoch erst wenn alle Nodes frei geworden sind. Dieses Verfahren ermöglicht eine exakte Zuordnung der Nodes wie sie in der ursprünglichen Veröffentlichung beschrieben wurde, führt aber zu einer zusätzlichen Wartezeit, weil die wartenden Jobs auf die Freigabe der neu zugewiesenen Nodes warten.

Umverteilung durch flexible Vereinbarungen

Um die Wartezeit bis zur Freigabe von der Nodes zu verkürzen, wurde die Umverteilung durch direkte Vereinbarungen (3.2) flexibler gestaltet. Beginnend mit der ältesten Vereinbarung wird in Schritt 0 versucht die Vereinbarung umzusetzen. Dabei können die Vereinbarungen neben den eigenen Nodes auch freie Nodes aus anderen Vereinbarungen zum Start verwenden. Werden freie

Nodes anderer Vereinbarungen verwendet, so erhält die verwendete Vereinbarung im Gegenzug eine noch nicht freie Node dieser Vereinbarung. Dadurch werden die direkten Vereinbarung aufgeweicht, ohne die ursprüngliche Umverteilung vollständig zu überschreiben.

Umverteilungspool

Durch die Verwendung eines Umverteilungspools kann bei der Umverteilung auf Vereinbarungen verzichtet werden. Statt einer Vereinbarung werden in Schritt 2 die durch Schrumpfen zu startenden Jobs und die dafür zu verwendenden Nodes in einem Pool gesammelt. In Schritt 0 wird versucht, die Jobs aus dem Umverteilungspool, sortiert nach dem Pool-Eintrittsalter, zu starten. Dabei werden sowohl die Nodes aus dem Umverteilungspool als auch die Nodes, die nach Beendigung eines Jobs frei werden, zum Starten verwendet. Stehen nicht genügend freie Nodes zum Starten zur Verfügung, wird der nächste Job aus dem Umverteilungspool betrachtet. Dies hat zur Folge, dass es keine feste Umverteilung von Nodes gibt und freie Nodes schnell wartenden Jobs zugewiesen werden. Jobs oder Nodes werden in den Schritten 1-3 nicht berücksichtigt, wenn sie Teil des Pools sind.

3.3. Übersicht der implementierten Algorithmen

Insgesamt wurden für zwei verschiedene Aspekte des Algorithmus jeweils drei verschiedene Strategien implementiert. Aus den unterschiedlichen Kombinationen dieser Strategien ergeben sich 9 Scheduling-Algorithmen. Diese Algorithmus-Varianten werden aufgrund der verwendeten Strategien benannt:

	Minimale Nodeanzahl	**Bevorzugte Nodeanzahl**	**Durchschnittliche Nodeauslastung**
direkte V.	min_agree	pref_agree	avg_agree
flexible V.	min_flex	pref_flex	avg_flex
Umv.-Pool	min_pool	pref_pool	avg_pool
	backfill		

Tabelle 3.1.: Implementierte Algorithmen

Das beschriebene Verhalten des Malleable-Basisalgorithmus (2.3) wird vom Algorithmus *min_agree* nachgebildet. Die minimale Nodeanzahl als Strategie für die *Nodeanzahl & Stellenwert* wurde bereits in der Veröffentlichung [4] untersucht und die verwendete direkte Vereinbarung wurde speziell für die Nachbildung der theoretischen Umverteilung entwickelt.

Neben den obigen Varianten wurde auch der Easy-Backfilling-Algorithmus aus Abschnitt 2.2 implementiert.

4. Simulationsumgebung

Zur Evaluierung der Scheduling-Algorithmen (3.3) wurde deren Verhalten simuliert. Dieses Kapitel beschreibt die Simulationsumgebung, gibt einen Überblick über die Konfiguration und beschreibt die Generierung der für die Simulation verwendeten Jobmenge.

4.1. ElastiSim

Die Simulationen wurden mit dem Batchsystem-Simulator *ElastiSim* [7] durchgeführt, der neben Rigid-Jobs auch Malleable-Jobs simulieren kann. Um HPC-Systeme zu simulieren, verwendet ElastiSim das speziell für die Simulation von verteilten Systemen entwickelte Simulationsframework SimGrid [8]. Zur Entwicklung der Scheduling-Algorithmen stellt ElastiSim eine Python-Schnittstelle bereit, die im Rahmen dieser Arbeit erweitert wurde, um die bevorzugte Anzahl von Nodes und die geschätzte Rechenzeit eines Jobs zu verwenden. Zur Verwendung dieser Schnittstelle wurden die entwickelten Scheduling-Algorithmen als Python-Skripte implementiert.

4.2. Konfiguration von ElastiSim

Jeder Scheduling-Algorithmus wird auf einem künstlichen Cluster mit verschiedenen generierten Job-Mengen simuliert. Der benötigte Rechenaufwand

eines Jobs ($flop_{total}$) wird als Anzahl von Fließkommaoperationen ($FLOP$) angegeben. Die für die Simulationen verwendeten Konfigurationen werden im folgenden Abschnitt näher beschrieben.

