TECHNISCHE UNIVERSITÄT
CHEMNITZ

Suchbasierte Algorithmen für das Scheduling unabhängiger paralleler Tasks

Von der Fakultät für Informatik der
Technischen Universität Chemnitz
genehmigte Dissertation
zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

Vorgelegt von
M.Sc. Robert Dietze,
geboren am 30. April 1988 in Frankenberg.

Tag der Einreichung: 25. Oktober 2021
Tag der Verteidigung: 26. April 2022

Gutachter: Prof. Dr. Gudula Rünger
Technische Universität Chemnitz

Prof. Dr. Wolfram Hardt
Technische Universität Chemnitz

https://nbn-resolving.org/urn:nbn:de:bsz:ch1-qucosa2-790089

Gliederung der Dissertation

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

Scheduling ist ein Entscheidungsprozess, der in vielen Bereichen der Fertigungswirtschaft, des Dienstleistungssektors und der Informatik eingesetzt wird. So wird Scheduling bereits seit den 50er Jahren unter anderem verwendet, um Maschinenbelegungspläne in Produktionsbetrieben und Pläne im Transportwesen zu erstellen. Im Allgemeinen bezeichnet Scheduling die zeitliche und räumliche Zuweisung einer Menge von Aufgaben zu einer Menge von Ressourcen, so dass bestimmte Optimalitätskriterien erfüllt werden. Ende der 60er Jahre wurde mit Schedulingverfahren erstmals auch die Ausführung von Programmen auf Rechnersystemen geplant [65]. Insbesondere mit dem Aufkommen von Mehrprozessor- und Mehrkernsystemen wurden und werden Verfahren benötigt, mit denen deren Leistungsfähigkeit ausgenutzt werden kann. So ermöglichen diese Systeme eine parallele Ausführung mehrerer eigenständiger Programme oder die Ausnutzung einer internen Parallelität eines Programms. Mithilfe von Schedulingverfahren sollen die (Teil-)Programme (*Tasks*) den Prozessoren des parallelen Systems (*Ressourcen*) zugewiesen werden, so dass ein oder mehrere Optimalitätskriterien erfüllt werden. Häufig betrachtete Kriterien sind beispielsweise die Minimierung der Gesamtausführungszeit oder die Reduzierung von Leerlaufzeiten. Zunehmend rückt auch eine Senkung des Energieverbrauchs in den Fokus [61].

Ein weiterer wichtiger Einsatzbereich für das Scheduling ist das wissenschaftliche Rechnen. In diesem Bereich werden zunehmend komplexere und rechenintensivere Anwendungen eingesetzt, die mit einem wachsenden Bedarf an Rechenleistung einhergehen. Beispiele hierfür sind Simulationen naturwissenschaftlicher Prozesse wie Festigkeitsberechnungen, Strömungssimulationen oder auch geologische und astrophysikalische Prozesse. Derartige Simulationen können je nach Einsatzgebiet Untersuchungen am realen System ersetzen (z.B. Wettervorhersage) oder anpassbarer und kostengünstiger realisieren (z.B. Crash- oder Windkanaltests) [64]. Die Ausführungszeit solch komplexer und rechenintensiver Anwendungen hängt maßgeblich von der effizienten Ausnutzung der vorhandenen Rechenressourcen ab. Der Einsatz von einzelnen Parallelrechnern oder miteinander verbundenen Rechnern ermöglicht eine Reduktion der Ausführungszeit, wenn beispielsweise eine Berechnung von mehreren Recheneinheiten gemeinsam bearbeitet wird. Heutzutage sind zudem Multicore-Prozessoren weit verbreitet, die über mehrere Prozessorkerne zur parallelen Berechnung verfügen. Der Vorteil solcher Architekturen ist, dass sie über einen gemeinsamen Speicher verfügen und somit die Zeit für den Datenaustausch erheblich reduziert

wird. Viele naturwissenschaftliche Anwendungen sind für eine parallele Berechnung geeignet, da sie sich in unabhängige Teilberechnungen zerlegen lassen.

Die parallele Ausführung einer Anwendung erfordert meist dessen Formulierung in einer parallelen Programmiersprache. So ermöglichen bestimmte Programmiermodelle eine Zerlegung komplexer Anwendungen in eine Menge von Tasks. Ein Task bezeichnet die kleinste Einheit eines parallelen Programms, die beim Scheduling berücksichtigt wird. Abhängig von der Modellierung und dem Abstraktionslevel kann ein Task einem Thread, einem Prozess oder einer Menge von Prozessen entsprechen [21]. Tasks können unabhängig voneinander sein oder erst nach der Beendigung anderer Tasks gestartet werden, wenn beispielsweise Datenabhängigkeiten vorliegen. Voneinander unabhängige Tasks können auf einem parallelen System parallel zueinander ausgeführt werden. Die räumliche und zeitliche Planung einer solchen Ausführung wird als *Task-Scheduling* bezeichnet. Die kleinste Verarbeitungseinheit eines Tasks kann je nach Abstraktionsebene ein Prozessorkern, ein Prozessor, ein Rechner oder eine Menge von Rechnern sein. Für die Erstellung eines Ablaufplans (engl. Schedule) können Schedulingverfahren eingesetzt werden, die die Tasks derart den Verarbeitungseinheiten zuweisen, dass ein vorgegebenes Optimalitätskriterium erfüllt wird. Ein häufig betrachtetes Kriterium ist dabei die Gesamtausführungszeit des Schedules (engl. *makespan*). Diese bezeichnet die Zeit zwischen dem Start des ersten und der Beendigung des letzten Tasks der parallelen Anwendung.

Parallele Tasks

Für die Implementierung paralleler Anwendungen existieren verschiedene Programmiermodelle. Zwei klassische Programmiermodelle sind Task- und Datenparallelität. Im *taskparallelen Programmiermodell* lassen sich verschiedene unabhängige Programmteile (Tasks) identifizieren, die parallel zueinander ausgeführt werden können. Die Tasks können unabhängig voneinander unterschiedliche Berechnungen auf verschiedenen Daten ausführen. Dieses Programmierkonzept wird auch als MPMD (multiple program, multiple data) bezeichnet. Im Fall von *Datenparallelität* lassen sich die Daten gleichmäßig auf die Prozessoren verteilen. Die Prozessoren führen dann in jedem Schritt die gleichen Berechnungen auf dem jeweiligen Teil der Daten aus. Dieser Programmieransatz ist auch als SPMD (single program, multiple data) bekannt. Eine Möglichkeit, die Skalierbarkeit paralleler Anwendungen zu erhöhen, ist die Vereinigung beider Programmiermodelle, die als *gemischte Parallelität* bezeichnet wird [23]. In diesem Modell können unabhängige Tasks nicht nur parallel zueinander, sondern jeweils selbst parallel ausgeführt werden. Im Folgenden werden diese Tasks als *parallele Tasks* bezeichnet.

In der Literatur werden abhängig von deren Eigenschaften verschiedene Begriffe zur Bezeichnung paralleler Tasks verwendet, z.B. *multiprocessor tasks* oder *moldable tasks*. Daher wird in Abschnitt 2.1.2 eine Klassifikation paralleler Tasks vorgestellt. In dieser Arbeit werden parallele Tasks betrachtet, die auf einer beliebigen, aber

während der Ausführung konstanten Anzahl Prozessorkerne eines Parallelrechners mit gemeinsamem Speicher ausgeführt werden können. Dies sind beispielsweise Anwendungen, die unter Verwendung eines threadbasierten Programmiermodells wie OpenMP oder Pthreads implementiert wurden und auf einem Multicore-System ausgeführt werden sollen. Für parallele Tasks gilt für gewöhnlich, dass die Ausführungszeit mit steigender Anzahl Prozessorkerne abnimmt. Innerhalb der meisten parallelen Programme ist der Kommunikations- und Synchronisationsaufwand jedoch so groß, dass diese ab einem gewissen Punkt kaum noch von einer steigender Kernanzahl profitieren. Die zusätzliche Schwierigkeit beim Scheduling unabhängiger, paralleler Tasks ist es daher, einen Kompromiss zwischen einer task- und einer datenparallelen Ausführung zu finden, wofür spezifische Schedulingverfahren benötigt werden.

Heterogene parallele Systeme

Die Wahl eines geeigneten Schedulingverfahrens hängt nicht nur von der Art der Tasks, sondern auch von der Zusammensetzung der Rechenressourcen ab. So können parallele Systeme unter anderem als homogen oder heterogen klassifiziert werden. Cluster miteinander verbundener Rechner werden als *homogener Cluster* bzw. *homogenes paralleles System* bezeichnet, wenn die beteiligten Rechner identisch sind und das Verbindungsnetzwerk für alle Rechner dieselben Eigenschaften besitzt. Liegt innerhalb eines parallelen Systems keine Homogenität vor, so wird von einem *heterogenen parallelen System* gesprochen. Wirtschaftliche Aspekte und der wachsende Bedarf paralleler wissenschaftlicher Anwendungen an Rechenleistung haben in vielen Forschungseinrichtungen dazu geführt, dass die parallelen Systeme über die Jahre stetig erweitert wurden. Die so entstandenen parallelen Systeme bilden einen Zusammenschluss verschiedener homogener paralleler Systeme, deren Rechenknoten abhängig von der verfügbaren Technologie unterschiedliche Eigenschaften besitzen. Die wesentlichen Unterschiede liegen dabei in den verbauten Prozessoren, die sowohl in ihrer Leistung als auch in der Anzahl verfügbarer Prozessorkerne voneinander abweichen können. Die Heterogenität eines solchen Systems kann dazu führen, dass die Laufzeit eines Programms zwischen einzelnen Rechnern variiert und muss daher beim Scheduling beachtet werden. So ist auf heterogenen Systemen die häufig eingesetzte gleichmäßige Aufteilung der Tasks auf die Rechner keine gute Wahl, da so die Heterogenität nicht berücksichtigt wird.

Heuristische Schedulingverfahren

Das Ziel des Schedulings paralleler Tasks auf heterogene parallele Systeme besteht darin, für jeden Task den zu verwendenden Rechenknoten sowie die Anzahl Prozessorkerne zu bestimmen, die zur Ausführung genutzt werden sollen. Zusätzlich muss eine zeitliche Reihenfolge festgelegt werden, da eine gleichzeitige Ausführung

mehrerer Tasks auf einem Prozessorkern nicht zulässig ist. Eine Möglichkeit, dieses Problem zu lösen, ist der Einsatz von Schedulingverfahren. Jedoch ist bereits das Schedulingproblem für Single-Prozessor-Tasks und parallele Systeme mit zwei Prozessoren NP-schwer [21]. Für das Scheduling paralleler Tasks kommt erschwerend hinzu, dass für jeden Task die Anzahl zu verwendender Prozessoren bestimmt werden muss.

Aufgrund der hohen Komplexität des Problems beruhen die meisten Schedulingverfahren für parallele Tasks auf heuristischen Vorgehensweisen, z.B. [51, 59, 60, 74]. Für den praktischen Einsatz sollte die Ausführungszeit eines Schedulingverfahrens in einer angemessenen Relation zur erwarteten Qualität der Lösung stehen. So kann je nach Einsatzgebiet der Fokus auf eher auf einer schnellen Lösungsfindung als auf deren Qualität liegen oder umgekehrt. Heuristische Verfahren können das Finden einer optimalen Lösung zwar nicht garantieren, kommen einer optimalen Lösung aber häufig sehr nahe. Gleichzeitig arbeiten die meisten Heuristiken vergleichsweise schnell, wobei deren worst-case-Laufzeit oftmals einer Polynomzeit niedriger Ordnung entspricht. In den letzten 50 Jahren wurden zahlreiche heuristische Schedulingverfahren basierend auf unterschiedlichen Ansätzen entwickelt. Approximationsalgorithmen wurden analytisch untersucht und können Schedules ermitteln, die eine bestimmte Lösungsqualität erreichen. List-Scheduling-Verfahren konstruieren Schedules iterativ, indem sie die Tasks nacheinander in einer bestimmten Reihenfolge zuweisen. Im Gegensatz zu konstruierenden Heuristiken starten modifizierende Heuristiken mit einer bestehenden Lösung und verändern diese bis ein Abbruchkriterium erreicht wird [28]. Letzterer Ansatz entspricht einer lokalen Suche in einem Lösungsraum, das heißt der Menge aller Schedules eines gegebenen Schedulingproblems.

Die Größe des Lösungsraums für ein Schedulingproblem steigt zumeist exponentiell mit der Taskanzahl. Ein Durchsuchen des gesamten Lösungsraums ist daher oft nicht praktikabel. Ein Ansatz zur Lösungsfindung ist der Einsatz lokaler Suchverfahren oder informierter Suchstrategien [66] wie der A*-Suche [35]. Informierte Suchstrategien nutzen problemspezifisches Wissen, um Lösungen effizienter finden zu können. Lokale Suchverfahren führen ausgehend von einer Anfangslösung lokale Veränderungen an der aktuellen Lösung durch, um auf diese Weise eine bessere Lösung zu finden. Bezogen auf das Task-Scheduling entspricht das der schrittweisen Transformation eines bestehenden Schedules hin zu einem besseren Schedule bezüglich der Optimierungsfunktion. Lokale Suchverfahren, die stets den bestmöglichen Transformationsschritt durchführen, wie der Bergsteigeralgorithmus (engl. hill climbing), finden häufig nur lokale Optima. Um lokalen Optima zu entkommen, werden Verfahren benötigt, die auch schlechtere Lösungen in ihre Suche einbeziehen. Eine Kategorie derartiger Verfahren, die in den letzten Jahren verstärkt zur Lösung kombinatorischer Optimierungsprobleme eingesetzt wurden, sind sogenannte Metaheuristiken.

Metaheuristiken

Der Begriff *Metaheuristik* geht auf Fred Glover [30] zurück und bezeichnet abstrakte Lösungsverfahren, die, problemspezifisch implementiert, auf beliebige kombinatorische Optimierungsprobleme angewendet werden können. Bei diesen Verfahren handelt es sich um lokale Suchverfahren, die über Strategien zum Entkommen aus lokalen Optima verfügen und so in der Lage sind, globale Optima zu finden. Metaheuristiken starten mit einer oder mehreren Lösungen und suchen wiederholt in der Nachbarschaft der aktuellen Lösung nach einer neuen, aber nicht zwangsläufig besseren Lösung. Nachbarschaftsbeziehungen werden dabei durch kleine Veränderungen der jeweils aktuellen Lösung abgebildet. Im Wesentlichen unterscheiden sich Metaheuristiken in der Art wie Nachbarschaften gebildet und Lösungen daraus ausgewählt werden. Je nach Verfahren werden Lösungen konstruiert, modifiziert oder miteinander kombiniert, um neue Lösungen zu erhalten. So wurden zahlreiche, oftmals naturinspirierte Verfahren vorgestellt, wie die *Tabu-Suche*, *Ant Colony Optimization*, *Particle Swarm Optimization*, *Artificial Immune Systems* und *Genetische Algorithmen*. Metaheuristiken haben sich in den letzten Jahren als außergewöhnlich effizient erwiesen, so dass sie zur Lösung verschiedenster kombinatorischer Optimierungsprobleme eingesetzt werden [32]. So werden sie verbreitet für das Finden eines kürzesten Wegs eingesetzt, z.B. zur Lösung des Problems des Handlungsreisenden [52]. Aber auch in praktischen Anwendungen wie zur Navigation von Fahrzeugen oder Robotern [11, 36], dem Routing in Computernetzwerken [49] und der Chip-Herstellung [75] werden verstärkt Metaheuristiken eingesetzt.

Viele Schedulingprobleme lassen sich als kombinatorisches Optimierungsproblem formulieren, die dann mithilfe von Metaheuristiken gelöst werden können. So wurden zahlreiche Algorithmen zur Lösung verschiedener Schedulingprobleme vorgestellt, die auf Metaheuristiken basieren [40]. Diese Algorithmen wurden jedoch für das Scheduling sequentieller Tasks entwickelt und eignen sich daher nicht für das komplexere Scheduling paralleler Tasks. Nach aktuellem Stand scheint kein metaheuristisches Verfahren für das Scheduling paralleler Tasks auf heterogene parallele Systeme zu existieren.

Beiträge der Arbeit

Die vorliegende Arbeit befasst sich mit dem suchbasierten Scheduling paralleler Tasks auf heterogene parallele Systeme. Das Optimierungsziel des betrachteten Schedulingproblems ist die Minimierung des Makespans. Zu diesem Zweck werden vier Schedulingverfahren entwickelt, die auf ausgewählten inkrementellen und modifizierenden Suchstrategien basieren. So verwendet das Verfahren Heterogeneous Parallel task scheduling based on A* (HP*) eine informierte Suchstrategie basierend auf der A*-Suche. Bei der konstruierenden Vorgehensweise von HP* werden die Tasks nacheinander den Prozessorkernen der Rechenknoten zugewiesen.

Für jede Zuweisung wird dabei jede Reihenfolge der Tasks und jede mögliche Kombination von Prozessorkernen betrachtet. Auf diese Weise ist HP* in der Lage eine optimale Lösung zu finden. Des Weiteren werden eine Restkostenschätzung zur effizienteren Suche und drei Strategien zur Eingrenzung des Suchraums präsentiert.

Als Ansatz für die beiden Schedulingverfahren Multiprocessor Task Tabu Search Scheduling (MTaSS) und Simulated Annealing Multiprocessor Task Scheduling (SAMT) wurden Metaheuristiken verwendet. Beide Verfahren beginnen mit einem Schedule und transformieren diesen schrittweise, um einen Schedule mit geringerem Makespan zu finden. Dabei nutzt MTaSS die Vorgehensweise einer Tabu-Suche, bei der in jedem Schritt die Transformation durchgeführt wird, die den geringsten Makespan zur Folge hat. Um Kreisläufe und ein Feststecken in lokalen Optima zu vermeiden, werden besuchte Schedules in einer Tabu-Liste gespeichert. Diese Tabu-Liste bildet eine Art Gedächtnis und schließt bereits besuchte Schedules von der Suche aus. Der Ablauf von SAMT basiert auf der Methode der simulierten Abkühlung (Simulated Annealing). In jedem Durchlauf des Verfahrens wird eine zufällige Transformation am aktuellen Schedule durchgeführt und ein resultierender Schedule mit geringerem Makespan als neue Lösung akzeptiert. Mit einer temperaturabhängigen Wahrscheinlichkeit werden auch schlechtere Lösungen akzeptiert. Diese Wahrscheinlichkeit nimmt zusammen mit der Temperatur in jedem Durchlauf ab.

Das Water-Level-Search-Verfahren lässt sich in zwei Phasen unterteilen. In der ersten Phase werden die Tasks nacheinander den Prozessorkernen der Rechenknoten zugewiesen und die ermittelten Endezeiten der Tasks aufgesammelt. In der zweiten Phase werden die gesammelten Endezeiten als ein Limit für den Makespan betrachtet. Gesucht wird die geringste der Endezeiten, die das Scheduling aller Tasks ermöglicht.

Die vorgestellten Verfahren können anhand ihrer Vorgehensweise in zwei Klassen eingeteilt werden. Inkrementelle suchbasierte Schedulingverfahren beginnen mit einem leeren Schedule und weisen die Tasks schrittweise zu, bis schlussendlich ein vollständiger Schedule konstruiert worden ist. Modifizierende suchbasierte Schedulingverfahren starten mit einer Anfangslösung, das heißt einem vollständigen Schedule. Dieser Schedule wird solange durch die schrittweise Neuzuweisung einzelner Tasks modifiziert, bis ein bestimmtes Abbruchkriterium erreicht ist. Zur Klasse der inkrementellen suchbasierten Schedulingverfahren gehören HP* und Water-Level-Search, während MTaSS und SAMT den modifizierenden suchbasierten Schedulingverfahren zugeordnet werden können.