Scheduler-Konfiguration

In dieser Arbeit startet der Scheduler alle 60 Sekunden einen neuen Durchlauf des Scheduling-Algorithmus. Zusätzlich wird ein neuer Durchlauf ausgeführt, wenn ein Job gestartet oder beendet wird. Um auf mögliche Änderungen der zugewiesenen Nodes eines Malleable-Jobs schnell reagieren zu können, wird zudem ein neuer Durchlauf gestartet, wenn ein Malleable-Job eine Berechnungsphase beendet.

Cluster-Konfiguration

Für alle Simulationen wurde ein künstliches Cluster mit 128 identischen Nodes verwendet. Jede Node hat eine Rechenleistung von einem TeraFLOP pro Sekunde.

Applikationsmodell-Konfiguration

ElastiSim verwendet ein *Applikationsmodell* zur mathematischen Modellierung der benötigten FLOP eines Jobs. Das verwendete Applikationsmodell berechnet mit Hilfe der Formel 4.1 die FLOP-Anzahl eines Jobs, die pro Node innerhalb einer Berechnungsphase des Jobs benötigt wird.

$$flop_{phase} = \frac{flop_{total}}{phases_{total}} * \frac{1}{speedup} \tag{4.1}$$

Die FLOP-Anzahl ($flop_{phase}$) eines Jobs errechnet sich aus $flop_{total}$ dividiert durch die Anzahl der Berechnungsphasen $phases_{total}$ und dem *Speedup* (Formel 4.2). Der Speedup gibt die erwartete Beschleunigung eines Jobs an, die ein Job für die zugewiesene Nodeanzahl erreicht. In der Praxis steigt dieser Speedup jedoch nicht linear an, da Teile der Berechnungen eines Jobs nicht parallelisierbar sind und ein zusätzlicher Overhead durch die Netzwerkkommunikation entsteht. In dieser Arbeit wird das Beschleunigungsverhalten eines Jobs mit dem Amdahlschen Gesetz [9] modelliert.

$$speedup = \frac{1}{s + \frac{p}{nodes}} \tag{4.2}$$

Die daraus resultierende Formel 4.2 berechnet den Speedup für einen Job, der aus einem parallelisierbaren Anteil p und einem seriellen Anteil s mit $p+s = 1$ besteht und mit *nodes* vielen Nodes beschleunigt wird. Abbildung 4.1 zeigt beispielhaft das Speedup-Verhalten für $p = 0,99$.

Job-Konfiguration

Die zu simulierenden Jobs werden in Form einer Liste an den Simulator übergeben. Für jeden Job wird der Jobtyp (Rigid oder Malleable) und der Eingangszeitpunkt angegeben. Zusätzlich werden Nodeanzahlen angegeben, bei einem Rigid-Job die Anzahl der benötigten Nodes, bei einem Malleable-Job

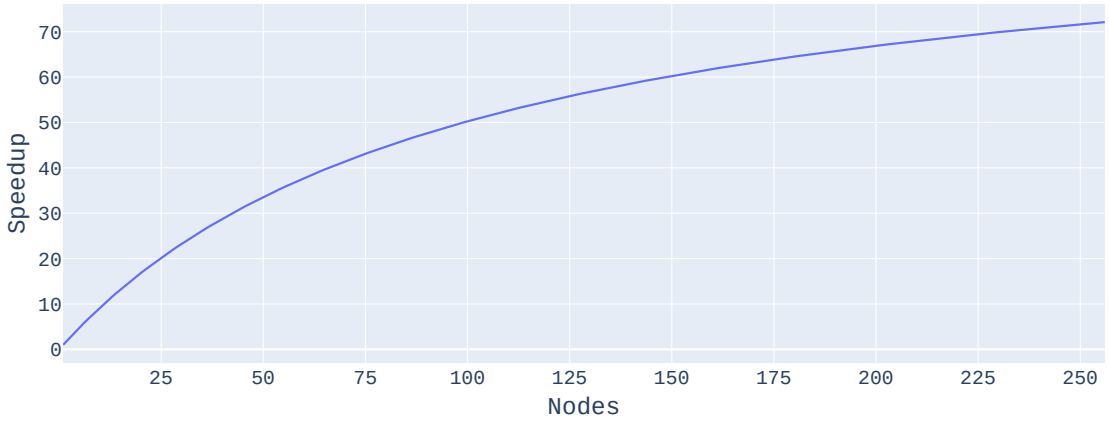

Abbildung 4.1.: Speedup eines Jobs mit parallelisierbarem Anteil p = 0,99

die Ober-/Untergrenze sowie die bevorzugte Anzahl an Nodes. Für die Werte des Job-Applikationsmodells werden die Anzahl der Berechnungsphasen, der parallelisierbare Berechnungsanteil und die benötigte FLOP-Anzahl für die Berechnung des Jobs angegeben.

4.3. Generierung der Jobs

Durch die Verwendung unterschiedlicher Zufallsgenerator-Seeds bei der Generierung der Jobs und durch unterschiedliche Verhältnisse von Malleable- zu Rigid-Jobs wurden verschiedene Jobmengen für die Auswertung wie folgt generiert:

Die Jobs werden solange generiert, bis die Gesamtdurchlaufzeit aller Jobs, also die Zeit die das HPC-System für die Berechnungen aller Jobs benötigt, 30 Tage beträgt. Dabei wird angenommen, dass keine Verzögerungen durch

das Scheduling auftreten und jeder Job mit seiner bevorzugten/erforderlichen Nodeanzahl rechnet.

Bei der Generierung wird zunächst die Eingangszeit des Jobs innerhalb dieser 30 Tage zufällig bestimmt. Anschließend wird der parallelisierbare Anteil p zufällig aus den Werten {0,95, 0,955, ... , 0,99, 0,995, 0,999} gewählt. Ausgehend vom parallelisierbaren Anteil werden die Werte für die minimale, maximale und bevorzugte Anzahl von Nodes bestimmt. Dazu wird die Effizienz des Jobs betrachtet, die das Verhältnis vom erwarteten Speedup zur zugewiesenen Nodeanzahl darstellt (Formel 4.3).

$$efficiency = \frac{speedup}{nodes} \tag{4.3}$$

Für die minimale Anzahl an Nodes wird die größte Zweierpotenz gewählt, die noch einen Effizienzwert von über 95% hat. Für die bevorzugte Anzahl an Nodes wird die größte Zweierpotenz gewählt, die noch einen Effizienzwert von über 80% hat. Für die maximale Anzahl an Nodes wird die größte Zweierpotenz gewählt, die einen Effizienzwert von über 50% hat. Diese Grenzwerte für die Effizienz wurden aus einer Veröffentlichung [5] entnommen, in der ebenfalls eine bevorzugte Nodeanzahl für Malleable-Jobs untersucht wurde.

Diese Effizienzgrenzen sind beispielhaft für $p = 0,99$ in Abbildung 4.2 dargestellt. Für $p = 0,99$ ergeben sich folgende auf Zweierpotenzen gerundete Nodeanzahlen: eine minimale Nodeanzahl von 4, eine bevorzugte Nodeanzahl

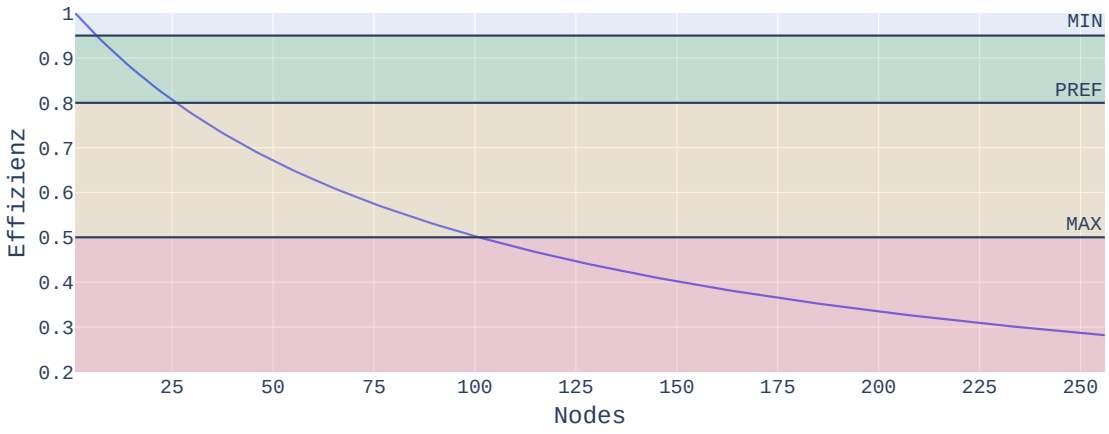

Abbildung 4.2.: Effizienzgrenzen für p = 0,99

von 16 und eine maximale Nodeanzahl von 64. Bei der Generierung eines Rigid-Jobs wird für die erforderliche Nodeanzahl die größte Zweierpotenz mit einem Effizienzwert von über 80% gewählt. Diese Nodeanzahl für Rigid-Jobs wurde so gewählt, dass sie der bevorzugten Nodeanzahl eines Malleable-Jobs mit gleichem Anteil p entspricht.