Alle vier Verfahren werden in Laufzeitmessungen sowohl untereinander als auch mit existierenden List-Scheduling-Heuristiken verglichen. Dazu werden die parallelen Tasks anhand der erzeugten Schedules ausgeführt und die Gesamtlaufzeiten gemessen. Als parallele Tasks werden Anwendungen der SPLASH-3-Benchmark-Suite und eine praxisnahe Simulationsanwendung zur Bauteilbelastung verwendet. Ein wei-

teres essentielles Kriterium für Schedulingverfahren ist die Laufzeit der Verfahren selbst, die ebenfalls untersucht wird.

Die von Schedulingverfahren getroffenen Entscheidungen basieren zumeist auf Kostenmodellen. Daher wird eine Modellierung der Kosten für parallele Tasks auf heterogenen parallelen Systemen vorgestellt. Das präsentierte Kostenmodell entspricht einer parametrisierten Laufzeitformel, die das Laufzeitverhalten gewöhnlicher paralleler Anwendungen abbilden soll. Dazu setzt sich die Laufzeitformel aus einem parallelen und einem sequentiellen Anteil sowie einem logarithmischen Anteil für den Overhead der parallelen Ausführung zusammen. Das Kostenmodell wird in Bezug auf die Parameterbestimmung und die Genauigkeit evaluiert.

Gliederung der Arbeit

Die Arbeit ist wie folgt gegliedert. Kapitel 2 stellt das betrachtete Schedulingproblem für parallele Tasks vor und präsentiert Ansätze und Verfahren für dessen Lösung. Des Weiteren wird eine Klassifikation paralleler Tasks gegeben und es werden Möglichkeiten zur Konstruktion von Schedules für parallele Tasks vorgestellt.

In Kapitel 3 wird die Modellierung der Laufzeiten paralleler Tasks auf heterogenen parallelen Systemen erläutert. Es werden parallele Anwendungsprogramme vorgestellt, die in den Laufzeitmessungen dieser Arbeit als parallele Tasks betrachtet und ausgeführt werden. Anhand dieser Anwendungen wird auch das beschriebene Kostenmodell hinsichtlich der Parameterbestimmung und der Genauigkeit ausgewertet.

In Kapitel 4 wird mit HP* ein auf dem A*-Suchalgorithmus basierendes Schedulingverfahren für parallele Tasks präsentiert. Neben dem Ablauf dieses Verfahrens wird der zugrundeliegende Suchraum sowie dessen Eingrenzung beschrieben. Zudem wird gezeigt, dass HP* garantiert eine optimale Lösung findet. In Laufzeitmessungen werden die Ausführungszeiten der erzeugten Schedules sowie des Verfahrens selbst mit existierenden Schedulingverfahren verglichen.

In den Kapiteln 5 und 6 werden mit MTASS und SAMT weitere suchbasierte Schedulingverfahren vorgestellt. Während MTASS als Ansatz eine Tabu-Suche verwendet, basiert SAMT auf dem Verfahren der simulierten Abkühlung (Simulated Annealing). Für beide Verfahren wird jeweils der Aufbau und die Traversierung des Suchraums beschrieben. Die Algorithmen beider Verfahren werden erläutert und anschließend deren Komplexität analysiert. Die Leistungsfähigkeit beider Verfahren wird anhand von Laufzeitmessungen mit existierenden Schedulingverfahren verglichen.

Kapitel 7 präsentiert das suchbasierte Water-Level-Search-Verfahren, das entfernt auf der List-Scheduling-Heuristik des Water-Level-Verfahrens basiert. Beide Algorithmen werden vorgestellt und anschließend in Laufzeitmessungen mit parallelen Anwendungen untersucht und mit existierenden Schedulingverfahren verglichen.

Kapitel 8 fasst die Ergebnisse der Arbeit in einem Fazit zusammen.

2 Scheduling unabhängiger paralleler Tasks auf heterogene Plattformen

Parallele Tasks können aufgrund unterschiedlicher Eigenschaften in verschiedene Arten untergliedert werden. In diesem Kapitel wird daher zunächst eine Klassifizierung paralleler Tasks gegeben und Anwendungen paralleler Tasks vorgestellt. Anschließend folgt die Definition des in dieser Arbeit betrachteten Schedulingproblems. Des Weiteren werden verschiedene Ansätze für das Scheduling paralleler Tasks sowie populäre Schedulingverfahren beschrieben. Den Abschluss des Kapitels bilden Möglichkeiten zur Konstruktion von Schedules für parallele Tasks.

2.1 Programmiermodelle auf Basis paralleler Tasks

Bei der Implementierung komplexer paralleler Anwendungen werden häufig task- oder datenparallele Programmiermodelle eingesetzt. Anwendungen auf Basis task-paralleler Programmiermodelle lassen sich in eine Menge von Teilaufgaben (Tasks) zerlegen, die jeweils auf einem Prozessor ausgeführt werden können. Dieses Modell entspricht dem MPMD-Programmierkonzept (multiple program, multiple data), da die Tasks meist unterschiedliche Berechnungen auf verschiedenen Daten ausführen. Voneinander unabhängige Tasks können dabei parallel zueinander auf verschiedenen Prozessoren ausgeführt werden. Zwischen den Tasks können allerdings auch Daten- oder Kontrollabhängigkeiten vorliegen, die eine gewisse Reihenfolge vorgeben. Tasks mit Abhängigkeiten werden meist als Task-Graphen dargestellt, in denen die Knoten den Tasks entsprechen und die Kanten die Abhängigkeiten darstellen. In Abbildung 2.1 a) ist ein Beispiel eines solchen Task-Graphen bestehend aus neun Tasks dargestellt. Bei voneinander abhängigen Tasks kann mit der Ausführung eines Tasks erst begonnen werden, wenn alle seine Vorgänger im Task-Graphen beendet wurden.

Im datenparallelen Programmiermodell führt ein einzelnes paralleles Programm Berechnungen für unterschiedliche Teile der Daten auf mehreren Prozessoren aus. Dabei bearbeiten die verschiedenen Prozessoren unterschiedliche Daten des Programms. Der getrennte Zugriff auf die Daten kann je nach Speicherorganisation durch eine explizite Verteilung der Daten oder durch Eingrenzung der Datenbereiche erfolgen. Dieses Programmiermodell wird auch als SPMD-Modell (single program, multiple data) bezeichnet, da die Prozessoren die gleichen Operationen jeweils auf den ihnen zugeordneten Daten ausführen.

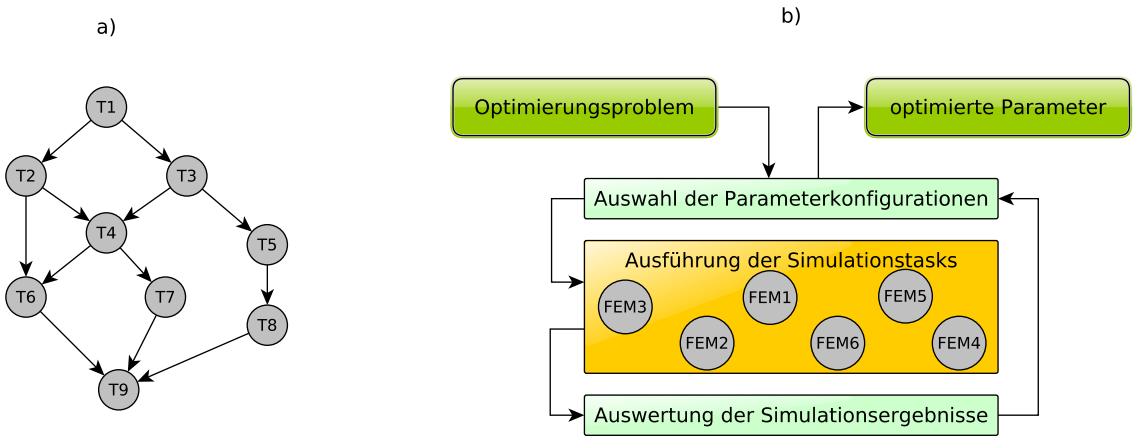


Abbildung 2.1: a) Beispiel eines Task-Graphen, dessen Kanten Kontroll- oder Datenabhängigkeiten repräsentieren können. b) Schematischer Ablauf der simulationsbasierten Optimierung kurzfaserverstärkter Kunststoffbauteile.

Anwendungen, die beide Programmiermodelle verwenden, werden auch als gemischt-parallele Anwendungen bezeichnet. Eine solche Anwendung lässt sich in eine Menge von Tasks zerlegen, die jeweils selbst von mehreren Prozessoren abgearbeitet werden können [25]. Diese Tasks werden als parallele Tasks oder auch als multiprocessor tasks, malleable tasks, moldable tasks oder rigid tasks bezeichnet. Eine Klassifikation der verschiedenen Begriffe wird in Abschnitt 2.1.2 gegeben. Parallele Tasks besitzen eine interne Datenparallelität, die mithilfe threadbasierter Programmiertechniken wie OpenMP oder Pthreads oder auf Basis nachrichtenbasierter Ansätze wie MPI realisiert sein kann. Daher können parallele Tasks sowohl für Systeme mit gemeinsamem als auch für Systeme mit verteiltem Speicher eingesetzt werden. Wie im taskparallelen Modell können auch zwischen parallelen Tasks Abhängigkeiten bestehen. In der vorliegenden Arbeit werden voneinander unabhängige parallele Tasks für Systeme mit gemeinsamem Speicher betrachtet, z.B. die Ausführung threadbasierter paralleler Anwendungen auf Multicore-Systemen.

2.1.1 Anwendungen paralleler Tasks

Durch den Einsatz neuer Fertigungsanlagen, Kommunikationssysteme und paralleler Computersysteme wird das Konzept der parallelen Tasks in vielfältigen Bereichen verwendet. Komplexe Anwendungen aus dem naturwissenschaftlichen oder technischen Bereich verwenden häufig parallele numerische Bibliotheken. Innerhalb solcher Anwendungen wird eine Vielzahl an Berechnungen ausgeführt, die so selbst wieder auf mehreren Prozessoren ausgeführt werden können. Beispiele für solche parallel ausführbare Berechnungen sind die schnelle Fourier-Transformation (FFT), die

Cholesky-Faktorisierung oder die Matrix-Multiplikation nach Strassen. Diese parallelen Berechnungen können als parallele Tasks aufgefasst werden.

Eine Anwendung paralleler Tasks, die in dieser Arbeit betrachtet wird, ist eine komplexe Simulationsanwendung [16]. Diese wurde im Rahmen des Forschungsclusters MERGE [47] entwickelt und wird zur simulationsbasierten Optimierung kurzfaserverstärkter Kunststoffbauteile eingesetzt. In Abbildung 2.1 b) ist der Ablauf der Simulationsanwendung schematisch dargestellt. Nachdem das Optimierungsproblem definiert wurde, folgt die Auswahl der Parameterkonfigurationen. Zu untersuchende Parameter sind etwa der Faseranteil oder die Einspritzposition des Faser-Kunststoff-Gemischs. Für jede Parameterkonfiguration wird anschließend eine Simulation ausgeführt. Aufgrund der hohen Anzahl an zu untersuchenden Parametern wird in jedem Durchlauf eine Vielzahl an Simulationen gestartet. Die genaue Anzahl hängt dabei von der Anzahl an Parametern sowie der Optimierungsmethode ab, liegt aber meist im zwei- bis dreistelligen Bereich. Im hier abgebildeten Beispiel handelt es sich jeweils um rechenintensive Anwendungsprogramme auf Basis der Finite-Elemente-Methode (FEM) zur Simulation der Bauteilherstellung und -belastung. Diese FEM-Anwendungen sind voneinander unabhängige parallele Programme und können daher sowohl parallel zueinander als auch jede für sich selbst parallel ausgeführt werden. Somit bilden die FEM-Anwendungen eine Menge unabhängiger paralleler Tasks, die im Folgenden als Simulationstasks bezeichnet werden. Nach der Beendigung aller Simulationstasks werden die erhaltenen Ergebnisse ausgewertet. Wenn das Optimierungskriterium erfüllt wurde, werden die optimierten Parameter zurückgeliefert. Andernfalls werden weitere Parameterkonfigurationen durch die Ausführung neuer Simulationstasks ausgewertet.

2.1.2 Klassifikation paralleler Tasks

Mit dem Begriff parallele Tasks werden im Allgemeinen Teilaufgaben (Tasks) einer Anwendung bezeichnet, die jeweils selbst von mehreren Prozessoren abgearbeitet werden können. Abhängig von der Anwendung sowie dem zugrundeliegenden Programmiermodell können parallele Tasks unterschiedliche Eigenschaften besitzen. Daraus resultieren viele spezielle Schedulingprobleme, in denen parallele Tasks unter den verschiedensten Begriffen verwendet werden. So werden parallele Tasks in der Literatur (vgl. [63, 26]) beispielsweise unter den Begriffen *multiprocessor tasks*, *malleable tasks*, *moldable tasks* und *rigid tasks* geführt. Die Begriffe gehen einher mit den verschiedenen Annahmen, die für parallele Tasks getroffen werden können, werden aber nicht immer eindeutig verwendet. Das Hauptmerkmal zur Klassifikation paralleler Tasks ist die Anzahl zu verwendender Prozessoren. So ist diese für multiprocessor tasks und rigid tasks fest vorgegeben. Bei M-tasks und moldable tasks wird die Anzahl Prozessoren vor der Ausführung vom Scheduler festgelegt und bleibt im Gegensatz zu malleable tasks während der Ausführung konstant. Eine weitere

Begriff	Eigenschaften	Auftreten
multiprocessor tasks	• fest vorgegebene Anzahl an Prozessoren • identische Prozessoren mit gemeinsamem Speicher	[5]
hierarchische multi-processor tasks (M-tasks)	• beliebige Anzahl Prozessoren möglich • M-tasks können selbst wieder aus M-tasks bestehen • Rechensysteme mit verteiltem Speicher	[24, 62, 63]
rigid tasks	• Anzahl an Prozessoren fest vorgegeben	[79, 80]
malleable tasks	• beliebige Anzahl Prozessoren möglich • Anzahl kann sich während der Laufzeit anhand eines vorher festgelegten Plans ändern	[6, 37, 48]
moldable tasks	• beliebige Anzahl Prozessoren möglich • Anzahl bleibt während der gesamten Ausführung konstant	[2, 7]

Tabelle 2.1: Arten paralleler Tasks und deren Eigenschaften sowie wissenschaftliche Arbeiten, in denen der jeweilige Begriff verwendet wird.

Unterscheidung paralleler Tasks kann anhand des zugrundeliegenden Programmiermodells getroffen werden. Beispielsweise sind multiprocessor tasks für Systeme mit gemeinsamem Speicher und M-tasks für Systeme mit verteiltem Speicher ausgelegt. Tabelle 2.1 gibt einen Überblick über die gebräuchlichsten Arten paralleler Tasks und deren Eigenschaften. Zu jedem Begriff sind zudem wissenschaftliche Arbeiten aufgelistet, in denen der jeweilige Begriff verwendet wird.

Die meisten parallelen Anwendungen entsprechen moldable tasks, denn die genaue Anzahl zu verwendender Prozessoren ist oftmals nicht vorgegeben, sondern wird abhängig von der Problemgröße oder der Verfügbarkeit gewählt. Auch die in dieser Arbeit betrachteten parallelen Tasks gehören zur Klasse der moldable tasks.

Beispiel zum Scheduling verschiedener Arten paralleler Tasks

Die Eigenschaften der verschiedenen Arten paralleler Tasks sollen an einem Beispiel verdeutlicht werden. Dazu werden die folgenden vier unabhängigen Tasks betrachtet:

- ein rigid task (blau),
- ein malleable task (grau) und
- zwei moldable tasks (grün und orange).

Diese Tasks sollen den 6 Prozessorkernen eines Multicore-Systems zugewiesen werden. Abbildung 2.2 zeigt die Illustration zwei mögliche Schedules für eine solche Zuweisung. Gezeigt ist die gebräuchliche Darstellung von Schedules als zweidimensionales Gantt-Diagramm. An der Abszisse sind dabei die Prozessorkerne des parallelen Systems und an der Ordinate ist die Zeit abgetragen. Die parallelen Tasks

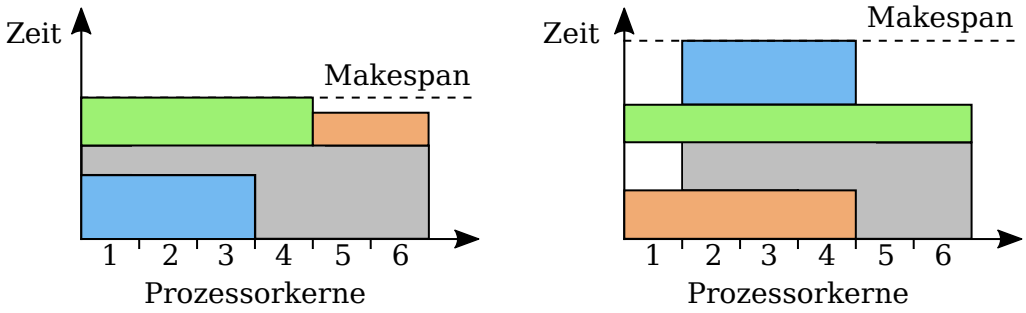


Abbildung 2.2: Beispiel für zwei mögliche Schedules, in denen jeweils verschiedene Arten von unabhängigen parallelen Tasks auf 6 Prozessorkerne verteilt sind: ein rigid task (blau), ein malleable task (grau) und zwei moldable tasks (grün und orange).

sind als einander nicht überlappende Rechtecke dargestellt, die vertikal durch ihre Start- und Endezeit begrenzt sind. Die horizontale Größe und Position ergibt sich aus der Menge verwendeter Prozessorkerne.

Die vier parallelen Tasks können anhand ihrer Eigenschaften unterschiedlich den Prozessorkernen zugewiesen werden und aufgrund ihrer Unabhängigkeit auch in der Reihenfolge variieren. Der rigid task (blau) wird auf einer festen Anzahl Prozessorkerne ausgeführt, die im gezeigten Beispiel 3 beträgt. Dieser Task kann also immer nur einer Menge aus genau 3 Kernen zugewiesen werden. Im linken Schedule ist das die Menge der Prozessorkerne $\{1, 2, 3\}$ und im rechten Schedule die Menge $\{2, 3, 4\}$. Ein malleable task (grau) kann die von ihm verwendete Menge an Prozessorkernen während seiner Ausführung verändern. So beginnt der malleable task im linken Schedule auf den Kernen $\{4, 5, 6\}$ und nutzt nach der Beendigung des rigid tasks zusätzlich noch dessen Kerne. Im rechten Schedule wird der malleable task auf den Kernen $\{5, 6\}$ gestartet und seine Ausführung direkt nach dem Ende des orangen Tasks um die Kerne $\{2, 3, 4\}$ erweitert. An dieser Stelle wird deutlich, dass Lücken in Schedules entstehen können, die aber nach Möglichkeit vermieden werden sollten. Die beiden moldable tasks (grün und orange) können während des Schedulings einer beliebigen, aber festen Menge an Prozessorkernen zugewiesen werden, die während der Ausführung konstant bleibt. So verwenden die beiden Tasks im linken Schedule die Kerne $\{1, 2, 3, 4\}$ und $\{5, 6\}$ und im rechten Schedule die Kerne $\{1, 2, 3, 4, 5, 6\}$ und $\{1, 2, 3, 4\}$. Abhängig von der Programmstruktur eines parallelen Tasks beeinflusst die verwendete Menge an Prozessorkernen dessen Laufzeit.