Ausgehend von der bevorzugten/erforderlichen Nodeanzahl eines Jobs wird die Anzahl der FLOP bestimmt. Die benötigte FLOP-Anzahl eines Jobs liegt zwischen 5 TeraFLOP und 400 PetaFLOP und steigt proportional zur bevorzugten/erforderlichen Nodeanzahl des Jobs. Diese Werte wurden experimentell ermittelt, so dass $\sim$ 2000 Jobs für den Zeitraum von 30 Tagen generiert werden. Abschließend wird die Anzahl der Berechnungsphasen für den Job bestimmt, so dass eine Berechnungsphase mit der bevorzugter Nodeanzahl eine Minute dauert.

4.4. Simulationseingaben

Unterschiedliche Werte für das Verhältnis von Rigid- zu Malleable-Jobs und der bei der Job-Generierung verwendete Seed erzeugten verschiedene Jobmengen. Folgende Werte wurden bei der Generierung verwendet:

Malleable %	0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%
Seed	S0, S1, S2, S3, S4, S5, S6, S7, S8, S9

Tabelle 4.1.: Simulationsparamenter

Daraus ergeben sich $11 * 10 = 110$ verschiedene Jobmengen. Jede dieser Jobmengen wurde mit jedem der 10 implementierten Scheduling-Algorithmen simuliert, so dass insgesamt $110 * 10 = 1100$ Simulationen durchgeführt wurden.

5. Simulationsergebnisse und Auswertung

5.1. Simulationsergebnisse

Jeder Datenpunkt ist der durchschnittliche Wert aller Simulationen (Seeds *S0-S9*), die für den angegebenen Algorithmus und Malleable-Anteil durchgeführt wurden. Die folgenden Abbildungen 5.1-5.7 zeigen die Änderungen der Metriken für jede Algorithmus-Variante für verschiedene Anteile von Malleable-Jobs. Bei einem Malleable-Anteil von 0% sind alle Algorithmus-Varianten identisch mit dem Easy-Backfilling-Algorithmus, der als Referenzwert durch eine rote horizontale Linie dargestellt ist.

Gesamtlaufzeit aller Jobs

Die Gesamtlaufzeit gibt die Zeit vom Einreichen des ersten Jobs bis zum Berechnungsende aller Jobs an. Eine kürzere Gesamtlaufzeit deutet auf eine effizientere Verarbeitung der eingereichten Jobs hin. Wie bereits erwähnt wurden die eingereichten Jobmengen der Simulationen so generiert, dass die optimale theoretische Gesamtlaufzeit 720 Stunden (30 Tage) beträgt.

Easy-Backfilling (2.2) benötigt eine Gesamtlaufzeit von 836,5 Stunden. Durch den Einsatz von Malleable-Jobs verbessert sich dieser Wert auf eine Laufzeit von 724 Stunden bei einem Malleable-Anteil von 100%, was einer um 13,4% schnelleren Laufzeit entspricht. Insbesondere die ersten 10-40% bringen deutliche

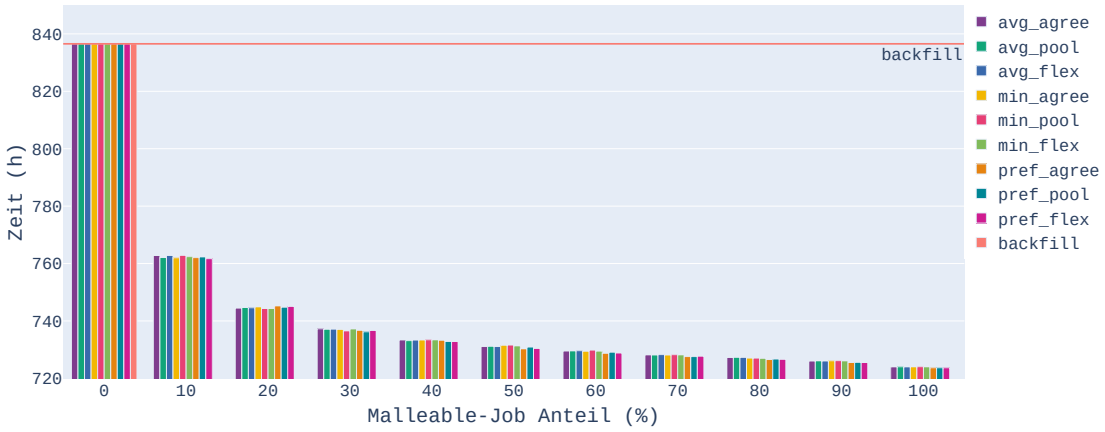

Abbildung 5.1.: Gesamtlaufzeit

Verbesserungen, ein Malleable-Anteil von 10% reduziert die Laufzeit bereits um 8,7%. Ab einem Malleable-Anteil von 40% flacht diese zusätzliche Verbesserung stark ab. Zwischen den verschiedenen Malleable-Algorithmen gibt es keine signifikanten Unterschiede. Bei der *Umverteilungsart* ist keine bessere Variante erkennbar. Lediglich bei *Nodeanzahl & Stellenwert* zeigt sich eine leicht bessere Laufzeit der *pref*-Varianten, allerdings unterscheiden sich die Werte nur um weniger als 1% von den anderen Algorithmus-Varianten.