2.2 Definition des Schedulingproblems für parallele Tasks

In Abhängigkeit von der Art der Tasks, der Eigenschaften der Rechenressourcen, der einzuhaltenden Regeln und des Optimalitätskriteriums ergeben sich verschiedene Schedulingprobleme. Im Allgemeinen wird zwischen *statischem* und *dynamischen* Scheduling sowie zwischen *online* und *offline* Scheduling unterschieden. Bei statischem Scheduling stehen die Menge der Tasks und die Rechenressourcen bereits vor der Laufzeit fest und bleiben währenddessen unverändert. Im Gegensatz dazu können sich beim dynamischen Scheduling die Anforderungen aufgrund der jeweils aktuellen Bedarfssituation während des Schedulings ändern. So können beispielsweise im Verlauf des Schedulings weitere Tasks hinzukommen, die das Unterbrechen bereits in der Ausführung befindlicher Tasks erfordern. Beim offline Scheduling wird die komplette Ablaufplanung vor der Ausführung durchgeführt. Dazu müssen bereits im Vorhinein alle Informationen vorhanden sein, die für das Scheduling benötigt werden, wie etwa die Tasks und deren Laufzeiten sowie die Anzahl der Rechenressourcen und deren Eigenschaften. Ein online Scheduler trifft hingegen alle Entscheidungen zur Laufzeit und auf Grundlage der aktuellen Situation. Beispiele für die Anwendung von online Schedulern finden sich in [34] und [78]. Sowohl beim statischen als auch beim dynamischen Scheduling kann die Ausführung der Tasks unterbrechbar (preemptive) oder nicht-unterbrechbar (non-preemptive) sein.

Eine weitere Unterscheidung ergibt sich aus der Struktur der Tasks. So können die Tasks *Abhängigkeiten* zueinander haben oder komplett *unabhängig* voneinander sein. Solche Abhängigkeiten können sowohl Daten- als auch Kontrollabhängigkeiten sein. Daher muss beim Scheduling voneinander abhängiger Tasks die durch die Abhängigkeiten gegebene Reihenfolge der Tasks eingehalten und die Zeit für einen möglichen Datenaustausch berücksichtigt werden. Im Gegensatz dazu können unabhängige Tasks in einer beliebigen, durch das Scheduling bestimmten Reihenfolge und unter Umständen auch parallel zueinander ausgeführt werden. Somit ergeben sich für das Scheduling unabhängiger Tasks mehr Möglichkeiten als für Tasks mit Abhängigkeiten, was zu einer deutlich höheren Komplexität führt.

Diese Arbeit beschäftigt sich mit dem statischen Offline-Scheduling unabhängiger, paralleler (moldable) Tasks, deren Ausführung nicht unterbrochen werden kann. Das betrachtete Schedulingproblem besteht aus n_T unabhängigen parallelen Tasks, die einer heterogenen Plattform aus n_N Rechenknoten zugewiesen werden sollen. Im Folgenden werden die konkreten Eigenschaften der parallelen Tasks und der heterogenen Plattform sowie das Ziel des Schedulings definiert. Tabelle 2.2 listet die verwendete Notation zur Beschreibung des Schedulingproblems auf.

Parallele Tasks. Betrachtet werden n_T unabhängige parallele Tasks $T_1, \ldots, T_{n_T}$, die die Gesamtheit einer komplexen Anwendung repräsentieren. Die konkrete Programmstruktur der Tasks selbst ist unbekannt. Jeder Task kann auf einer

Notation	Bedeutung
n_T	Anzahl paralleler Tasks
$T_1, \ldots T_{n_T}$	Menge unabhängiger paralleler Tasks
n_N	Anzahl der Rechenknoten der heterogenen Plattform
$N_1, \ldots N_{n_N}$	Menge der Rechenknoten der Zielplattform
p_j	Anzahl der Prozessorkerne des Rechenknotens N_j
K	Gesamtanzahl Prozessorkerne der heterogenen Plattform
F	Gesamtleistungsfaktor der heterogenen Plattform
f_j	Leistungsfaktor des Rechenknotens N_j
N_r	Referenzrechenknoten
$t_{i,j}(p)$	Parallele Laufzeit des Tasks T_i auf p Kernen des Rechenknotens N_j
s_i	Startzeit des Tasks T_i
e_i	Endezeit des Tasks T_i
$M(S)$	Makespan von Schedule S

Tabelle 2.2: Verwendete Notation zur Beschreibung des Schedulingproblems

beliebigen Anzahl an Prozessorkernen eines einzelnen Rechenknotens ausgeführt werden. Diese Anzahl wird vor der Ausführung festgelegt und bleibt bis zur Beendigung des Tasks unverändert. Die Unterbrechung und spätere Fortsetzung eines Tasks ist nicht möglich (non-preemptive execution). Jeder Task $T_i, i \in 1, \ldots, n_T$ besitzt Berechnungskosten $t_{i,j}(p)$. Diese repräsentieren die parallele Ausführungszeit von T_i auf p Prozessorkernen des Rechenknotens $N_j, j \in 1, \ldots, n_N$. Die Berechnungskosten können entweder durch Messungen bestimmt oder mithilfe eines Kostenmodells berechnet werden.

Heterogene Plattform. Die Zielplattform stellt ein heterogenes paralleles System dar, das aus den n_N Rechenknoten $N_1, \ldots, N_{n_N}$ besteht. Als Rechenknoten werden Multicore-Systeme mit unterschiedlichen Hardwareeigenschaften betrachtet, was zur Heterogenität der gesamten Plattform führt. Präziser ausgedrückt, verfügen die Systeme über verschiedene Prozessoren, die sich in der Anzahl von Prozessorkernen und der Prozessorleistung unterscheiden können. So besitzt jeder Rechenknoten $N_j, j \in 1, \ldots, n_N$ eine beliebige, aber feste Anzahl p_j Prozessorkerne sowie einen Leistungsfaktor f_j, der die Rechengeschwindigkeit des Knotens im Verhältnis zu einem Referenzrechenknoten angibt. Als Referenzrechenknoten wird ein beliebiger Rechenknoten N_r des heterogenen Systems festgelegt. Die Gesamtanzahl K im heterogenen parallelen System vorhandener Prozessorkerne kann somit wie folgt beschrieben werden:

$$K = \sum_{j=1}^{n_N} p_j \tag{2.1}$$

Der Gesamtleistungsfaktor F des heterogenen Systems entspricht dem Produkt aus der Anzahl Prozessorkerne p_j und dem Leistungsfaktor f_j aufsummiert über alle Rechenknoten N_j und wird wie folgt ermittelt:

$$F = \sum_{j=1}^{n_N} p_j \cdot f_j \tag{2.2}$$

Schedule. Ein Schedule beschreibt die zeitliche und räumliche Zuweisung der parallelen Tasks zu einer Menge von Prozessorkernen der Rechenknoten. Somit stellt ein Schedule eine Lösung des Schedulingproblems dar und regelt die Ausführung der taskbasierten Anwendung. Konkret ordnet ein Schedule S jedem parallelen Task T_i eine Menge von Prozessorkernen p eines Rechenknotens N_j zu und legt die Startzeit s_i und die Endezeit e_i fest. Die Endezeit e_i ergibt sich dabei aus der Summe der Startzeit s_i und der parallelen Laufzeit $t_{i,j}(p)$, das heißt, es gilt:

$$e_i = s_i + t_{i,j}(p) \tag{2.3}$$

Einem Task können somit bis zu p_j Kerne zugewiesen werden, wobei jeder Prozessorkern zu einem Zeitpunkt höchstens einen Task ausführen kann. Deshalb entspricht die frühest mögliche Startzeit eines Tasks der größten Endezeit aller Tasks, die bereits den ermittelten Kernen zugewiesen sind. Eine weitere Konsequenz ist, dass abhängig von Anzahl zugewiesener Kerne auf jedem Rechenknoten bis zu p_j Tasks gleichzeitig ausgeführt werden können. Der *Makespan* $M(S)$ eines Schedules S ist definiert als die Zeit zwischen der Startzeit des ersten Tasks und der Endezeit des letzten Tasks. Es wird angenommen, dass der erste Task zum Zeitpunkt 0 beginnt und der Makespan somit folgendermaßen berechnet werden kann:

$$M(S) = \max_{i=1,\ldots,n_T} e_i \tag{2.4}$$

Das Optimierungsziel des betrachteten Schedulingproblems ist es für eine gegebene Menge paralleler Tasks und eine Zielplattform einen Schedule mit minimalem Makespan zu bestimmen.

2.3 Ansätze für das Scheduling paralleler Tasks

Parallele Tasks können auf einer beliebigen Anzahl Prozessoren ausgeführt werden, wobei mehr Prozessoren üblicherweise zu kürzeren Tasklaufzeiten führen. Andererseits können voneinander unabhängige Tasks parallel zueinander ausgeführt werden. Neben dem Bestimmen einer Ausführungsreihenfolge der parallelen Tasks und der Zuweisung zu den Rechenknoten muss für jeden Task zusätzlich noch die Anzahl zu verwendender Prozessoren bestimmt werden. Eine Aufgabe des Schedulings ist

es demnach, für jeden Task die Menge an Prozessoren zu bestimmen, so dass ein festgelegtes Optimalitätskriterium erreicht wird, z. B. die Minimierung der gesamten Ausführungszeit. Entscheidend für die Qualität der Lösung ist es, einen guten Kompromiss zwischen einer task- und einer datenparallelen Ausführung zu finden. Das führt zu der Frage, ob mehr Tasks auf jeweils weniger Prozessoren oder weniger Tasks auf jeweils mehr Prozessoren ausgeführt werden sollen.

In den letzten Jahrzehnten wurden verschiedenste Schedulingprobleme für parallele Tasks untersucht. Zur deren Lösung wurden diverse Algorithmen vorgestellt und verglichen. Jedoch ist Task-Scheduling bis auf einige wenige Spezialfälle ein NP-schweres Problem. In [22] wurde gezeigt, dass das Scheduling von rigid tasks NP-schwer ist. Daraus folgt, dass auch das in dieser Arbeit betrachteten Schedulingproblem NP-schwer ist, da es sich beim Scheduling von rigid tasks lediglich um einen Spezialfall dieses Problems handelt. Zusätzlich zur Reihenfolge der Tasks und deren Zuordnung zu den Prozessorkernen muss im Gegensatz zu den rigid tasks noch die Anzahl der Prozessorkerne bestimmt werden.

Da das Finden einer optimalen Lösung in polynomieller Zeit nicht garantiert werden kann, beschränken sich die meisten Schedulingverfahren auf Approximationsalgorithmen und Heuristiken. Diese Ansätze wägen für gewöhnlich zwischen der Komplexität des Schedulingalgorithmus und der Qualität der Lösung ab. Die beiden am häufigsten eingesetzten Klassen heuristischer Schedulingalgorithmen sind List-Scheduling-Verfahren und suchbasierte Schedulingverfahren. In [10] und [38] werden jeweils Algorithmen beider Klassen miteinander verglichen. Alle darin vorgestellten Algorithmen betrachten allerdings *sequentielle Tasks*, die jeweils nur auf genau einem Prozessor oder Prozessorkern ausgeführt werden können. Daher lassen sich diese Verfahren nicht auf das komplexere Schedulingproblem für parallele Tasks anwenden. Im Folgenden werden beide Herangehensweisen und deren wichtigste Vertreter beschrieben.

2.3.1 List-Scheduling-Verfahren

Ein weitverbreiteter Ansatz für das Task-Scheduling sind List-Scheduling-Verfahren. Diese sortieren die Tasks zunächst nach einem bestimmten Kriterium, z. B. nach ihrer Laufzeit. Anschließend werden die Tasks in dieser Reihenfolge den Prozessoren zugewiesen. Auch bei der Zuweisung können sich die Verfahren unterscheiden, so etwa in der Auswahl der Prozessoren. Im Schedulingverfahren Heterogeneous Earliest Finish Time (HEFT)[76] wird beispielsweise jeder sequentielle Task dem Prozessor zugewiesen, der die früheste Endezeit des Tasks zur Folge hat. Beim Scheduling paralleler Tasks muss vor der Zuweisung der Tasks allerdings noch die zu verwendende Anzahl an Prozessoren bestimmt werden. Entsprechende Verfahren führen dazu vor der Zuweisungsphase eine Allokationsphase aus, in der die Prozessoranzahl für jeden Task ermittelt wird. Die konkrete Zuweisung zu einer Menge von Prozessoren

erfolgt nachdem deren Anzahl für jeden Task feststeht. Als Vertreter dieser Kategorie sind beispielsweise Critical Path and Area-based Scheduling (CPA)[29], Critical Path Reduction (CPR)[59] und Two Step Allocation and Scheduling (TSAS)[60] zu nennen. Diese Algorithmen sind allerdings nur für homogene parallele Systeme und somit nicht für das in dieser Arbeit betrachtete Schedulingproblem geeignet.

Verfahren für das Scheduling paralleler Tasks mit Abhängigkeiten auf heterogene parallele Systeme sind zum Beispiel Heterogeneous Critical Path and Allocation (HCPA)[50] und Δ-Critical Task Set (Δ-CTS)[74], die im Folgenden näher vorgestellt werden. Beide Algorithmen verwenden für ihre Berechnungen den sogenannten bottom-level eines Tasks. Als *bottom-level* wird die Länge des längsten Pfades vom jeweiligen Task zum Endknoten im Task-Graphen bezeichnet. Die Länge eines Pfads ergibt sich aus der Summe der Berechnungs- und Kommunikationszeiten aller Tasks auf dem Pfad. Als kritischer Pfad wird der längste Pfad im Task-Graphen bezeichnet. Angewendet auf unabhängige Tasks entspricht das bottom-level eines Tasks exakt dessen sequentieller Laufzeit.

Heterogeneous Critical Path and Allocation (HCPA)

Wie beschrieben ist das List-Scheduling-Verfahren CPA für homogene parallele Systeme konzipiert. Bei HCPA handelt es sich nicht um ein komplett neues Verfahren, sondern um eine Modifikation von CPA für heterogene parallele Systeme. Diese Modifikation beruht auf dem Konzept eines „virtuellen homogenen Referenzclusters“, der die gleiche Rechenleistung wie der heterogene Cluster besitzt. Die Prozessoren des Referenzclusters haben die gleiche Leistung wie der schwächste Prozessor des realen Systems. Die Prozessoranzahl des Referenzclusters ergibt sich aus der Anzahl realer Prozessoren jeweils multipliziert mit dem Verhältnis der Leistung des Prozessors zur Leistung des Referenzprozessors. Daraus folgt, dass der Referenzcluster zwar leistungsschwächere, aber dafür mehr Prozessoren als der reale Cluster hat, um auf die selbe Gesamtleistung zu kommen. Das Scheduling von HCPA erfolgt ebenfalls in einer Allokationsphase gefolgt von einer Zuweisungsphase.

In der Allokationsphase wird die Anzahl zu verwendender Prozessoren auf dem Referenzcluster zunächst für jeden Task mit 1 initialisiert. In jeder Iteration dieser Phase wird die Prozessoranzahl eines Tasks auf dem kritischen Pfad um 1 erhöht. Die Iteration stoppt sobald das größte bottom-level aller Tasks kleiner oder gleich der mittleren Prozessorzeit aller Tasks ist oder kein Prozessor mehr hinzugefügt werden kann. Zur Überprüfung der letztgenannten Bedingung wird die für den Referenzcluster ermittelte Prozessoranzahl wieder für das reale System umgerechnet. Diese umgerechnete Anzahl darf die zur Verfügung stehenden Prozessoren des realen Systems nicht überschreiten.

In der Zuweisungsphase wird jeder Task den Prozessoren zugewiesen, auf denen die früheste Endezeit erreicht wird. Dazu wird die für den Referenzcluster ermittelte Prozessoranzahl in die entsprechende Anzahl für jeden Rechenknoten des realen Sys-

tems umgerechnet. Somit kann eine Zuweisung zu mehreren Rechenknoten möglich sein, von denen die mit der frühesten Endezeit inklusive eventueller Datenumverteilungskosten gewählt wird.

Δ-Critical Task Set (Δ-CTS)

Ein Nachteil des strikten Vorgehens nach dem List-Scheduling-Prinzip ist, dass die Zuweisung eines Tasks unter Umständen negativen Einfluss auf alle folgenden Tasks hat. Diesen Nachteil versucht das Schedulingverfahren Δ-CTS abzuschwächen, indem es einen modifizierten List-Scheduling-Ansatz verwendet. Anstatt einzelner Tasks werden Gruppen von Tasks mit ähnlicher Priorität gebildet, die anhand des bottom-level ermittelt wird. Für jeden Task einer solchen Gruppe wird die Anzahl zu verwendender Prozessoren bestimmt, auf denen die früheste Endezeit erreicht wird. In die Ermittlung der Prozessoranzahl fließen sowohl die Gruppengröße als auch die Leistung der Prozessoren ein. Anschließend werden diese Tasks den Prozessoren zugewiesen, auf denen die früheste Endezeit erreicht wurde. Danach wird jeweils mit der nächsten Gruppe fortgefahren bis alle Tasks zugewiesen wurden.

2.3.2 Suchbasierte Schedulingalgorithmen

Task-Scheduling kann auch als kombinatorisches Optimierungsproblem formuliert werden. In diesem soll eine Menge von Tasks einer Menge von Prozessoren unter Einhaltung bestimmter Regeln so zugewiesen werden, dass ein festgelegtes Optimalitätskriterium erreicht wird. Daher können auch klassische Suchverfahren zur Lösung von Schedulingproblemen eingesetzt werden. Im Gegensatz zu List-Scheduling-Verfahren oder anderen Heuristiken benötigen Suchverfahren jedoch mehr Rechenaufwand, führen aber häufig zu besseren Ergebnissen. In den letzten Jahrzehnten war der Einsatz von Suchverfahren aufgrund der hohen Rechenlast für den praktischen Einsatz als Schedulingverfahren eher ungeeignet. Heutzutage ermöglichen moderne leistungsstarke Computer und parallele Systeme dagegen eine Anwendung solcher Verfahren in angemessener Zeit.

Lokale und globale Suchverfahren

Eine Suche nach dem globalen Optimum eines NP-schweren Schedulingproblems benötigt viel Zeit, da unter Umständen der komplette Suchraum durchsucht werden muss. Die A*-Suche [35] ist ein informiertes Suchverfahren, das unter gewissen Voraussetzungen das Finden eines globalen Optimums garantieren kann. Hauptsächlich wird die A*-Suche zum Auffinden eines kürzesten Pfades zwischen zwei Knoten in einem gewichteten Graphen eingesetzt. Eine weitere Anwendung ist beispielsweise das Parsing mit probabilistischen kontextfreien Grammatiken in der Computerlinguistik [43]. In [14] wurde gezeigt, dass die A*-Suche eine optimale Lösung findet,

wenn die verwendete Kostenfunktion bestimmte Eigenschaften erfüllt. Schedulingalgorithmen auf Basis der A*-Suche werden in [45] und [72] vorgestellt. Die beiden Arbeiten enthalten auch Vorschläge zur Reduzierung des Suchraums, wurden aber für sequentielle Tasks mit Abhängigkeiten entworfen und sind daher nicht für das Scheduling paralleler Tasks geeignet.