Durchschnittliche Systemauslastung

Die Systemauslastung gibt den prozentualen Anteil der verwendeten Nodes über den gesamten Zeitraum an. Eine höhere Systemauslastung lässt auf eine bessere Nutzung aller Nodes schließen.

Dabei erreichte Easy-Backfilling (2.2) eine Systemauslastung von 86,02%. Die verschiedenen Malleable-Algorithmen unterscheiden sich ebenfalls nicht

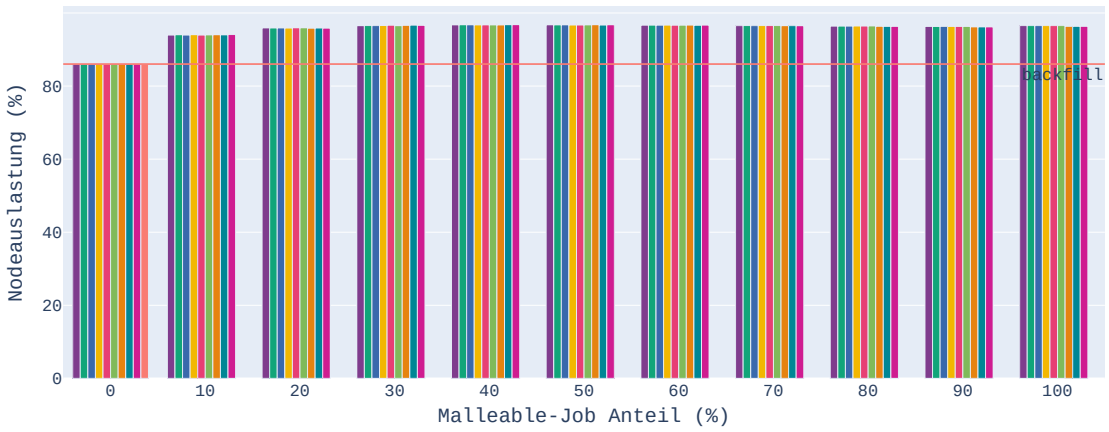

Abbildung 5.2.: Durchschnittliche Systemauslastung

signifikant, allgemein konnte die Systemauslastung auf eine maximale Systemauslastung von 96,5% gesteigert werden. Auch hier sorgen bereits kleine Malleable-Anteile für deutliche Leistungssteigerungen. Ab einem Malleable-Anteil von 30% bleibt die Auslastung gleich, mit Abweichungen von weniger als 0,25%.

Durchschnittliche Wartezeit/Durchlaufzeit eines Jobs

Die Zeit von der Eingabe des Jobs bis zum Beginn der Ausführung ist die Wartezeit eines Jobs. Wartezeit und Rechenzeit zusammen ergeben die Durchlaufzeit, die für den Anwender von besonderem Interesse ist, da bei einer geringen durchschnittlichen Durchlaufzeit die Berechnungsergebnisse der Jobs schneller zur Verfügung stehen.

Beide Zeiten konnten durch den Einsatz von Malleable-Jobs verbessert werden. Bereits bei einem Malleable-Anteil von 10% sinkt die durchschnittliche Wartezeit um 64,8%, bei einem Malleable-Anteil von 100% um bis zu 97,3%. Die

Abbildung 5.3.: Durchschnittliche Wartezeit

Abbildung 5.4.: Durchschnittliche Durchlaufzeit

Durchlaufzeit sinkt bei einem Anteil von 10% um 63,5%, bei einem Anteil von 100% um 93,6%. Im Vergleich zu den anderen Algorithmus-Varianten weisen die *pref*-Varianten mit steigendem Malleable-Anteil eine um wenige Minuten geringere Warte-/Durchlaufzeit auf.

Durchschnittliche Rechenzeit eines Jobs

Der Zeitraum der Berechnung eines Jobs wird als Rechenzeit bezeichnet. Im Gegensatz zu den vorherigen Metriken verschlechtert sich die Rechenzeit