Lokale Suchverfahren nutzen die Informationen der Umgebung einer Lösung, um eine bessere Lösung zu finden. Daher konvergieren diese Verfahren deutlich schneller, haben aber den Nachteil in lokalen Optima festzustecken. Ein Beispiel dafür ist der Bergsteigeralgorithmus (engl. hill climbing), bei dem ausgehend von einer zufälligen Lösung in jedem Schritt eine bessere Lösung in der unmittelbaren Umgebung gesucht wird. Wenn eine bessere Lösung gefunden wird, beginnt die Suche von dieser Lösung erneut, andernfalls stoppt der Algorithmus.

Metaheuristiken

Eine Möglichkeit lokalen Optima zu entkommen, ist die Anwendung sogenannter Metaheuristiken. Diese akzeptieren dazu unter gewissen Voraussetzungen auch schlechtere Lösungen, garantieren jedoch nicht das Finden einer globalen optimalen Lösung. Der Begriff Metaheuristik wurde erstmals von Glover [30] vorgestellt und seitdem sind zahlreiche Metaheuristiken auf unterschiedliche Optimierungsprobleme angewendet worden. Eine Metaheuristik definiert keine problemspezifische Heuristik, sondern beschreibt eine abstrakte Folge von Schritten zur Anwendung auf beliebige Problemstellungen. Zur Lösung eines konkreten Problems müssen die jeweiligen Schritte problemspezifisch angepasst werden.

Auch zur Lösung verschiedener Schedulingprobleme wurden Algorithmen vorgestellt, die auf Metaheuristiken basieren. Viele dieser Algorithmen sind dabei von Vorgängen aus der Natur inspiriert. So existieren unter anderem Genetische Algorithmen, die mithilfe von Selektions-, Rekombinations- und Mutationsoperatoren evolutionäre Prozesse zur Optimierung von Lösungen nutzen. Andere Algorithmen modellieren das Verhalten von Tierschwärmen, z. B. von Ameisen (engl. Ant Colony Optimization) bzw. von Vogel- oder Fischschwärmen (engl. particle swarm optimization). Tabelle 2.3 listet die populärsten Metaheuristiken für das Task-Scheduling auf und verweist auf entsprechende Veröffentlichungen. Mit Ausnahme von [9] wurden die genannten Verfahren für sequentielle Tasks entwickelt und sind somit nicht für das Scheduling paralleler Tasks geeignet. Nach aktuellem Stand scheint kein metaheuristisches Verfahren für das Scheduling paralleler (moldable) Tasks zu existieren, da in [9] für jeden Task die Anzahl zu verwendender Prozessoren fest vorgegeben ist.

Metaheuristik	Literatur-verweis	Betrachtetes Schedulingproblem
Genetische Algorithmen	[1, 33, 71]	sequentielle Tasks mit Abhängigikeiten, heterogener Cluster
	[53]	sequentielle Tasks mit Abhängigikeiten, homogener Cluster
Ant Colony Optimization	[77]	sequentielle Tasks mit Abhängigikeiten, heterogener Cluster
	[8]	sequentielle Tasks mit Abhängigikeiten, homogener Cluster
Particle Swarm Optimization	[67, 68]	sequentielle Tasks mit Abhängigikeiten, heterogener Cluster
Tabu-Suche	[81]	sequentielle Tasks mit Abhängigikeiten, heterogener Cluster
	[9]	unabhängige rigid tasks, homogenes paralleles System
Simulated Annealing	[4, 57]	sequentielle Tasks mit Abhängigikeiten, heterogener Cluster

Tabelle 2.3: Populäre Metaheuristiken zur Lösung diverser Schedulingprobleme und Verweise auf entsprechende wissenschaftliche Arbeiten.

2.4 Konstruktion von Schedules für parallele Tasks

Die Lösung des Schedulingproblems für parallele Tasks erfordert die Konstruktion mindestens eines Schedules. Die Konstruktion von Schedules kann im Wesentlichen in zwei Ansätze unterteilt werden. Dies ist zum einen der inkrementelle Aufbau eines Schedules, der zumeist von List-Scheduling-Verfahren, aber auch von einigen suchbasierten Schedulingverfahren eingesetzt wird. Zum anderen kann ein existierender Schedule durch Transformationsschritte in einen anderen Schedule überführt werden. Nachfolgend werden diese beiden Ansätze im Zusammenhang mit relevanten Begriffen erläutert.

2.4.1 Inkrementeller Aufbau eines Schedules

Eine Möglichkeit zur Konstruktion eines Schedules ist die schrittweise Zuweisung der Tasks zu den Prozessorkernen der Rechenknoten. Häufig werden die Tasks vor der Zuweisung sortiert, zum Beispiel anhand ihrer Laufzeit oder bestimmter Prioritäten. Die Zuweisung erfolgt dann zumeist im Hinblick auf das Optimierungsziel des jeweiligen Schedulingproblems, zum Beispiel das Erreichen der frühesten Endezeit. Die Zwischenschritte, die bei einem solchen inkrementellen Aufbau eines Schedules auftreten können lassen sich in die folgenden drei Kategorien unterteilen:

Leerer Schedule. Ein Schedule, in dem noch keine Zuweisungen definiert sind, wird als leer bezeichnet.

Partieller Schedule. In einem partiellen Schedule wurde nur eine Teilmenge aller Tasks den Prozessorkernen zugewiesen.

Vollständiger Schedule. Ein Schedule ist vollständig, wenn darin alle Tasks einer Menge von Prozessorkernen zugewiesen wurden. Ein vollständiger Schedule stellt eine Lösung des Schedulingproblems dar.

Der inkrementelle Aufbau eines Schedules beginnt mit einem leeren Schedule. Die schrittweise Zuweisung von Tasks zu Prozessorkernen führt jeweils zu partiellen Schedules und schlussendlich zu einem vollständigen Schedule. Bei der Zuweisung eines Tasks entspricht die frühest mögliche Startzeit des Tasks der größten Endezeit aller Tasks, die bereits den entsprechenden Prozessorkernen zugewiesen sind.

Der inkrementelle Aufbau eines Schedules führt zur Aufteilung der Taskmenge in die disjunkten Teilmengen der bereits zugewiesenen Tasks und der noch nicht zugewiesenen (verbleibenden) Tasks. Zur Auswertung der verbleibenden Tasks werden folgende Begriffe eingeführt:

Definition 2.1 (Verbleibende Rechenzeit)**.** Für die Zuweisung eines parallelen Tasks werden p Prozessorkerne eines Rechenknotens N_j für die Dauer der parallelen Laufzeit $t_{i,j}(p)$ benötigt. Die tatsächlich verbleibende Rechenzeit ergibt sich als Produkt aus paralleler Laufzeit, Anzahl Prozessorkerne und dem Leistungsfaktor f_j aufsummiert über alle verbleibenden Tasks. Da die konkrete Zuweisung der verbleibenden Tasks noch unbekannt ist, wird eine sequentielle Ausführung auf einem Kern des Referenzrechenknotens angenommen. Somit sind die Werte für p und f_j gleich 1 und die verbleibende Rechenzeit R_n eines partiellen Schedules S_n wird daher als die Summe der sequentiellen Laufzeiten der verbleibenden Tasks auf dem Referenzrechenknoten N_r bezeichnet. Sei U_n die Menge der verbleibenden Tasks eines partiellen Schedules S_n, dann berechnet sich R_n wie folgt:

$$R_n = \sum_{T_i \in U_n} t_{i,r}(1). \tag{2.5}$$

Definition 2.2 (Verfügbare Prozessorzeit)**.** Für einen Schedule S_n bezeichnet die verfügbare Prozessorzeit $P(S_n)$ die maximale Rechenzeit, die zusätzlich ausgeführt werden kann ohne den Makespan $M(S_n)$ zu erhöhen. Sei $C_{j,k}$ die Menge der Tasks, die im Schedule S_n dem Kern k des Rechenknotens N_j zugewiesen wurden, dann lässt sich die verfügbare Prozessorzeit $P(S_n)$ wie folgt berechnen:

$$P(S_n) = \sum_{j=1}^{n_N} \sum_{k=1}^{p_j} (M(S_n) - \max_{T_i \in C_{j,k}} e_i). \tag{2.6}$$

2.4.2 Modifikation eines bestehenden Schedules

Ein intuitiver Ansatz zur Lösungsfindung, auf dem die meisten Suchverfahren basieren, ist das schrittweise Verändern einer Anfangslösung. In Bezug auf das Scheduling paralleler Tasks entspricht eine Anfangslösung einem Schedule, der beispielsweise durch ein Schedulingverfahren oder durch die zufällige Zuweisung der Tasks zu den Rechenressourcen generiert werden kann. An einem solchen Schedule werden dann schrittweise kleine Änderungen vorgenommen, bis ein Abbruchkriterium erreicht wird.

Die kleinstmögliche Änderung beim Scheduling paralleler Tasks ist die Neuzuweisung eines einzelnen Tasks. Eine Neuzuweisung bezeichnet dabei das Entfernen eines Tasks aus dem Schedule und die anschließende Zuweisung an eine Menge von Prozessorkernen eines Rechenknotens. Alle Tasks, deren Startzeit von von diesem Task abhing, können nach dem Entfernen zur nächstmöglichen früheren Startzeit beginnen. Die anschließende Zuweisung des entfernten Tasks zu den Prozessorkernen erfolgt zur frühestmöglichen Startzeit. Mithilfe dieser Vorgehensweise kann ein Schedule in einen anderen Schedule transformiert werden. Ein so erzeugter Schedule wird im Folgenden als Nachbar des Ausgangsschedules bezeichnet. Da eine Neuzuweisung jedes Tasks auf jede Kombination der Prozessorkerne eines jeden Rechenknotens möglich ist, kann ein Schedule große Anzahl an Nachbarn besitzen, die im Folgenden als Nachbarschaft bezeichnet wird. Die Gesamtheit aller Nachbarn bildet den Lösungsraum des Schedulingproblems.

Definition 2.3 (Nachbar eines Schedules)**.** Ein Schedule S' wird als *Nachbar* eines Schedules S im Lösungsraum bezeichnet, wenn er sich in genau einer Zuweisung eines Tasks unterscheidet. Um ausgehend von einem Schedule S einen benachbarten Schedule S' zu erhalten, muss demnach nur ein Task aus dem Schedule S entfernt werden und anschließend einer anderen Menge von Prozessorkernen zugewiesen werden. Die Prozessorkerne, denen der Task zugewiesen wird, können demselben Rechenknoten angehören oder sich auf einem anderen Rechenknoten befinden.

Definition 2.4 (Nachbarschaft eines Schedules)**.** Als Nachbarschaft eines Schedules wird die Menge aller Nachbarn dieses Schedules bezeichnet. Die Nachbarschaft zu einem Schedule kann erzeugt werden, in dem jeder Nachbar generiert wird. Dazu wird jeder Task separat jeder Kombination der Prozessorkerne jedes Rechenknotens zugewiesen. Jede neue Zuweisung entspricht einem Nachbarn im Lösungsraum.

Beispiel für die Generierung der Nachbarschaft eines Schedules

Die Abbildung 2.3 zeigt einen Teil des Lösungsraums für das Scheduling zweier Tasks (gelb und grau) auf einen Rechenknoten mit 2 Prozessorkernen. Die Nachbarschaftsbeziehungen zwischen einzelnen Schedules sind durch Pfeile markiert. Jeder

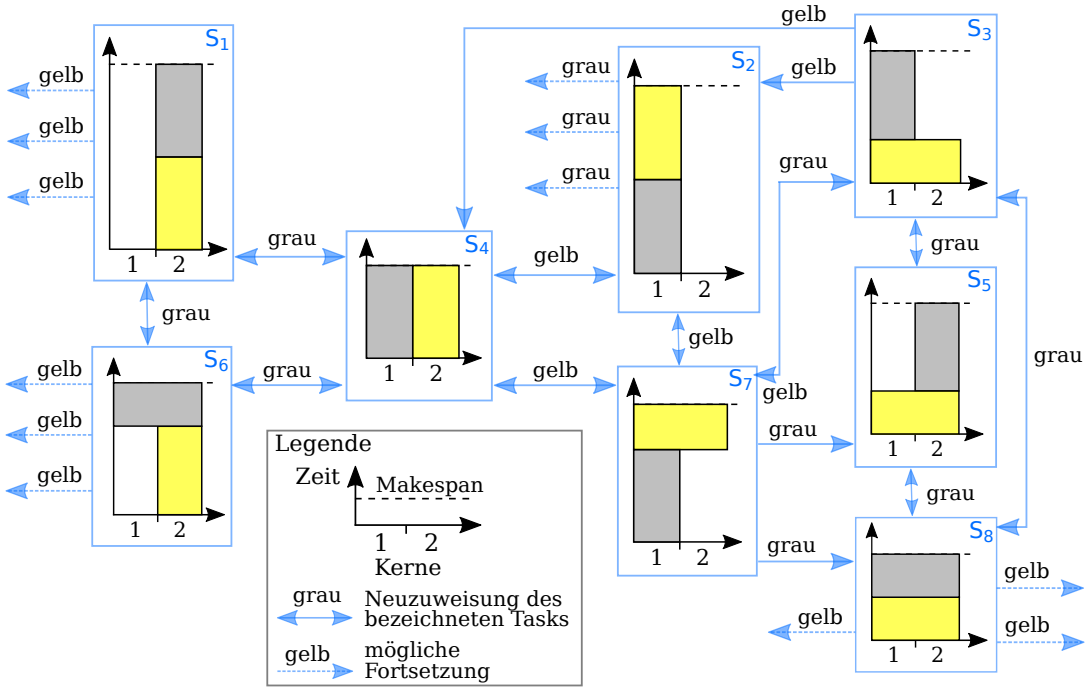


Abbildung 2.3: Teil des Lösungsraums für das Scheduling zweier unabhängiger paralleler Tasks (gelb und grau) auf einen Rechenknoten mit 2 Prozessorkernen.

Pfeil kennzeichnet die Transformation eines Schedules in einen benachbarten Schedule durch Neuzuweisung des angegebenen Tasks. Ausgehend von Schedule S_4 kann eine Neuzuweisung des gelben Tasks zu den Schedules S_2 und S_7 führen. Dazu wird der gelbe Task von Kern 2 entfernt und entweder Kern 1 (Schedule S_2) oder parallel den Kernen 1 und 2 (Schedule S_7) zugewiesen. Vom Schedule S_7 führt eine Neuzuweisung des gelben Tasks wiederum zu den Nachbarn S_2 (Zuweisung zu Kern 1) und S_4 (Zuweisung zu Kern 2). Eine Neuzuweisung des grauen Tasks in S_7 kann in den Schedules S_3, S_5 und S_8 resultieren. Dazu wird der graue Task aus S_7 entfernt, wodurch der gelbe Task nun früher starten kann. Anschließend wird der graue Task allen Teilmengen der Prozessorkerne zugewiesen und kann somit erst beginnen, wenn der gelbe Task beendet wurde. Daraus ergeben sich die Nachbarn S_3 (Zuweisung zu Kern 1), S_5 (Zuweisung zu Kern 2) und S_8 (parallele Zuweisung zu den Kernen 1 und 2).

3 Kostenmodellierung paralleler Tasks auf heterogenen Systemen

In nahezu allen statischen Task-Scheduling-Verfahren basieren die Entscheidungen auf den Laufzeiten der Tasks. Allerdings stehen die exakten Tasklaufzeiten in praktischen Einsatzgebieten zumeist nicht zur Verfügung. Eine Messung der Laufzeiten sämtlicher Tasks wäre zu zeitaufwendig im Vergleich zur Reduzierung der Gesamtausführungszeit durch ein Schedulingverfahren. Daher werden in diesen Fällen meist modellierte Tasklaufzeiten verwendet. Eine Herausforderung ist jedoch das Finden einer geeigneten Modellierung für die Laufzeit paralleler Tasks. Eine solche Modellierung kann beispielsweise durch Auswertung der Programmcodes der Tasks erfolgen. Derartige Ansätze für parallele Programme werden unter anderem in den Modellen PRAM [27], BSP [73] und LogP [13] genutzt. Eine weitere Möglichkeit für parallele Tasks, die auch im Folgenden angewendet wird, ist die Verwendung einer parametrisierte Laufzeitformel, deren Parameter mithilfe einiger weniger Messungen geschätzt werden können.

3.1 Modellierung der Tasklaufzeiten

Für die Lösung des in Abschnitt 2.2 beschriebenen Schedulingproblems, wird die Laufzeit $t_{i,j}(p)$ des Tasks T_i in Abhängigkeit von der Anzahl der verwendeten Prozessorkerne p des Rechenknotens N_j benötigt. Da die Programmstruktur der betrachteten parallelen Tasks unbekannt ist, kann diese nicht zur Laufzeitmodellierung genutzt werden. Aus diesem Grund wird in dieser Arbeit eine allgemeine parametrisierte Laufzeitformel verwendet.

In [20] wird ein Modell betrachtet, in dem parallele Programme aus einem sequentiellen Anteil der Länge c_1 und einem perfekt parallelisierbaren Anteil der Länge c_2 bestehen. Die Laufzeit $t(n)$ solcher Programme auf n Prozessoren wird dann anhand der Formel $t(n) = c_1 + c_2/n$ modelliert. In dieser Arbeit wird eine ähnliche Laufzeitformel verwendet, die allerdings zusätzlich einen logarithmischen Anteil enthält. Dieser Anteil repräsentiert den Overhead, der durch die Parallelisierung der Tasks entsteht, zum Beispiel durch Kommunikation und Synchronisierung innerhalb der parallelen Tasks. Auf diese Weise bildet die Laufzeitformel das Laufzeitverhalten gewöhnlicher paralleler Anwendungen besser ab. Ein weiterer Aspekt des betrachteten Schedulingproblems, der bei der Laufzeitmodellierung berücksichtigt werden muss, ist die Heterogenität der Zielplattform. Aufgrund der unterschiedlichen Ar-

chitekturen der eingesetzten Systeme können sich auch die Tasklaufzeiten zwischen den Systemen unterscheiden. Zur Modellierung dieser Laufzeitunterschiede enthält die Laufzeitformel zusätzlich noch einen sogenannten Leistungsfaktor. Die folgende Formel gibt die Laufzeit $t_{i,j}$ für die Ausführung des Tasks $T_i, i \in 1, \ldots, n_T$ auf p Prozessorkernen des Rechenknotens $N_j, j \in 1, \ldots, n_N$ an:

$$t_{i,j}(p) = f_j \cdot (a_i/p + b_i + c_i \cdot \log p) \tag{3.1}$$

Dabei bezeichnet f_j den Leistungsfaktor des Rechenknotens N_j. Dieser Faktor f_j modelliert die Rechenleistung des Knotens N_j relativ zu einem beliebigen, aber festen Rechenknoten $N_r, r \in 1, \ldots, n_N$. Für die Berechnung von f_j werden die sequentiellen Laufzeiten $t_{i,j}(1)$ und $t_{i,r}(1)$ des Tasks T_i auf den Rechenknoten N_j bzw. N_r gemessen und anschließend der Leistungsfaktor $f_j = t_{i,j}(1)/t_{i,r}(1)$ gebildet.
Der verbleibende Teil der Formel 3.1 beschreibt die Laufzeit von Task T_i auf dem Rechenknoten N_r. Dieser Teil setzt sich zusammen aus:

- einem parallelen Anteil a_i der Laufzeit, der mit steigender Anzahl an Prozessorkernen linear abnimmt
- einem sequentiellen Anteil b_i der Laufzeit, der unabhängig von der Anzahl an Prozessorkernen ist und daher konstant bleibt
- einem Anteil c_i für den Overhead der parallelen Ausführung (Kommunikation und Synchronisation), der logarithmisch mit der Anzahl an Prozessorkernen steigt.