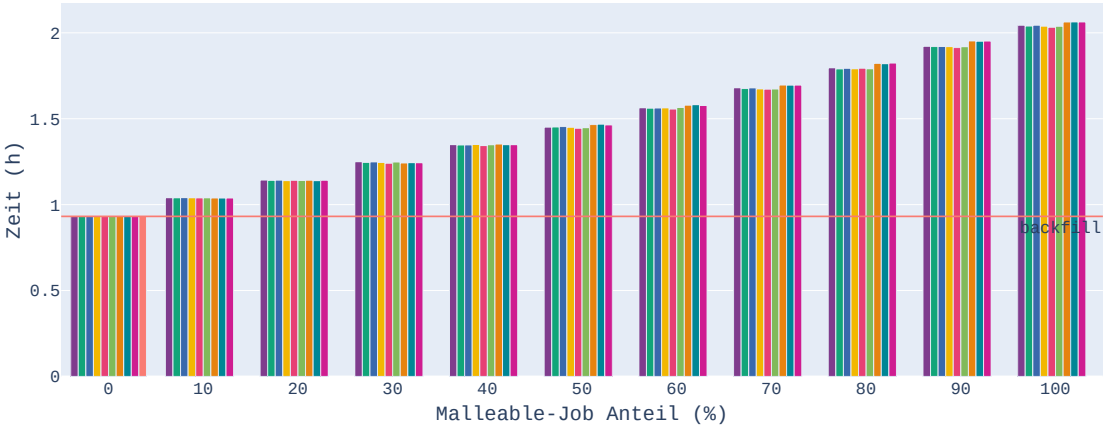

Abbildung 5.5.: Durchschnittliche Rechenzeit

kontinuierlich mit steigendem Malleable-Anteil. Die ursprüngliche Rechenzeit von 0,93 Stunden erhöht sich bei einem Malleable-Anteil von 100% auf 2,06 Stunden, eine Steigerung um 121%. Auch hier zeigen sich keine signifikanten Unterschiede zwischen den Malleable-Algorithmen, jedoch weisen die *pref*-Varianten mit steigendem Malleable-Anteil eine etwas längere Rechenzeit auf.

Durchschnittliche Expand/Shrink-Events pro Malleable-Job

Diese Metriken beschreiben die durchschnittliche Anzahl an Expand/Shrink-Events pro Malleable-Job. Ein Shrink-Event zeigt an, dass ein Job aus *rm_jobs* angewiesen wurde, ein oder mehrere Nodes zum Ende der Berechnungsphase freizugeben. Ein Expand-Event zeigt an, dass

einem Job aus *rm_jobs* ein oder mehrere Nodes zugewiesen wurde. Die

Abbildung 5.6.: Durchschnittliche Expand-Events pro Malleable-Job

Abbildung 5.7.: Durchschnittliche Shrink-Events pro Malleable-Job

Anzahl an Events pro Malleable-Job nimmt mit steigendem Malleable-Anteil stetig ab. Diese Abnahme ist darauf zurückzuführen, dass mit steigendem Malleable-Anteil zwar mehr Events auftreten, sich diese aber auf mehr Malleable-Jobs verteilen. Die *pref*-Varianten haben aufgrund der zwei nacheinander durchgeführten Expand/Shrink-Durchläufe (3.1) etwas mehr Events als die

anderen Algorithmus-Varianten. Dieser Unterschied ist bei den Expand-Events ausgeprägter als bei den Shrink-Events. Die meisten Events pro Malleable-Job treten bei einem Malleable-Anteil von 10% auf. Dabei verwenden die *pref*-Varianten durchschnittlich 3,8 Expand-Events, während die anderen Varianten durchschnittlich 3,44 Expand-Events pro Job verwenden. Bei den Shrink-Events verwenden die *pref*-Varianten mit 3,7 Shrink-Events pro Job etwas mehr Events als die anderen Varianten mit 3,62 Shrink-Events.

5.2. Auswertung

Die Simulationsergebnisse zeigen, dass die entwickelten Scheduling-Algorithmen alle untersuchten Metriken des HPC-Systems mit Ausnahme der durchschnittlichen Rechenzeit durch den Einsatz von Malleable-Jobs verbessert haben. Insbesondere durch die verringerte Durchlaufzeit der Jobs erhalten die Nutzer schneller die Ergebnisse ihrer eingereichten Jobs. Diese Verbesserung der Durchlaufzeit ist auf die stark reduzierten Wartezeiten zurückzuführen, die die verlängerten Rechenzeiten mehr als ausgleichen.

Diese Vorteile treten bereits bei einem geringen Malleable-Anteil auf. Ein Malleable-Anteil von 10% erreicht bereits ca. 65% der maximalen Malleable-Verbesserung, ein Malleable-Anteil von 40% sogar über 90%. Eine Umfrage des US Exascale Computing Project [10] zeigt, dass 39% der Jobs eine Änderung der Prozessoranzahl unterstützen könnten, wenn *Checkpoints*

implementiert würden, von denen aus die Jobs neu gestartet werden. Auch wenn durch das Neustarten ein zusätzlicher Overhead entsteht ist dieses Checkpoint-Verhalten vergleichbar mit dem in dieser Arbeit verwendeten Job-Modell aus Berechnungsphasen, weshalb ein Malleable-Anteil von 40% in der Praxis denkbar wäre. Damit könnte die Gesamtdurchlaufzeit aller Jobs um 12,3% und die durchschnittliche Durchlaufzeit eines Jobs um 87,1% reduziert werden.