Zur Bestimmung der Parameter a_i, b_i und c_i werden zunächst die Laufzeiten von Task T_i für verschiedene Anzahlen an Prozessorkernen auf dem Rechenknoten N_r gemessen. Anschließend werden die Parameter mit der Methode der kleinsten Quadrate nach dem Levenberg-Marquardt-Algorithmus[46] ermittelt. Für die Auswahl des Rechenknotens N_r empfiehlt es sich, den Rechenknoten mit der größten Anzahl verfügbarer Prozessorkerne zu wählen. Falls nötig, kann auf diese Weise jede mögliche Anzahl an Prozessorkernen in die Messung einfließen.

3.1.1 Betrachtete parallele Anwendungsprogramme

Die in dieser Arbeit vorgestellten Schedulingverfahren werden mithilfe praktisch durchgeführter Experimente analysiert und verglichen. In diesen Experimenten werden jeweils Mengen von parallelen Anwendungen anhand der erzeugten Schedules auf einem heterogenen Rechencluster ausgeführt. Betrachtet werden dabei zum einen die FEM-Simulationstasks der im Abschnitt 2.1.1 beschriebenen simulationsbasierten Optimierung. Neben dieser praxisnahen Anwendung werden zum anderen parallele Programme der SPLASH-3 Benchmark-Suite [69] als parallele Tasks verwendet. Im Folgenden werden die genutzten Anwendungen im Detail beschrieben.

Anwendungen		Kernel	
Benchmark	Beschreibung	Benchmark	Beschreibung
BARNES	Simulation eines Partikelsystems mit dem Barnes-Hut-Algorithmus. Die Anzahl der Partikel beträgt 2^{18}.	CHOLESKY	Cholesky-Faktorisierung der dünnbesetzten Matrix „tk29.O“ der Größe 13992×13992.
FMM	Simulation eines Partikelsystems mit der schnellen Multipol-Methode. Die Anzahl der Partikel beträgt 2^{19}.	FFT	eindimensionale schnelle Fourier-Transformation komplexer Datenpunkte. Anzahl Datenpunkte: 2^{24}
OCEAN	Simulation von Ozeanbewegungen mit einem rot-schwarz Gauß-Seidel-Löser auf einem 2050×2050-Gitter.	LU	LU-Faktorisierung einer dichtbesetzten Matrix. Die Größe der Matrix beträgt 4096×4096.
WATER-SPATIAL	Simulation von Wassermolekülen in einem 3D-Gitter. Die Anzahl Moleküle beträgt 32768.	RADIX	Integer-Sortierung mit der Methode Radixsort. Anzahl Integer: 50 Mio.

Tabelle 3.1: Auswahl von Anwendungen und Kernel der SPLASH-3 Benchmark-Suite, die in dieser Arbeit als parallele Tasks verwendet werden.

FEM-Simulationstasks

Zur Lösung partieller Differentialgleichungen wurde an der Professur Numerische Mathematik der Technischen Universität Chemnitz die parallele Anwendung SPC-PM3-AdH [3] entwickelt. Diese Anwendung setzt eine Finite-Elemente-Methode (FEM) ein, die im Gegensatz zu anderen Lösern eine adaptive Verfeinerung vornimmt, um eine höhere Genauigkeit zu erzielen. Für die Parallelisierung der Anwendung wurde OpenMP verwendet, wodurch die parallele Ausführung auf Systemen mit gemeinsamem Speicher ermöglicht wird. Ein FEM-Simulationstask entspricht einer Ausführung von SPC-PM3-AdH, die auf einer beliebigen Anzahl an Prozessorkernen eines Rechenknotens durchgeführt werden kann und somit einen parallelen Task darstellt. Die in dieser Arbeit betrachteten FEM-Simulationstasks verwenden SPC-PM3-AdH zur Simulation der Belastung eines Bauteils. Bei diesem Bauteil handelt es sich um den Demonstrator „Platte mit Einleger“, für den eine Krafteinwirkung simuliert wird [44].

SPLASH-3-Benchmark-Tasks

Die SPLASH-3 Benchmark-Suite enthält sowohl komplette parallele Anwendungen als auch einzelne mathematische Berechnungen, die als Kernel bezeichnet werden

und ebenfalls parallel ausgeführt werden können. Die Anwendungen repräsentieren verschiedenste naturwissenschaftliche, technische und grafische Berechnungen und Simulationen. Die Kernel entsprechen dagegen eher kleineren häufig verwendeten mathematische Berechnungen wie eine schnelle Fourier-Transformation, Matrixoperationen und Sortierverfahren. Als parallele Tasks zur Ausführung in den Messungen in dieser Arbeit wurden vier Anwendungen und vier Kernel der SPLASH-3 Benchmark-Suite gewählt. Der Fokus lag bei dieser Auswahl auf einer angemessenen und ähnlichen Ausführungszeit sowie der Möglichkeit der Ausführung auf einer beliebigen Anzahl an Prozessorkernen. Tabelle 3.1 listet die gewählten Benchmark-Tasks auf. Falls nicht anders angegeben, wurden bei den Benchmark-Tasks die vorgegebenen Parameter „parsec-simlarge“ verwendet.

3.2 Evaluierung des Kostenmodells

Für den praktischen Einsatz eines auf Messungen basierenden Kostenmodells ist es wichtig, eine möglichst hohe Genauigkeit mit möglichst wenigen zusätzlichen Messungen zu erhalten. Daher werden nachfolgend für das im vorigen Abschnitt vorgestellte Kostenmodell verschiedene Strategien zur Parameterbestimmung vorgestellt. Anschließend wird die Genauigkeit des Kostenmodells in Abhängigkeit von der Anzahl durchgeführter Messungen analysiert.

3.2.1 Parameterbestimmung

Die in Gleichung (3.1) angegebene Laufzeitformel Tasks $T_i, i \in 1, \ldots, n_T$ hängt von den drei Parametern a_i, b_i und c_i sowie dem Leistungsfaktor f_j des Rechenknotens $N_j, j \in 1, \ldots, n_N$ ab. Für die Parameterbestimmung wird zu Beginn der Rechenknoten N_r mit der größten Anzahl verfügbarer Prozessorkerne gewählt. Die Bestimmung des Leistungsfaktors erfolgt für jeden Rechenknoten und ist unabhängig von den drei anderen Parametern. Dafür werden zunächst die sequentiellen Laufzeiten $t_{i,j}(1)$ des Tasks T_i auf allen Rechenknoten $N_j, j \in 1, \ldots, n_N$ gemessen. Der jeweilige Leistungsfaktor f_j wird anschließend wie folgt als Verhältnis der sequentiellen Laufzeiten zueinander berechnet:

$$f_j = t_{i,j}(1)/t_{i,r}(1) \tag{3.2}$$

Zur Bestimmung der Parameter a_i, b_i und c_i werden die parallelen Laufzeiten $t_{i,r}(p)$ des Tasks T_i auf dem Rechenknoten N_r für verschiedene Anzahlen p von Prozessorkernen gemessen. Mit der Methode der kleinsten Quadrate werden anschließend die Parameter a_i, b_i und c_i ermittelt. Das Messen für alle Kernanzahlen der p_r Prozessorkerne des Rechenknotens N_r wäre in den meisten Fällen zu zeitaufwändig. Daher werden folgende drei Strategien zur Auswahl der Kernanzahlen für die Laufzeitmessung vorgeschlagen und miteinander verglichen:

Vollständig: Alle Anzahlen von Prozessorkernen des Rechenknotens N_r werden verwendet, wofür p_r Messungen benötigt werden.

Zweierpotenzen: Nur für die $\lfloor \log_2 p_r \rfloor + 1$ Zweierpotenzen der Kernanzahlen zwischen 1 und p_r wird eine Messung durchgeführt.

Minimal: Es werden jeweils 1, $\lfloor \frac{p_r}{2} \rfloor$ und p_j Prozessorkerne für die Messung verwendet, was der minimalen Anzahl an Messungen zur Bestimmung der drei Parameter entspricht.

Die Messungen müssen nur für Tasks mit verschiedenen Laufzeiten und Rechenknoten mit unterschiedlicher Hardware durchgeführt werden und können automatisiert vor dem Scheduling ausgeführt werden.

3.2.2 Genauigkeit des Kostenmodells

Die meisten statischen Schedulingverfahren beziehen in ihre Entscheidungen unter anderem die Kosten der Tasks ein. Die Genauigkeit dieser Kosten im Vergleich zu den tatsächlichen Kosten, z. B. der Tasklaufzeit kann das Scheduling entscheidend beeinflussen. Dennoch werden für das Scheduling oftmals keine exakten Laufzeitvorhersagen benötigt, sondern vielmehr ein korrektes Verhältnis der Kosten der Tasks untereinander. Um die Genauigkeit des vorgestellten Kostenmodells zu überprüfen, wurde das Laufzeitverhalten für parallele Tasks der SPLASH-3-Benchmark-Suite (siehe Abschnitt 3.1.1) untersucht. Für die Messungen wurde der Rechner sb1 verwendet, der über zwei Intel Xeon-Prozessoren E5-2650 mit insgesamt 16 Kernen und 32 GB RAM verfügt. Jeder Task wurde jeweils mit 1 bis 16 Threads ausgeführt und die Laufzeit gemessen. Lediglich für Tasks der Typen OCEAN, FFT und RADIX ist die Threadanzahl auf ein Vielfaches von 2 beschränkt.

Abbildung 3.1 zeigt die parallelen Laufzeiten der Anwendungen (links) und der Kernel (rechts) in Abhängigkeit der Threadanzahl. Dabei werden sowohl die gemessen Werte als Messpunkte als auch die modellierte Laufzeit für die vollständige Messstrategie gezeigt. Die Ergebnisse zeigen für alle Tasks eine parallele Laufzeit, die mit steigender Threadanzahl abnimmt. Erkennbar ist auch, dass die gewählte Laufzeitformel mit den zuvor bestimmten Parametern das Laufzeitverhalten der Tasks sehr gut abbildet. Lediglich für einzelne Threadanzahlen sind die Laufzeiten der BARNES- (4, 5, 8 und 10) und der WATER-SPATIAL-Anwendungstasks (7, 11 und 13) höher als die modellierten Laufzeiten. Dieses Verhalten lässt sich auf eine ungünstige Aufteilung der Daten insbesondere bei Gitter-basierten Berechnungen zurückführen. Die Modellierung solcher unregelmäßiger Laufzeitverhalten ist mit einer einzelnen Laufzeitformel nicht praktikabel umsetzbar, da dafür eine zu große Anzahl Parameter nötig wäre. Die parallelen Laufzeiten des CHOLESKY-Kernel sind zu kurz, um von einer höheren Threadanzahl zu profitieren. Dennoch wurde auch dieses Verhalten von der Laufzeitformel angemessen modelliert.

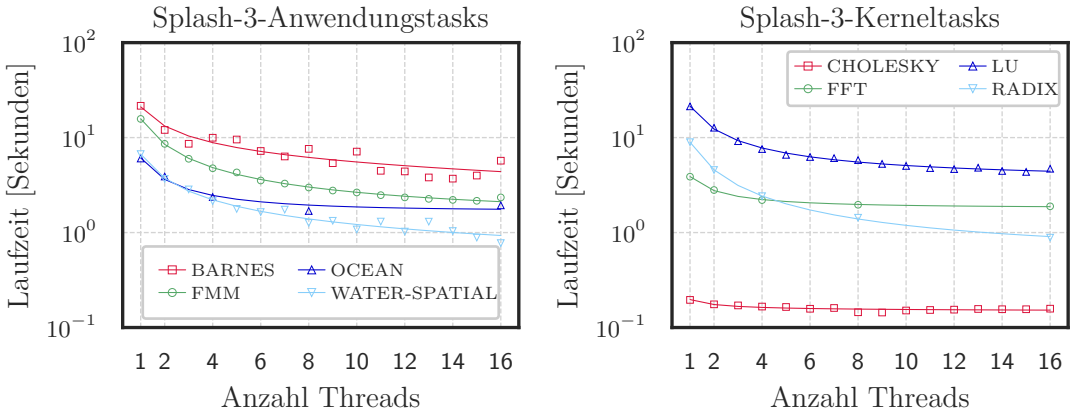


Abbildung 3.1: Gemessene (Messpunkte) und modellierte (Kurve) parallele Laufzeiten von Anwendungstasks (links) und Kerneltasks (rechts) der SPLASH-3-Benchmark-Suite auf dem Rechenknoten sb1.

	Anwendungstasks			
Messstrategie	BARNES	FMM	OCEAN	WATER-SPATIAL
Vollständig	27.8%	9.8%	15.4%	20.3%
Zweierpotenzen	70.6%	8.0%	15.4%	31.0%
Minimal	63.7%	10.1%	21.9%	31.2%
	Kerneltasks			
Messstrategie	CHOLESKY	FFT	LU	RADIX
Vollständig	8.0%	1.4%	5.8%	3.0%
Zweierpotenzen	7.7%	1.4%	8.7%	3.0%
Minimal	17.0%	2.9%	9.6%	4.1%

Tabelle 3.2: Größter Unterschied zwischen gemessener und modellierter paralleler Laufzeit für 3 verschiedene Messstrategien.

Zusätzlich zur vollständigen Messstrategie wurde die parallele Laufzeit der Tasks auch für die beiden anderen Strategien „Zweierpotenzen“ und „Minimal“ modelliert. Tabelle 3.2 zeigt für jede der drei Strategien den größten Unterschied zwischen der modellierten und der gemessenen Laufzeit. Eine Beobachtung ist, dass unabhängig von der Strategie die Unterschiede für die Anwendungstasks größer ausfallen als für die Kernel. Begründet werden kann dies damit, dass das stabilere Laufzeitverhalten der kleineren Kerneltasks besser vorhersagbar ist als für die komplexen und diversen Anwendungstasks. Die großen Unterschiede bei den BARNES-Anwendungstasks decken sich mit dem in Abbildung 3.1 erkennbaren unregelmäßigen Laufzeitverhalten. Insgesamt lässt sich feststellen, dass für fast alle Tasks die Unterschiede bei der Anwendung der vollständigen Messstrategie am kleinsten sind. Mit Ausnahme der BARNES- und der WATER-SPATIAL-Anwendungstasks wurden auch für die Messungen von Zweierpotenzen als Threadanzahlen gute Ergebnisse erreicht. Die Minimalanzahl von 3 Messungen führte bei fast allen Tasks zu den größten Unterschieden zwischen gemessener und modellierter Laufzeit. Nur bei den FMM-Anwendungstasks und den Kernels FFT und RADIX wurde bereits mit 3 Messungen ein Ergebnis erreicht, das durch weitere Messungen kaum noch verbessert wird.

4 Optimales Scheduling auf Basis des A*-Suchalgorithmus

Die A*-Suche [35] ist eine informierte Suchstrategie, die vollständig, optimal und optimal effizient ist [14]. Das bedeutet, dass die A*-Suche immer die optimale Lösung findet, falls diese existiert und dass kein anderer Algorithmus existiert, der dafür weniger Schritte benötigt. In diesem Kapitel wird die A*-Suche als Ansatz zur optimalen Lösung des in Abschnitt 2.2 beschriebenen Schedulingproblems angewendet. Als Resultat wird ein neues Verfahren für das Scheduling von parallelen Tasks auf heterogene Rechenressourcen vorgestellt. Dieses Schedulingverfahren trägt den Namen Heterogeneous Parallel task scheduling based on A* (HP*) [19]. Es wird gezeigt, wie das betrachtete Schedulingproblem als Suchraum zur Anwendung der A*-Suche formuliert werden kann. Des Weiteren werden drei Strategien zur Eingrenzung des Suchraums präsentiert, die eine effizientere Suche ermöglichen. Den Abschluss des Kapitels bildet ein Vergleich von HP* mit existierenden Schedulingverfahren anhand von Laufzeitmessungen für die Ausführung von Benchmark-Tasks auf einem heterogenen Rechencluster.

4.1 Idee des A*-Suchalgorithmus und dessen Anwendung als Schedulingverfahren

Die A*-Suche wird hauptsächlich zur Ermittlung eines kürzesten Pfades zwischen zwei Knoten in einem gewichteten Graphen eingesetzt. Gegenüber uninformierten Suchverfahren wie dem Dijkstra-Algorithmus nutzt der A*-Suchalgorithmus zusätzlich eine Schätzfunktion, um effizienter eine Lösung finden zu können. *Schätzfunktionen* sind die gebräuchlichste Form, um problemspezifisches Wissen in ein Suchverfahren zu integrieren und basieren meist auf einer weniger restriktiven Problemstellung oder gesammelten Erfahrungswerten [66]. Zumeist geben diese Schätzfunktionen eine Abschätzung der Restkosten an, die ausgehend von einer Teillösung benötigt werden, um eine optimale Lösung zu erhalten. Ein anschauliches Beispiel für die Anwendung der A*-Suche ist das Finden des kürzesten Weges zwischen zwei Städten in einem Routenplaner. In diesem Beispiel ist die exakte Entfernung nur für direkt miteinander verbunden Städte bekannt. Als Restkostenschätzung wird für jede Stadt die Luftliniendistanz zum Zielort verwendet. Während eine uninformierte Su-

che nur die direkte Entfernung zwischen benachbarten Städten berücksichtigt, kann die A*-Suche unter Berücksichtigung der Luftlinie zielgerichteter vorgehen.

Die Idee der A*-Suche ist die Auswahl des Knotens, der am schnellsten in Richtung des Zielknotens führt. Dazu werden die Knoten eines Graphen anhand einer Kostenfunktion bewertet, die durch

$$f(n) = g(n) + h(n) \tag{4.1}$$

beschrieben wird, wobei die verwendeten Funktionen im gesamten Kapitel folgende Bedeutungen haben:

Benennung	Notation	Bedeutung
Gesamtkostenschätzung	$f(n)$	erwartete Kosten des kürzesten Wegs vom Startknoten über n zum Zielknoten
Teilpfadkosten	$g(n)$	Pfadkosten des bereits ermittelten kürzesten Wegs vom Startknoten zum Knoten n
Restkostenschätzung	$h(n)$	Schätzfunktion für die Kosten vom Knoten n zum Zielknoten

Die Suche beginnt im Startknoten, von dem aus schrittweise weitere Knoten entdeckt und besucht werden. Während der Suche können die Knoten in die folgenden drei Gruppen eingeteilt werden:

- Zu *unbekannten Knoten* ist noch kein Weg bekannt.
- Zu *entdeckten Knoten* wurde bereits ein Weg gefunden, der jedoch nicht zwangsläufig der kürzeste Weg ist.
- *Besuchte Knoten* wurden vom Algorithmus verarbeitet und damit der kürzeste Weg zu diesen Knoten ermittelt.

Zu Beginn der A*-Suche ist nur der Startknoten bekannt. Ausgehend vom Startknoten wird in jeder Iteration der Knoten besucht, der die geringste Gesamtkostenschätzung aller entdeckten Knoten aufweist. Besitzt der besuchte Knoten unbekannte Nachbarknoten, so werden diese den entdeckten Knoten zugeordnet und die Gesamtkostenschätzung berechnet. Anschließend wird mit der Auswahl des Knotens mit der geringsten Gesamtkostenschätzung aller bisher entdeckten Knoten eine neue Iteration begonnen. Mit dem Besuch des Zielknotens wurde der kürzeste Weg vom Startknoten aus gefunden und die A*-Suche endet.

4.1.1 Beispiel für das Finden eines kürzesten Wegs mithilfe der A*-Suche

Anhand eines Beispiels soll der Ablauf der A*-Suche für das Finden eines kürzesten Wegs veranschaulicht werden. Betrachtet wird die Planung einer Route zwischen

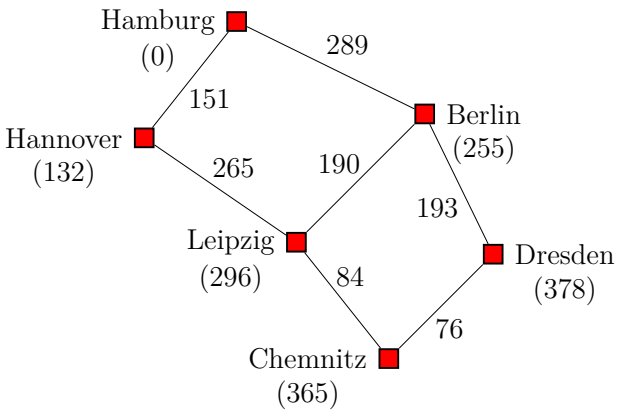
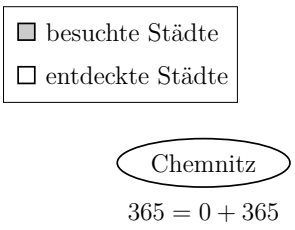


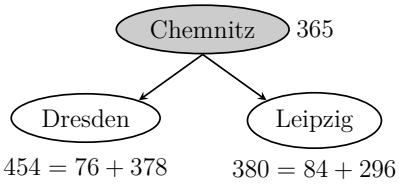
Abbildung 4.1: Vereinfachte Straßenkarte deutscher Städte. Die Kanten bezeichnen direkte Verbindungen unter Angabe der Entfernung. Die Werte in Klammern entsprechen der Luftliniendistanz nach Hamburg.

zwei Städten, wobei nur die Entfernung zwischen benachbarten Städten sowie die Luftliniendistanz zum Zielort bekannt sind. Abbildung 4.1 zeigt eine vereinfachte Straßenkarte, in der die direkten Verbindungen zwischen den Städten durch Kanten gekennzeichnet sind. Die Kantengewichte bezeichnen die Entfernungen zwischen den Städten, die zur Berechnung der Pfadkosten genutzt werden. Gesucht ist der kürzeste Weg von `Chemnitz` nach `Hamburg`. Als Restkostenschätzung wird die Luftliniendistanz zum Zielort `Hamburg` verwendet, die als Wert in Klammern für jede Stadt angegeben ist.

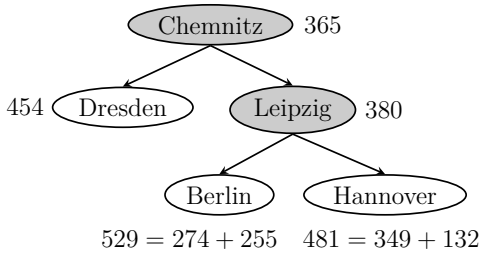
In der Abbildung 4.2 sind die Städte zu sehen, die jeweils in den Iterationen der A*-Suche entweder entdeckt oder besucht wurden. Darin werden entdeckte Städte weiß und besuchte Städte grau dargestellt. Mithilfe der Kanten wird gekennzeichnet, von welcher Stadt aus eine Stadt entdeckt wurde. Auf diese Weise lässt sich am Ende der Suche der kürzeste Weg rekonstruieren. Abbildung 4.2a zeigt die Situation zu Beginn der Suche. Lediglich der Startort `Chemnitz` und dessen Gesamtkostenschätzung sind bekannt. Da bisher noch kein Weg zurückgelegt wurde, sind die Teilpfadkosten 0 und die Gesamtkostenschätzung entspricht der Restkostenschätzung, wodurch sich ein Wert von 365 ergibt. In dieser Iteration ist `Chemnitz` die einzige Stadt und somit die Stadt mit der geringsten Gesamtkostenschätzung. Daher folgt, wie in Abbildung 4.2b zu sehen ist, der Besuch der Stadt `Chemnitz`. Von `Chemnitz` aus werden dabei die Städte `Dresden` und `Leipzig` entdeckt und deren Gesamtkostenschätzungen ermittelt. Diese setzen sich jeweils aus den von `Chemnitz` ausgehenden Teilpfadkosten und der Luftliniendistanz nach `Hamburg` zusammen. Für die Strecke von `Chemnitz` nach `Dresden` betragen die Pfadkosten 76 und von `Chemnitz` nach `Leipzig` 84. Zu diesen Werten wird die von der Restkostenschätzung betrachtete Luftliniendistanz nach `Hamburg` addiert. Für `Dresden` beträgt diese 378 und für `Leipzig` 296. Somit



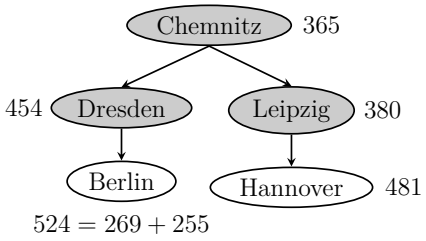
(a) Initialisierung des Startorts `Chemnitz` als entdeckte, aber noch nicht besuchte Stadt.



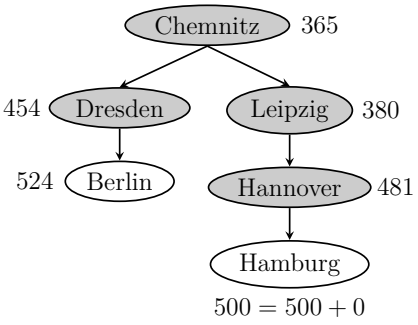
(b) Besuch von `Chemnitz` und der Entdeckung der Nachbarstädte `Dresden` und `Leipzig`.



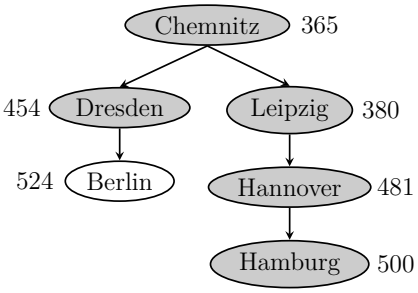
(c) Besuch von `Leipzig` und der Entdeckung der Nachbarstädte `Berlin` und `Hannover`.



(d) Besuch von `Dresden` und der Entdeckung eines kürzeren Wegs nach `Berlin`.



(e) Besuch von `Hannover` und der Entdeckung der Nachbarstadt `Hamburg`.



(f) Ende der A*-Suche nach dem Besuch des Zielorts `Hamburg` mit kürzestem Weg über Leipzig und Hannover.

Abbildung 4.2: Darstellung der durch die A*-Suche in den jeweiligen Iterationsschritten entdeckten und besuchten Städte für das Finden eines kürzesten Wegs von `Chemnitz` nach `Hamburg` im Beispiel aus Abbildung 4.1. Angabe der Kosten als Summe aus den von Chemnitz ausgehenden Teilpfadkosten und der Restkostenschätzung (Luftlinie) für den Zielort `Hamburg`.

wurden am Ende der Iteration die Städte `Dresden` mit den erwarteten Gesamtkosten 454 und `Leipzig` mit den erwarteten Gesamtkosten 380 entdeckt.

In der nächsten Iteration wird `Leipzig` als Stadt mit der geringsten Gesamtkostenschätzung ausgewählt und besucht. Dabei werden die Städte `Berlin` und `Hannover` entdeckt und deren Gesamtkostenschätzung berechnet. An dieser Stelle wird der Einfluss der Schätzfunktion deutlich. So hätte ein Algorithmus ohne Schätzfunktion aufgrund der kürzeren direkten Verbindung zunächst `Dresden` besucht und sich damit weiter vom Zielort `Hamburg` entfernt. Abbildung 4.2c zeigt die Situation nach dem Besuch der Stadt Leipzig. Zu den Pfadkosten von `Chemnitz` nach `Leipzig` von 84 addieren sich die Pfadkosten von 190 nach `Berlin` bzw. 265 nach `Hannover` und die jeweiligen Luftliniendistanzen nach `Hamburg`. Am Ende der Iteration wird `Leipzig` als besucht markiert, so dass `Dresden` (Gesamtkostenschätzung 454), `Berlin` (Gesamtkostenschätzung 529) und `Hannover` (Gesamtkostenschätzung 481) die aktuell entdeckten Städte sind.

Die Stadt `Dresden` besitzt nun die geringste Gesamtkostenschätzung aller entdeckten Städte und wird daher als nächstes von der A*-Suche besucht. Dabei wird erneut `Berlin` entdeckt, jedoch mit einer geringeren Gesamtkostenschätzung als zuvor. Da die Gesamtkostenschätzung für `Berlin` für die Route über `Dresden` geringer ausfällt, wird die Route nach `Berlin` über `Leipzig` im weiteren Verlauf nicht mehr betrachtet. Im Anschluss wird die aktuelle Iteration beendet, indem `Dresden` als besucht markiert wird. In Abbildung 4.2d ist die resultierende Situation dargestellt.

Die A*-Suche wählt als nächstes die Stadt `Hannover` aus, die eine geringere Gesamtkostenschätzung im Vergleich zu `Berlin` besitzt. `Hannover` wird daraufhin als besucht markiert und die Nachbarstadt `Hamburg` zu den bisher entdeckten Städten hinzugefügt. Da `Hamburg` der Zielort ist, liegt der Wert der Restkostenschätzung bei 0 und die Gesamtkosten entsprechen den aufsummierten Pfadkosten von `Chemnitz` über `Leipzig` und `Hannover`. Abbildung 4.2e zeigt die aktuell entdeckten und besuchten Städte mit deren Gesamtkostenschätzung. Der Zielort `Hamburg` wurde zwar entdeckt, jedoch endet der Algorithmus erst nach dem Besuch des Ziels, da möglicherweise ein anderer kürzerer Weg dorthin existieren könnte.

Die finale Iteration der A*-Suche für das gezeigte Beispiel ist in Abbildung 4.2f zu sehen. Von den beiden entdeckten Städten `Berlin` und `Hamburg` besitzt `Hamburg` die geringere Gesamtkostenschätzung und wird daher von der A*-Suche ausgewählt. Der Weg über `Berlin` nach `Hamburg` kann nicht kürzer sein, da die Restkostenschätzung stets die tatsächlichen Kosten unterschätzt und somit die Gesamtkostenschätzung entlang eines Pfades niemals abnehmen kann. Mit dem Besuch des Zielorts endet der Algorithmus und der kürzeste Weg von `Chemnitz` nach `Hamburg` wurde gefunden. Dieser führt über `Leipzig` und `Hannover` und hat die Gesamtkosten 500.

4.1.2 Idee zur Anwendung der A*-Suche als Schedulingverfahren

Die A*-Suche wurde bereits zur Lösung von Schedulingproblemen eingesetzt [39, 45, 70]. Allerdings beschäftigen sich diese Arbeiten mit dem Scheduling sequentieller Tasks, wobei jeder Task genau einem Prozessor zugewiesen wird. Der verwendete Ansatz zur Anwendung der A*-Suche als Schedulingverfahren lässt sich jedoch auch auf weitere Schedulingprobleme übertragen. Zur Lösung des in dieser Arbeit betrachteten Schedulingproblems für parallele Tasks wird ebenfalls dieser Ansatz verwendet, der im Folgenden kurz vorgestellt wird.

Zur Verwendung der A*-Suche als Schedulingverfahren wird zunächst eine Formulierung des Schedulingproblems als Graphstruktur vorgeschlagen. Dieser Graph hat eine Baumstruktur und wird während der Suche sukzessive aufgebaut. In einem solchen Baum entsprechen die Knoten partiellen Schedules, in denen jeweils eine Teilmenge der Tasks den Prozessoren zugewiesen wurde. Die Wurzel des Baums wird durch einen leeren Schedule repräsentiert, in dem noch kein Task zugewiesen wurde. In jeder Ebene wird ein weiterer Task zugewiesen, so dass schlussendlich die Blattknoten vollständigen Schedules entsprechen, in denen alle Tasks zugewiesen wurden. Somit gibt die Tiefe eines Schedules im Baum an, wie viele Tasks bereits den Prozessoren zugewiesen wurden.

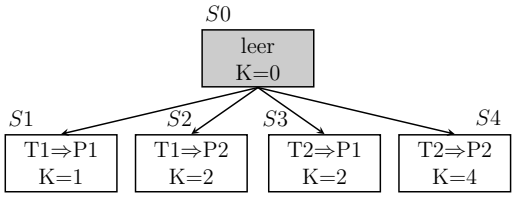
Der Startknoten für die A*-Suche ist die Wurzel des Baums und Zielknoten sind alle vollständigen Schedules in den Blättern. Die konkrete Berechnung der Gesamtkostenschätzung der Knoten anhand derer die A*-Suche ihre Entscheidungen trifft, kann je nach betrachtetem Schedulingproblem variieren. Zumeist entsprechen die Teilpfadkosten jedoch dem Makespan der partiellen Schedules, während die Restkostenschätzung aus den Kosten der verbleibenden Tasks ermittelt wird.

Ausgehend von der Wurzel wird in jeder Iteration jeweils der entdeckte Knoten mit der geringsten Gesamtkostenschätzung besucht. Der Schedule dieses Knotens wird erweitert, indem jeder verbleibende Task separat jedem Prozessor zugewiesen wird. Jede einzelne dieser Zuweisungen erzeugt aus dem ursprünglichen Schedule einen neuen partiellen Schedule, der genau eine weitere Zuweisung eines Tasks enthält. Die so erzeugten Schedules bilden die Nachfolger des besuchten Knotens im Baum und werden den entdeckten Knoten zugeordnet. Das Verfahren endet, sobald ein Blattknoten besucht wird und so die Lösung in Form eines vollständigen Schedules gefunden wurde.

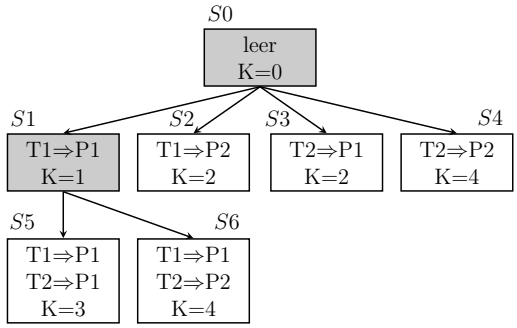
Abbildung 4.3 zeigt die beschriebene Anwendung der A*-Suche auf das Scheduling an einem Beispiel. In diesem Beispiel sollen die zwei sequentiellen Tasks $T1$ und $T2$ den zwei Prozessoren $P1$ und $P2$ derart zugewiesen werden, dass die Gesamtkosten minimal werden. Die Kosten der Tasks $T1$ und $T2$ auf den Prozessoren $P1$ und $P2$ sind in Abbildung 4.3a zu sehen. In Abbildung 4.3b ist die Situation zu Beginn des Verfahrens dargestellt. Der initiale leere Schedule $S0$ wird besucht, wobei ausgehend von $S0$ vier neue Schedules $S1, \ldots, S4$ erzeugt werden, indem jeder Task separat jedem Prozessor zugewiesen wird.

	Prozessoren	
Tasks	P1	P2
T1	1	2
T2	2	4

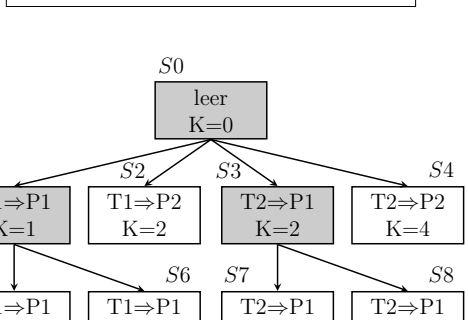
(a) Kosten der Tasks $T1$ und $T2$ auf den Prozessoren $P1$ und $P2$.



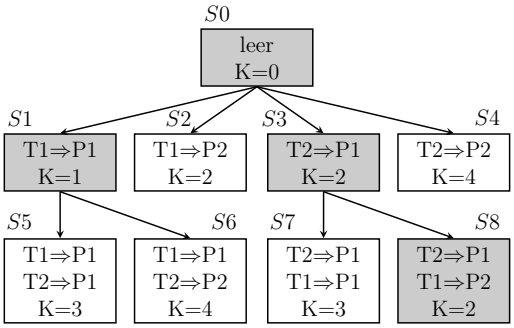
(b) Besuch des initialen leeren Schedules $S0$ und Entdeckung neuer partieller Schedules durch separate Zuweisung jedes Tasks zu jedem Prozessor.



(c) Besuch des Schedules $S1$ und Entdeckung der Schedules $S5$ und $S6$.



(d) Besuch des Schedules $S3$ und Entdeckung der Schedules $S7$ und $S8$.



(e) Ende der A*-Suche mit dem Besuch des vollständigen Schedules $S8$.

Abbildung 4.3: Beispiel zur Anwendung der A*-Suche auf das Scheduling zweier sequentieller Tasks $T1$ und $T2$ auf zwei Prozessoren $P1$ und $P2$.

In der darauffolgenden Iteration wird der entdeckte partielle Schedule $S1$ mit den geringsten Gesamtkosten $K = 1$ besucht. Aus diesem Schedule $S1$ werden zwei neue Schedules $S5$ und $S6$ erzeugt, indem zusätzlich der verbleibende Task $T2$ jeweils einem der beiden Prozessoren zugewiesen wird. Der resultierende Baum mit nun 2 besuchten und 5 entdeckten Schedules ist in Abbildung 4.3c zu sehen.

Abbildung 4.3d zeigt die Situation am dem Ende der nächsten Iteration. Der partielle Schedule $S3$ mit den geringsten Gesamtkosten $K = 2$ wurde besucht. Durch Hinzufügen der Zuweisung des verbleibenden Tasks $T1$ zu je einem der beiden Prozessoren wurden zwei neue Schedules $S7$ und $S8$ generiert.

In einer weiteren Iteration wird erneut der Schedule mit den geringsten Gesamtkosten besucht. Bei diesem Schedule $S8$ handelt es sich um einen vollständigen Schedule mit den Gesamtkosten $K = 2$, in dem der Task $T1$ dem Prozessor $P2$ und der Task $T2$ dem Prozessor $P1$ zugewiesen wurde. Somit wurde eine Lösung des Schedulingproblems gefunden und das Verfahren terminiert.

Im Folgenden werden Arbeiten vorgestellt, die sich mit der Anwendung der A*-Suche auf Schedulingprobleme für sequentielle Tasks befassen. Anschließend wird zunächst der A*-Suchalgorithmus vorgestellt, bevor mit dem Verfahren HP* eine Möglichkeit präsentiert wird, die A*-Suche für das Scheduling von parallelen Tasks auf heterogene Rechenressourcen einzusetzen.

4.2 Anwendung der A*-Suche auf weitere Schedulingprobleme

Der A*-Suchalgorithmus wurde bereits als Lösungsansatz für verschiedene Schedulingprobleme betrachtet. Im Folgenden werden zentrale Arbeiten zu diesem Thema vorgestellt. Alle diese Arbeiten beschäftigen sich jedoch mit dem Scheduling sequentieller Tasks mit Abhängigkeiten. In dieser Arbeit werden die darin vorgestellten Methoden aufgegriffen und für das Scheduling unabhängiger paralleler Tasks angepasst. Das Scheduling unabhängiger paralleler Tasks ist im Vergleich jedoch komplexer, da deutlich mehr Zuweisungen möglich sind und folglich eine größere Anzahl potentieller Lösungen zu betrachten ist.