Darüber hinaus hat sich gezeigt, dass im Durchschnitt wenige Expand/Shrink-Events auftreten. Bei dem realisierbaren Malleable-Anteil von 40% treten bereits weniger als zwei Events pro Malleable-Job auf. Daher ist der zusätzliche Overhead von Events, der bei der Umverteilung der Nodes durch den Neustart des Jobs entsteht, gering. Die Ergebnisse weiterer Simulationen, die diesen Overhead berücksichtigen, sollten sich nicht wesentlich von den beschriebenen Simulationsergebnissen unterscheiden.

Die verschiedenen Algorithmus-Varianten erzielen vergleichbare Verbesserungen. Obwohl die *pref*-Varianten in den meisten Fällen etwas besser abschneiden, ist unklar, ob diese Verbesserung die zusätzliche Angabe einer bevorzugten Nodeanzahl durch den Nutzer rechtfertigt. Die vermuteten Vorteile des Umverteilungspools als *Umverteilungsart* wurden durch die Simulationsergebnisse nicht bestätigt. Entgegen der Annahme schneidet der Umverteilungspool etwas schlechter als die anderen *Umverteilungsarten* ab. Die Unterschiede sind jedoch zu gering und schwankend um eine fundierte Aussage treffen zu können.

6. Verwandte Arbeiten

Nach aktuellem Stand der Technik verwendet nur ein kleiner Teil aller HPC-Systeme Malleable-Jobs. Viele Scheduler unterstützen nativ keine Malleable-Jobs und bei der Entwicklung von Malleable-Jobs entsteht ein Mehraufwand für die Entwickler. Die Entwicklung von Schedulern für Malleable-Jobs und verschiedene Ansätze für eine einfachere Entwicklung von Malleable-Jobs werden in den letzten Jahren vermehrt untersucht.

Mehrere Studien haben gezeigt, dass Malleable-Jobs zu Verbesserungen im HPC führen [11, 12, 13]. Vergleichbar mit den Ergebnissen dieser Arbeit konnten kürzere Durchlaufzeiten und Wartezeiten der Jobs, sowie Verbesserungen bei der Systemauslastung und beim Energieverbrauch erzielt werden. Zur Unterstützung von Malleable-Scheduling-Algorithmen wurden zudem Erweiterungen für den bekannten Scheduler *Slurm* [12, 13] entwickelt.

Zur Vereinfachung der Entwicklung von Malleable-Jobs wurde die Verwendung von Checkpoints, die eine Umverteilung der Nodes durch einen Job-Neustart ermöglichen [14], untersucht. Außerdem werden Ansätze entwickelt, die ohne einen Neustart eine Umverteilung der Nodes ermöglichen [12, 6].

Neben dem Einsatz von Malleable-Jobs wird der Einsatz von *Evolving*-Jobs untersucht [1]. Die selbstständig angeforderte Umverteilung der Nodes erlaubt es Evolving-Jobs ihre Nodeanzahl an die momentane Berechnung des Jobs

anzupassen. Dadurch könnte ein serieller Berechnungsteil nur eine Node verwenden, während zu einem stark parallelisierbaren Berechnungsteil viele Nodes zugewiesen werden. Zur Unterstützung von Evolving-Jobs werden allerdings Anpassungen des Schedulers benötigt, um auf die Änderungen der geforderten Nodeanzahl eines Evolving-Jobs reagieren zu können.

7. Zusammenfassung

In dieser Arbeit wurde ein kürzlich veröffentlichter Scheduling-Algorithmus für Malleable-Jobs weiterentwickelt. Es wurden drei Varianten für *Nodeanzahl & Stellenwert* eines Jobs (3.1), sowie drei Varianten für die *Umverteilungart* von zugewiesenen Nodes (3.2) untersucht. Das Verhalten der daraus resultierenden neun Algorithmus-Varianten wurde durch Simulationen mit künstlichen Jobmengen verglichen und anschließend anhand verschiedener Simulationsmetriken evaluiert. Bei der Auswertung der Simulationsergebnisse konnten keine signifikanten Unterschiede zwischen den Algorithmus-Varianten festgestellt werden. Zwar erzielten die *pref*-Varianten geringfügig bessere Simulationsergebnisse, diese waren jedoch zu gering, um den damit verbundenen Mehraufwand für den Anwender zu rechtfertigen.

Die Vorteile von Malleable-Jobs konnten durch Simulationen mit unterschiedlichen Anteilen von Malleable-Jobs nachgewiesen werden. Im Vergleich zu den Simulationen ohne Malleable-Jobs konnte durch den Einsatz von Malleable-Jobs die Gesamtdurchlaufzeit aller generierten Jobs verringert, die Systemauslastung deutlich erhöht und die durchschnittliche Durchlaufzeit der Jobs stark reduziert werden. Bereits mit einem geringen Anteil von Malleable-Jobs ließen sich signifikante Verbesserungen erzielen.