Kafil und Ahmad [39]

Bei [39] handelt es sich um eine der ersten Arbeiten, die den A*-Algorithmus für die Zuweisung von Tasks zu Prozessoren nutzen. Darin werden drei Algorithmen zur optimalen und suboptimalen Lösung vorgestellt. Als Rechenressource wird eine heterogene Menge von Prozessoren betrachtet, die über ein Verbindungsnetzwerk miteinander kommunizieren können. Die Prozessoren sind dabei in einem Gitter angeordnet. Im behandelten Schedulingproblem soll diesen Prozessoren eine Menge von sequentielle Tasks zugewiesen werden. Zwischen den Tasks können Kommunikationsabhängigkeiten vorliegen, daher sind die Tasks als Task-Graph gegeben, in dem diese Abhängigkeiten abgebildet sind. Neben den Kosten der Ausführung eines Tasks auf einem Prozessor müssen die zusätzlichen Kommunikationskosten für den Datenaustausch zwischen Prozessoren berücksichtigt werden.

Es werden ein paralleler Algorithmus, ein Clustering-Algorithmus und ein suboptimaler Algorithmus vorgestellt, die alle auf dem A*-Suchalgorithmus basieren. Der Suchraum aller drei Algorithmen wird als Baum beschrieben, dessen Wurzel ein leerer Schedule ist. Jeder innere Knoten entspricht einem partiellen Schedule und in den Blättern befinden sich vollständige Schedules. Die Anzahl der Ebenen im Baum

entspricht der Taskanzahl, da in jeder Ebene ein weiterer Task zum Schedule hinzugefügt wird. Zur Erzeugung eines Kindknotens zu einem Knoten im Baum wird dem entsprechenden Schedule ein weiterer Task hinzugefügt. Der Task wird dabei jedem Prozessor separat zugewiesen, so dass die Anzahl der Kindknoten für jeden Knoten gleich der Anzahl vorhandener Prozessoren ist. Analog zum A*-Suchalgorithmus werden die Entscheidungen auf Basis einer Gesamtkostenschätzung getroffen. Dabei entsprechen die Teilpfadkosten dem Makespan des (partiellen) Schedules, während in die Berechnung der Restkostenschätzung die Kosten der Tasks einfließen, die noch nicht zugewiesen wurden. Da die tatsächlichen Kosten dieser verbleibenden Tasks unbekannt sind, werden für die Restkostenabschätzung die minimalen Kosten für deren Ausführung und Kommunikation verwendet.

Im präsentierten parallelen Algorithmus arbeitet jeder Prozess auf einer Teilmenge der Knoten des Baums. Ohne Kommunikation würden auf diese Weise zusätzliche Bereiche durchsucht, die keine optimale Lösung enthalten. Dieser Mehraufwand wird durch den Austausch von Knoten vermieden. Findet ein Prozess eine Lösung, so teilt er diese allen anderen Prozessen mit. Diese suchen daraufhin ausschließlich nach besseren Lösungen und terminieren, falls die Kosten ihrer aktuell betrachteten Lösung größer sind.

Der Clustering-Algorithmus fasst Tasks zusammen, die aufgrund ihrer hohen Kommunikationskosten demselben Prozessor zugewiesen werden sollten. Die zusammengefassten Tasks werden nach ihren Ausführungs- und Kommunikationskosten sortiert und in dieser Reihenfolge den Prozessoren zugewiesen. Eine optimale Lösung kann mit dieser Vorgehensweise allerdings nur garantiert werden, wenn alle Prozessoren im Netzwerk direkt miteinander verbunden sind.

Im suboptimalen Algorithmus werden ab einer bestimmten Suchtiefe Lösungen ignoriert. Auf diese Weise können Lösungen schneller gefunden und der hohe Speicherbedarf des A*-Algorithmus bewältigt werden. Mithilfe verschiedener Heuristiken werden Teile des Suchraums ignoriert, wodurch zwar die Laufzeit des Algorithmus reduziert wird, aber eine Optimalität nicht mehr gegeben ist. So werden unter anderem jeweils nur die besten Teillösungen betrachtet oder die schlechtesten Teillösungen verworfen.

Kwok und Ahmad [45]

In [45] werden neben einem sequentiellen Schedulingverfahren auch ein paralleler Algorithmus und ein Approximationsverfahren vorgestellt, die auf der A*-Suche basieren. Im betrachteten Schedulingproblem soll eine Menge von sequentiellen Tasks, zwischen denen Kommunikationsabhängigkeiten vorliegen können, derart einer Menge von Prozessoren zugewiesen werden, dass der Makespan minimiert wird. Die Tasks sind in Form eines Task-Graphen gegeben, der die Abhängigkeiten zwischen den Tasks abbildet. Die Prozessoren sind über ein Netzwerk miteinander verbunden

und können verschiedene Eigenschaften haben, wodurch ein heterogenes paralleles System gebildet wird.

Für eine Anwendung des A*-Suchalgorithmus wird das Schedulingproblem zunächst als Zustandsraum beschrieben. Dabei entspricht jeder Zustand einem partiellen Schedule, in dem nur eine Teilmenge der Tasks den Prozessoren zugewiesen wurde. Der Anfangszustand, bei dem die Suche beginnt, wird durch einen leeren Schedule repräsentiert. Mithilfe eines Expansionsoperators wird ein Zustand erweitert, indem jeder verfügbare Task jeweils separat jedem Prozessor zugewiesen wird. Ein Task ist verfügbar, wenn alle seine Vorgänger im Task-Graphen zugewiesen wurden. Der Expansionsoperator erzeugt dabei für jede Zuweisung eines Tasks einen neuen Zustand, in dem genau ein Task mehr zugewiesen wurde. Ein Zielzustand ist erreicht, wenn alle Tasks zugewiesen wurden und somit ein vollständiger Schedule vorliegt. In einem solchen Zustand liegt eine Lösung des Schedulingproblems vor und die Suche endet.

Die Gesamtkostenschätzung für einen Zustand setzt sich aus den Teilpfadkosten und einer Restkostenschätzung zusammen. Die Teilpfadkosten eines Zustands entsprechen dabei dem Makespan des jeweiligen partiellen Schedules, das heißt der größten Endezeit der zugewiesenen Tasks. Der Task mit der größten Endezeit wird auch für die Berechnung der Restkostenschätzung verwendet. Durch die Abhängigkeiten im Task-Graphen wird für bestimmte Tasks eine Reihenfolge festgelegt, in der diese ausgeführt werden müssen. Die Restkostenschätzung entspricht der größten Summe der Ausführungszeiten aller solchen Abfolgen ab dem ermittelten Task.

Im sequentiellen Algorithmus wird ausgehend vom Startzustand in jedem Schritt der Zustand mit den geringsten Kosten aller unbearbeiteten Zustände gewählt. Handelt es sich beim gewählten Zustand um einen Zielzustand, stoppt der Algorithmus, ansonsten wird der Zustand expandiert, als bearbeitet markiert und erneut der Zustand mit den geringsten Kosten gewählt.

Im vorgestellten parallelen Algorithmus expandieren die Prozesse parallel unterschiedliche Zustände. Die besten der erzeugten Zustände werden im Round-Robin-Verfahren zwischen den Prozessen ausgetauscht. Zusätzlich wird bei dem Austausch auf eine gleichmäßige Verteilung der erzeugten Zustände geachtet. Sobald ein Prozess eine Lösung gefunden hat, wird diese allen Prozessen mitgeteilt, so dass diese ihre Suche beenden können.

Der Approximationsalgorithmus basiert auf einem in [58] vorgestellten Verfahren. Die entdeckten Zustände werden auf diejenigen Zustände begrenzt, deren Gesamtkostenschätzung nicht größer ist als die Kosten des aktuellen Zustands multipliziert mit einem Faktor $(1+\epsilon)$. Auf diese Weise wird eine Lösung gefunden deren Gesamtkosten die der optimalen Lösung um nicht mehr als das $(1+\epsilon)$-fache übersteigt. Es wird gezeigt, dass dieser Algorithmus selbst für kleine ϵ deutlich schneller eine Lösung findet als der sequentiellen Algorithmus.

Zusätzlich zu den Algorithmen werden drei Strategien vorgestellt, mit denen der Zustandsraum eingegrenzt werden kann. Die erste vorgestellte Strategie vermeidet das Erzeugen von Schedules mit gleichem Makespan. So können Fälle auftreten, in denen Tasks dieselben Abhängigkeiten besitzen und so zu Schedules mit gleichem Makespan führen. Von diesen Schedules wird vom Expansionsoperator jeweils nur ein Schedule erzeugt und die anderen verworfen. In einer weiteren Strategie wird vor der Ausführung des eigentlichen Verfahrens ein schnelles heuristisches Schedulingverfahren verwendet, um einen Makespan zu ermitteln. Dieser Makespan wird vom Expansionsoperator genutzt, um alle Zustände auszuschließen, deren partielle Kosten größer als dieser Makespan sind. Die dritte Strategie vermeidet die Erzeugung äquivalenter Zustände für homogene parallele Systeme. So kann die vom Expansionsoperator durchgeführte Zuweisung eines Tasks zu jedem Prozessor zu einer Menge von Zuständen führen, die sich lediglich in der Permutation der Prozessoren unterscheiden. Derartige äquivalente Zustände können vom Expansionsoperator erkannt und ausgeschlossen werden.

Sinnen und Shahul [70]

Die Arbeit von Kwok und Ahmad [45] wird in [70] erneut aufgegriffen. Anstelle von heterogenen parallelen Systemen werden jedoch homogene Systeme betrachtet. Die Beschreibung des Zustandsraums sowie des Schedulingverfahrens wurden ebenfalls aus [45] übernommen. Die Neuheiten liegen in der Präsentation einer verbesserten Funktion zur Berechnung der Gesamtkostenschätzung und weiterer Strategien zur Eingrenzung des Zustandsraums.

In die Berechnung der Gesamtkostenschätzung eines Zustands werden alle Tasks einbezogen, wohingegen in [45] nur der Task mit der größten Endezeit berücksichtigt wurde. So entspricht die Gesamtkostenschätzung dem größten *bottom level* aller zugewiesenen Tasks. Als bottom level eines Tasks wird die größte Summe der Ausführungszeiten aller Tasks bezeichnet, die ausgehend von diesem Task entlang eines Pfades im Task-Graphen liegen.

Eine der vorgestellten Strategien zur Eingrenzung des Zustandsraums greift die Erzeugung äquivalenter Zustände für homogene parallele Systeme aus [45] auf. Dabei wird eine Normalisierung der Schedules vorgeschlagen, mithilfe derer äquivalente Schedules erkannt und bis auf ein Exemplar übersprungen werden können. Weitere Strategien und Optimierungen für die Lösung des betrachteten Schedulingproblems mit der A*-Suche werden in [72] und [56] vorgestellt. In einer weiteren Arbeit [55] werden das vorgestellte Schedulingverfahren und dessen Optimierungen auf das Scheduling mit task duplication angewendet. Bei diesem Schedulingproblem können Tasks mehrfach im Schedule vorkommen.

Algorithmus 1: A*-Suchalgorithmus nach [35].

```
Eingabe: • gerichteter, gewichteter Graph mit Startknoten s und einer Menge von
           Zielknoten Z
Ausgabe: • kürzester Weg vom Start- zum Zielknoten
Initialisierung von s als entdeckter Knoten
while (es sind entdeckte Knoten vorhanden) do
    Besuch des entdeckten Knoten n mit der geringsten Gesamtkostenschätzung
    Zuordnung von n zu den besuchten Knoten
    if (n ∈ Z) then // ein Zielknoten wurde besucht
        Rekonstruktion des kürzesten Wegs über die Vorgängerrelation
        Terminierung des Algorithmus mit Ausgabe des kürzesten Wegs
    end
    else
        foreach (jeden Nachfolger u von n) do
            if (u wurde noch nicht besucht) then // unbekannten Knoten entdeckt
                Hinzufügen von u zu den entdeckten Knoten
                Vorgänger von u ist n
            end
            else if (die Gesamtkostenschätzung f(u) ist geringer als zuvor) then
             // ein kürzerer Weg zu einem bereits besuchten Knoten wurde gefunden
                Hinzufügen von u zu den entdeckten Knoten
                Vorgänger von u ist n
            end
        end
    end
end
```

4.3 Der A*-Suchalgorithmus

Die A*-Suche verfolgt das Ziel den kürzesten Weg von einem Startknoten zu einer nichtleeren Menge von Zielknoten zu finden. Für die Bewertung eines Knotens n wird die Gesamtkostenschätzung $f(n)$ verwendet, die die erwarteten Kosten eines Wegs vom Startknoten über einen Knoten n zu einem Zielknoten angibt. Die Gesamtkostenschätzung eines Knotens n setzt sich zusammen aus den Teilpfadkosten $g(n)$, die die Kosten des kürzesten Wegs vom Startknoten nach n bezeichnen und einer Restkostenschätzung $h(n)$, die die geschätzten Kosten des kürzesten Wegs von n zu einem Zielknoten angibt. Ausgehend vom Startknoten werden schrittweise weitere Knoten entdeckt und besucht. Die Knoten des Graphen werden während der Suche in *unbekannte Knoten*, *entdeckte Knoten* und *besuchte Knoten* unterteilt.

Algorithmus 1 zeigt den Ablauf der A*-Suche nach [35]. Zu Beginn ist dem Algorithmus nur der Startknoten als entdeckter Knoten bekannt. Anschließend wird

solange iteriert, bis entweder ein Zielknoten besucht wurde oder kein Knoten mehr vorhanden ist, der entdeckt, aber noch nicht besucht wurde. Im letzteren Fall endet der Algorithmus ohne Lösung, da kein Weg vom Start- zum Zielknoten existiert. In jeder Iteration wird aus den entdeckten Knoten der Knoten n mit der geringsten Gesamtkostenschätzung ausgewählt und besucht. Der Algorithmus terminiert, wenn es sich bei dem besuchten Knoten um einen Zielknoten handelt. Der so gefundene kürzeste Weg kann anschließend in umgekehrter Reihenfolge über die Vorgängerrelationen rekonstruiert werden. Beginnend mit dem Zielknoten werden dazu wiederholt die gespeicherten Vorgänger zum kürzesten Weg hinzugefügt, bis der Startknoten erreicht ist.

Handelt es sich beim besuchten Knoten n nicht um einen Zielknoten, so werden alle dessen Nachfolger u betrachtet. Jeder Nachfolger, der noch nicht besucht wurde, wird den entdeckten Knoten zugeordnet und dessen Vorgänger n gespeichert. Wenn für einen bereits besuchten Knoten eine geringere Gesamtkostenschätzung berechnet wird, dann wurde ein kürzerer Pfad zu diesem Knoten gefunden. In diesem Fall muss der Knoten erneut besucht werden und wird daher den entdeckten Knoten und dem Vorgänger n zugeordnet. Nachdem alle Nachfolger betrachtet wurden, beginnt eine neue Iteration mit der Auswahl des entdeckten Knotens, der nun die geringste Gesamtkostenschätzung aufweist.

Der A*-Suchalgorithmus findet eine optimale Lösung, falls diese existiert und die eingesetzte Restkostenschätzung *zulässig* und *konsistent* ist.

Definition 4.1 (Zulässige Restkostenschätzung)**.** Eine Restkostenschätzung $h(n)$ wird als zulässig bezeichnet, wenn die geschätzten Kosten $h(n)$ für jeden Knoten n im Graphen die tatsächlichen Kosten $h^*(n)$ unterschätzen und somit gilt:

$$h(n) \leq h^*(n) \;\; \forall \text{ Knoten } n \tag{4.2}$$

Definition 4.2 (Konsistente Gesamtkostenschätzung)**.** Die Gesamtkostenschätzung $f(n) = g(n) + h(n)$ wird als konsistent bezeichnet, wenn $f(n)$ entlang des Wegs vom Startknoten zu einem Zielknoten niemals abnimmt. Dies ist der Fall, wenn für jeden Knoten x und dessen Nachfolger y gilt:

$$g(x) + h(x) \leq g(y) + h(y) \tag{4.3}$$

Die Luftlinie beispielsweise ist eine zulässige Restkostenschätzung, da eine gerade Linie zwischen zwei Städten die kürzeste Entfernung abbildet und sie so die Länge des tatsächlichen Wegs nie übertrifft. In [66] wird mithilfe der Dreiecksungleichung [41] gezeigt, dass die Gesamtkostenschätzung für nicht-negative Pfadkosten und die Restkostenschätzung in Form der Luftlinie konsistent ist.

4.4 HP* – Scheduling paralleler Tasks mithilfe des A*-Suchalgorithmus

Der A*-Suchalgorithmus kann auch zur Lösung von Task-Scheduling-Problemen genutzt werden. In diesem Abschnitt wird beschrieben, wie der A*-Suchalgorithmus für das Scheduling unabhängiger paralleler Tasks verwendet werden kann. Das in Abschnitt 2.2 beschriebene Schedulingproblem wird dazu zunächst als gerichteter Graph formuliert. Mit Heterogeneous Parallel task scheduling based on A* (HP*) [18] wird ein auf dem A*-Algorithmus basierendes Verfahren für das Scheduling paralleler Tasks vorgestellt und dessen Optimalität gezeigt. Des Weiteren wird die Komplexität von HP* analysiert und es werden Methoden vorgestellt, um den Suchraum von HP* einzugrenzen. Die verwendete Notation zur Beschreibung des Algorithmus ist in Tabelle 4.1 aufgelistet.

4.4.1 Beschreibung des Suchraums für das Scheduling unabhängiger paralleler Tasks

Der A*-Suchalgorithmus sucht in einem gerichteten Graphen den kürzesten Weg von einem Startknoten zu einer Menge von Zielknoten. Der A*-Algorithmus kann somit auch zur Lösung von Schedulingproblemen eingesetzt werden, die als kürzeste-Wege-Problem beschrieben werden können. Für das Scheduling von sequentiellen Tasks mit Abhängigkeiten auf ein homogenes paralleles System [45] bzw. für heterogene Systeme [72] wurden bereits derartige Verfahren vorgestellt. In diesen Arbeiten wird das Schedulingproblem als Suche in einem Graph von miteinander verbundenen partiellen Schedules beschrieben. Dieser Ansatz wird im Folgenden für das Scheduling unabhängiger paralleler Tasks auf heterogene parallele System angewendet.

Zur Abbildung des in Abschnitt 2.2 beschriebenen Schedulingproblems als gerichteter Graph wird ein Baum konstruiert, der im Folgenden als *Scheduling Decision Tree* bezeichnet wird. Der Aufbau eines Scheduling Decision Tree bildet die inkrementelle Konstruktion eines vollständigen Schedules ab.

Definition 4.3 (Scheduling Decision Tree (SDT))**.** Ein Scheduling Decision Tree bezeichnet einen gerichteten Graphen mit Baumstruktur, dessen Knoten partielle Schedules sind und besitzt folgende Eigenschaften:

- Die Wurzel entspricht einem leeren Schedule.
- Die Nachfolger eines Schedules sind jeweils alle Schedules, in denen genau ein weiterer Task einer Menge von Rechenknoten zugewiesen wurde.
- In den Blättern des Baums befinden sich alle vollständigen Schedules.
- Das Gewicht $d(x, y)$ einer Kante von einem Schedule S_x zu einem Schedule S_y entspricht der Differenz der Makespans $M(S_x)$ und $M(S_y)$, das heißt $d(x, y) = M(S_y) - M(S_x)$.