Aufbauend auf dieser Arbeit können verschiedene Verbesserungen der Simulationseingaben zu noch realitätsnäheren Simulationsergebnissen führen. Neben einem detaillierteren Applikationsmodell, das den Overhead der Nodeumverteilung berücksichtigt, stellt die generierte Jobmenge eine potentielle Verbesserungsmöglichkeit dar. Diese Jobmenge könnte dafür entweder auf der Basis realer Jobdaten, oder unter Verwendung mathematischer Modelle wie dem Downey-Modell [15] generiert werden.

Es sollte auch darauf hingewiesen werden, dass in dieser Arbeit nur ein HPC-System aus 128 homogenen Nodes simuliert wurde. In weiteren Arbeiten könnte untersucht werden, welche Anpassungen der Scheduling-Algorithmen für die Unterstützung heterogener-Systeme erforderlich sind. Darüber hinaus würden Simulationen von größeren homogenen HPC-Systemen Aufschluss über die Skalierbarkeit der Scheduling-Algorithmen geben.

Literatur

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

[2] A. Mu’alem und D. Feitelson. “Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling”. In: *IEEE Transactions on Parallel and Distributed Systems* (2001), S. 529–543. doi: 10.1109/71.932708.

[3] S. Srinivasan, R. Kettimuthu, V. Subramani und P. Sadayappan. “Characterization of backfilling strategies for parallel job scheduling”. In: *Proceedings. International Conference on Parallel Processing Workshop*. 2002, S. 514–519. doi: 10.1109/ICPPW.2002.1039773.

[4] D. H. Lina, S. Ghafoor und T. Hines. “Scheduling of Elastic Message Passing Applications on HPC Systems”. In: *Job Scheduling Strategies for Parallel Processing*. 2023. doi: 10.1007/978-3-031-22698-4_9.

[5] J. Posner und C. Fohry. “Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing”. In: *50th International Conference on Parallel Processing Workshop*. ICPP Workshops ’21. Lemont, IL, USA: Association for Computing Machinery, 2021. doi: 10.1145/3458744.3473361.

[6] A. Gupta, B. Acun, O. Sarood und L. V. Kalé. "Towards realizing the potential of malleable jobs". In: *International Conference on High Performance Computing (HiPC)*. 2014, S. 1–10. doi: 10.1109/HiPC.2014.7116905.

[7] T. Özden, T. Beringer, A. Mazaheri, H. M. Fard und F. Wolf. "ElastiSim: A Batch-System Simulator for Malleable Workloads". In: *Proceedings of the 51st International Conference on Parallel Processing*. ICPP '22. Bordeaux, France: Association for Computing Machinery, 2023. doi: 10.1145/3545008.3545046.

[8] H. Casanova, A. Giersch, A. Legrand, M. Quinson und F. Suter. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". In: (2014), S. 2899–2917. doi: 10.1016/j.jpdc.2014.06.008.

[9] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *AFIPS Joint Spring Conference Proceedings 30*. 1967, S. 483–485. doi: 10.1145/1465482.1465560.

[10] D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz und G. R. Vallee. "A survey of MPI usage in the US Exascale Computing Project". In: *Concurrency and Computation: Practice and Experience (CCPE)* 32.3 (2020). doi: 10.1002/cpe.4851.

[11] R. Sudarsana und C. J.Ribbens. “Combining Performance and Priority for Scheduling Resizable Parallel Applications”. In: *Parallel and Distributed Computing (JPDC)* 87 (2016), S. 55–66. doi: 10.1016/j.jpdc.2015.09.007.

[12] M. Chadha, J. John und M. Gerndt. “Extending Slurm for Dynamic Resource-Aware Adaptive Batch Scheduling”. In: *Proceedings International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2020. doi: 10.1109/HiPC50609.2020.00036.

[13] S. Iserte, R. Mayo, E. S. Quintana-Ortí und A. J. Peña. “DMRlib Easy-coding and Efficient Resource Management for Job Malleability”. In: *Transactions on Computers (TC)* (2020). doi: 10.1109/TC.2020.3022933.

[14] A. Moody, G. Bronevetsky, K. Mohror und B. R. de Supinski. “Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System”. In: *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2010, S. 1–11. doi: 10.1109/SC.2010.18.

[15] A. B. Downey. “A parallel workload model and its implications for processor allocation”. In: *Proceedings International Symposium on High Performance Distributed Computing (HPDC)*. 1997. doi: 10.1109/HPDC.1997.622368.

A. Anhang

Digitale Abgabe

Inhalt der digitalen Abgabe:

- Digitale Fassung der Arbeit und der Grafiken
- Programmcode der implementierten Scheduling-Algorithmen

  und des erweiterten ElastiSim-Interface
- Alle verwendeten Simulationseingaben und Simulationsausgaben
- Tabellen mit Werten aller untersuchten Simulationsmetriken
- Programme zur Durchführung und Auswertung der Simulationen