Notation	Bedeutung
$T_1, \ldots T_{n_T}$	Menge unabhängiger paralleler Tasks
$N_1, \ldots N_{n_N}$	Menge der Rechenknoten der Zielplattform
p_j	Anzahl der Prozessorkerne des Rechenknotens N_j
f_j	Leistungsfaktor des Rechenknotens N_j
F	Gesamtleistungsfaktor der heterogenen Plattform
$f(n)$	Gesamtkostenschätzung für einen Schedule S_n
$g(n)$	Kosten (Makespan) des partiellen Schedules S_n
$h(n)$	Restkostenschätzung der verbleibenden Tasks eines Schedules S_n
$d(x, y)$	Kantengewicht zwischen zwei Schedules S_x und S_y
U_n	Menge der verbleibenden Tasks im Schedule S_n
R_n	verbleibende Rechenzeit im Schedule S_n
$t_{i,r}(1)$	sequentielle Laufzeit eines Tasks T_i auf dem Referenzrechenknoten N_r
$P(S_n)$	verfügbare Prozessorzeit im Schedule S_n
$M(S_n)$	Makespan des Schedules S_n
L_{open}	erzeugte und noch unbearbeitete Schedules
L_{closed}	bearbeitete Schedules
S_{cur}	aktueller Schedule
S_{init}	leerer Schedule

Tabelle 4.1: Verwendete Notation zur Beschreibung des Schedulingproblems (oben) und des Verfahrens HP* (unten)

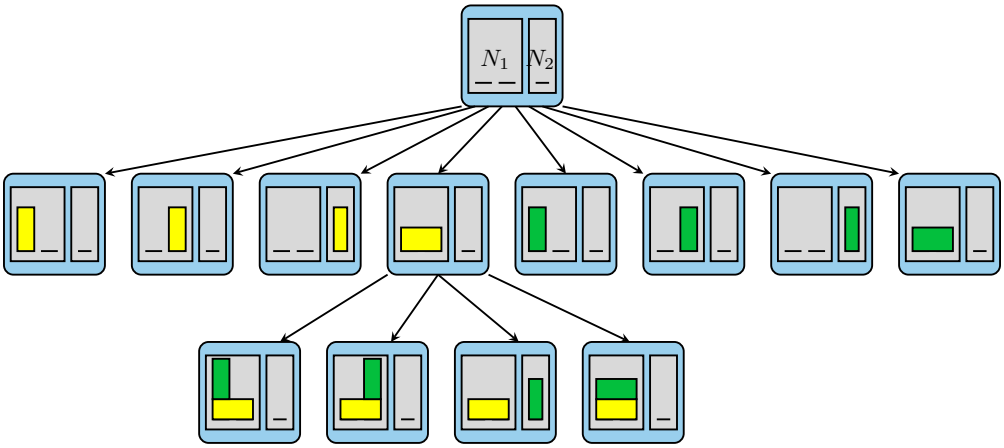


Abbildung 4.4: Teilgraph eines Scheduling Decision Tree für 2 parallele Tasks (gelb und grün) und 2 Rechenknoten N_1 und N_2 mit 2 bzw. 1 Prozessorkern.

Zur Erzeugung aller Nachfolger eines Schedules S_x wird jeder verbleibende Task jeweils allen nichtleeren Teilmengen der Prozessorkerne eines jeden Rechenknoten zugewiesen. Jede Zuweisung erzeugt einen neuen Schedule S_y und die entsprechende Kante mit dem Gewicht $d(x, y)$.

Beispiel für die Konstruktion eines Scheduling Decision Tree

In Abbildung 4.4 ist ein Teilgraph eines Scheduling Decision Tree für 2 parallele Tasks und 2 Rechenknoten mit 2 bzw. 1 Prozessorkern zu sehen. Die Wurzel des Baums ist der leere Schedule, in dem noch kein Task zugewiesen wurde. Davon ausgehend werden alle nachfolgenden Schedules hinzugefügt, indem jeweils jeder verbleibende Task (gelb bzw. grün) jeder nichtleeren Teilmenge der Prozessorkerne aller Rechenknoten im Schedule zugewiesen wird. Für einen Knoten ist in der Abbildung eine weitere derartige Expansion dargestellt, die die dritte Ebene des SDT erzeugt. Im ausgewählten Schedule wurde ein Task (gelb) 2 Prozessorkernen zugewiesen. Danach werden für den verbleibenden Task (grün) alle Nachfolger erzeugt. In den Blättern der dritten Ebene des Baumes befinden sich in diesem Beispiel mögliche Lösungen, das heißt vollständige Schedules. Um den kompletten SDT zu erhalten, ist eine Expansion aller partieller Schedules der mittleren Ebene des SDT notwendig.

Im folgenden Abschnitt wird das Schedulingverfahren HP* (Heterogeneous Parallel task scheduling based on A*) für das Scheduling von parallelen Tasks auf heterogene Rechenressourcen vorgestellt. Dieses Verfahren konstruiert den beschriebenen SDT schrittweise und stoppt sobald ein vollständiger Schedule gefunden wurde. Das Ergebnis ist der kürzeste Weg von der Wurzel zu einem Blattknoten

und entspricht der inkrementellen Konstruktion eines Schedules. Um eine optimale Lösung zu finden, muss HP* den SDT nicht zwangsläufig komplett konstruieren.

4.4.2 Ablauf des Schedulings mit HP*

Während des Schedulings erzeugt HP* sukzessive den Scheduling Decision Tree, bis ein vollständiger Schedule gefunden wird. Begonnen wird mit einem leeren Schedule. In jedem Durchlauf werden ausgehend vom aktuellen Schedule alle nachfolgenden Schedules generiert. Als neuer aktueller Schedule wird anschließend der Schedule mit der geringsten Gesamtkostenschätzung aller bisher erzeugten Schedules gewählt und mit diesem fortgefahren. Im Folgenden werden die Berechnung der Gesamtkostenschätzung und der Algorithmus des Schedulingverfahrens HP* im Detail vorgestellt.

Berechnung der Gesamtkostenschätzung

Analog zur A*-Suche wird auch in HP* eine Funktion $f(n) = g(n) + h(n)$ genutzt, um die Gesamtkostenschätzung eines Pfades von der Wurzel des Scheduling Decision Tree über den Schedule S_n zu einem vollständigen Schedule zu ermitteln. Die Teilpfadkosten $g(n)$ zu einem Schedule S_n entsprechen dessen Makespan und somit gilt $g(n) = M(S_n)$. Für jeden Schedule S_n mit Ausnahme der vollständigen Schedules existiert eine Menge U_n, die jeweils die verbleibenden Tasks enthält. Für die Berechnung der Restkostenschätzung $h(n)$ wird angenommen, dass sich diese verbleibenden Tasks U_n optimal über alle Prozessorkerne verteilen lassen. Diese Tasks werden dazu als ein zusammenhängender malleable task aufgefasst, der übergreifend die Prozessorkerne aller Rechenknoten nutzt, sobald diese verfügbar sind. Die sequentielle Laufzeit dieses malleable tasks entspricht der verbleibenden Rechenzeit (Definition 2.1 auf Seite 22) und ergibt sich aus der Summe der sequentiellen Laufzeiten aller verbleibender Tasks. Die parallele Laufzeit entspricht der sequentiellen Laufzeit geteilt durch die Anzahl verwendeter Prozessorkerne.

Die Berechnung der Restkostenschätzung $h(n)$ für einen Schedule S_n unterscheidet zwei Fälle. Falls die verfügbare Prozessorzeit $P(S_n)$ (Definition 2.2 auf Seite 22) größer ist als die verbleibende Rechenzeit R_n, dann gilt $h(n) = 0$. In diesem Fall wird durch die Restkostenschätzung angenommen, dass die verbleibenden Tasks nicht zu einer Erhöhung des Makespans $M(S_n)$ führen werden. Im anderen Fall reicht die verfügbare Prozessorzeit $P(S_n)$ nicht für die verbleibende Rechenzeit R_n aus. Die Restkostenschätzung nimmt dann an, dass sich die überschüssige Rechenzeit gleichmäßig auf dem gesamten heterogenen parallelen System verteilen lässt. Für die Berechnung wird also die Differenz aus verbleibender Rechenzeit und verfügbarer

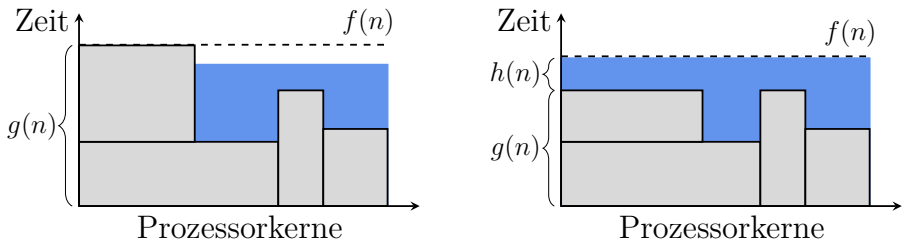


Abbildung 4.5: Illustration der Berechnung der Funktion $f(n)$ für Schedules mit bereits zugewiesenen Tasks (grau) und verbleibender Rechenzeit (blau), die kleiner (links) oder größer (rechts) als die verfügbare Prozessorzeit ist.

Prozessorzeit durch den Gesamtleistungsfaktor F geteilt. Somit ergeben sich für die Berechnung der Restkostenschätzung $h(n)$ die folgenden zwei Fälle:

$$h(n) = \begin{cases} \frac{R_n - P(S_n)}{F}, & \text{falls } R_n > P(S_n) \\ 0, & \text{andernfalls} \end{cases} \tag{4.4}$$

Abbildung 4.5 zeigt eine Illustration der beiden Fälle der Berechnung der erwarteten Gesamtkosten. Darin sind die bereits zugewiesenen Tasks grau und die verbleibende Rechenzeit blau dargestellt. Im linken Schedule ist die verfügbare Prozessorzeit größer als die verbleibende Rechenzeit und somit gilt $f(n) = g(n)$. Im rechten Schedule reicht die verfügbare Prozessorzeit nicht für die verbleibende Rechenzeit aus und somit gilt $f(n) = g(n) + h(n)$.

Algorithmus

In Algorithmus 2 ist der Ablauf des Schedulingverfahrens HP* dargestellt. Im Verlauf von HP* wird der Scheduling Decision Tree (Definition 4.3 auf Seite 46) schrittweise soweit wie nötig aufgebaut. Dazu werden alle Nachfolger des jeweils aktuell besuchten Schedules erzeugt. Schedules, die bereits erzeugt, aber noch nicht besucht wurden, werden im Folgenden als entdeckte Schedules bezeichnet. Zu Beginn wird mit S_{init} ein leerer Schedule erzeugt, der zu diesem Zeitpunkt der einzige entdeckte Schedule ist.

In jedem Durchlauf der Hauptschleife (Zeilen 3–14) wird der Schedule S_{cur} besucht, der die geringste Gesamtkostenschätzung $f(S_{cur})$ aller Schedules aufweist, die entdeckt, aber noch nicht besucht wurden. Falls es sich bei S_{cur} um einen vollständigen Schedule handelt, wurde die Lösung gefunden und der Algorithmus terminiert. Ist dies nicht der Fall, so werden, wie in Abschnitt 4.4.1 beschrieben, alle Nachfolger von S_{cur} im Scheduling Decision Tree erzeugt und den entdeckten Schedules zugeordnet. Bei der A*-Suche werden auch bereits besuchte Nachfolger eines Knotens

Algorithmus 2: Das Schedulingverfahren HP*.

Eingabe: • unabhängige parallele Tasks $T_1, \ldots, T_{n_T}$
• Rechenknoten $N_1, \ldots, N_{n_N}$
Ausgabe: • vollständiger Schedule

```
1  Initialisierung von S_init als leerer Schedule
2  S_init ist der einzige entdeckte Schedule
3  while (es sind entdeckte Schedules vorhanden) do
4  |  Besuch des entdeckten Schedules S_cur mit der geringsten Gesamtkostenschätzung
5  |  Zuordnung von S_cur zu den besuchten Schedules
6  |  if (S_cur ist ein vollständiger Schedule) then
7  |  |  Terminierung des Algorithmus mit Ausgabe der gefundenen Lösung S_cur
8  |  end
9  |  else
10 |  |  foreach (Nachfolger S von S_cur im Scheduling Decision Tree) do
11 |  |  |  Zuordnung von S zu den entdeckten Schedules
12 |  |  end
13 |  end
14 end
```

berücksichtigt, wenn deren Gesamtkostenschätzung verglichen mit dem vorigen Besuch geringer ausfällt. In HP* kann dieser Fall hingegen nicht eintreten, da es sich beim Scheduling Decision Tree um einen Graph mit Baumstruktur handelt.

4.4.3 Optimalität von HP*

In [14] wurde gezeigt, dass der A*-Suchalgorithmus eine optimale Lösung findet, wenn die Gesamtkostenschätzung zulässig und konsistent ist. Auf dieser Grundlage kann gezeigt werden, dass auch HP* eine optimale Lösung findet. Dazu wird zunächst gezeigt, dass die von HP* verwendete Restkostenschätzung zulässig ist und dass die verwendete Gesamtkostenschätzung konsistent ist.

Lemma 4.4. *Die von HP* verwendete Restkostenschätzung $h(n)$ (Gleichung (4.4)) ist zulässig und die verwendete Gesamtkostenschätzung $f(n)$ ist konsistent.*

Beweis. Im Scheduling Decision Tree entspricht der Nachfolger eines Schedules S_n einem Schedule S_m, in dem genau ein Task mehr zugewiesen wurde als im Schedule S_n. Dieser Task wurde p Prozessorkernen des Rechenknotens N_j zugewiesen, wobei $1 \leq p \leq p_j$ und $j \in \{1, \ldots, n_N\}$ gilt. Die Restkostenschätzung $h(n)$ gibt eine Schätzung für den Anstieg des Makespans $M(S_n)$ des Schedules S_n an, die auf den verbleibenden Tasks basiert. Die exakten Kosten dieses Anstiegs werden mit $h^*(n)$

bezeichnet. Es kann für jeden Schedule S_n gezeigt werden, dass folgende Aussage gilt und somit die Zulässigkeit gegeben ist:

$$h(n) \leq h^*(n)$$

Zur Berechnung von $h(n)$ wird Gleichung (4.4) (Seite 50) verwendet. Dabei wird ausschließlich der Fall betrachtet, dass für einen Schedule S_n die verbleibende Rechenzeit R_n größer als die verfügbare Prozessorzeit $P(S_n)$ ist. Im anderen Fall ist die Restkostenschätzung mit $h(n) = 0$ immer kleiner gleich den exakten Kosten $h^*(n)$, da es sich bei allen Kosten um nicht-negative Werte handelt. Während die Berechnung der Restkostenschätzung die sequentielle Laufzeit $t_{i,r}(1)$ verwendet, beruhen die exakten Kosten $h^*(n)$ auf der parallelen Laufzeit $t_{i,j}(p)$ des neu zugewiesenen Tasks T_i. Daher wird zur Formulierung auf der rechten Seite der Ungleichung anstelle der verbleibenden (sequentiellen) Rechenzeit R_n die tatsächliche Rechenzeit verwendet. Für die tatsächliche Rechenzeit werden p Prozessorkerne mit einem Leistungsfaktor f_j für die Laufzeit $t_{i,j}(p)$ benötigt. Das Einsetzen der Gleichung (4.4) liefert somit:

$$\frac{R_n - P(S_n)}{F} \leq \frac{t_{i,j}(p) \cdot p \cdot f_j - P(S_n)}{p \cdot f_j}$$

Nach Anwendung der Definition 2.1 (Seite 22) entspricht die verbleibende Rechenzeit R_n der Laufzeit $t_{i,r}(1)$ auf einem Kern des Referenzrechenknotens N_r. Nach Gleichung (3.1) (Seite 26) kann die Laufzeit $t_{i,j}(p)$ auf dem Rechenknoten N_j auch als Laufzeit $t_{i,r}(p)/f_j$ unter Verwendung des Referenzrechenknotens formuliert werden. Das Aufschlüsseln der verbleibenden Rechenzeit liefert:

$$\frac{t_{i,r}(1) - P(S_n)}{F} \leq \frac{t_{i,j}(p) \cdot p \cdot f_j - P(S_n)}{p \cdot f_j} = \frac{\frac{t_{i,r}(p)}{f_j} \cdot p \cdot f_j - P(S_n)}{p \cdot f_j}$$

Der Gesamtleistungsfaktor F (Gleichung 2.2 (Seite 16)) ist immer größer oder gleich eines der Summanden $p \cdot f_j$, da es sich um positive Zahlen handelt. Im Allgemeinen kann für parallele Tasks angenommen werden, dass $t_{i,r}(1)$ immer kleiner oder gleich $t_{i,r}(p) \cdot p$ ist. Daraus folgt, dass $h(n) \leq h^*(n)$ für jeden Schedule S_n gilt und somit laut Definition 4.1 (Seite 45) $h(n)$ eine zulässige Restkostenschätzung ist.

Die Gesamtkostenschätzung $f(n)$ eines Schedules S_n setzt sich aus dessen Makespan und dessen Restkostenschätzung zusammen, die beide jeweils als nicht-negative Werte definiert sind. Da die Restkostenschätzung zulässig und nicht-negativ ist, folgt daraus, dass $f(n)$ entlang des Wegs von einem leeren Schedule zu einem vollständigen Schedule niemals abnimmt. Damit ist nach Definition 4.2 (Seite 45) die Gesamtkostenschätzung $f(n)$ konsistent. □

Lemma 4.5. *HP* findet eine optimale Lösung, falls diese existiert, wenn die Restkostenschätzung $h(n)$ (Gleichung (4.4)) verwendet wird.*

Beweis. Wie in [14] gezeigt wurde, findet der A*-Suchalgorithmus garantiert einen kürzesten Weg in einem gerichteten Graphen, wenn eine zulässige und konsistente Kostenfunktion verwendet wird. Da die von HP* verwendete Gesamtkostenschätzung zulässig und konsistent ist, findet HP* garantiert einen kürzesten Weg von der Wurzel zu einem Blattknoten im Scheduling Decision Tree (SDT). Nach der Definition des SDT entspricht ein Blatt einem vollständigen Schedule. Begründen lässt sich die Optimalität dieses Schedules mit der Vorgehensweise des Algorithmus. So wählt HP* in jedem Schritt den Schedule mit der geringsten Gesamtkostenschätzung aller bis dahin entdeckten Schedules aus. Der erste vollständige Schedule, der entdeckt wird, wird als Lösung des Schedulingproblems zurückgeliefert. Da in jedem Schritt der Schedule mit der geringsten Gesamtkostenschätzung besucht wird und diese Gesamtkostenschätzung konsistent ist, kann kein anderer Schedule im SDT eine geringere Gesamtkostenschätzung haben. Für einen vollständigen Schedule ist die Restkostenschätzung gleich 0, da keine nicht zugewiesenen Tasks mehr verbleiben. Die Gesamtkostenschätzung eines vollständigen Schedules besteht somit nur noch aus den Teilpfadkosten, die dem Makespan entsprechen. Daraus folgt, dass HP* garantiert den Schedule mit dem geringsten Makespan und somit eine optimale Lösung findet. □

4.4.4 Eingrenzen des Suchraums

Der von HP* erzeugte Scheduling Decision Tree (SDT) kann Schedules enthalten, die für das Finden einer optimalen Lösung nicht benötigt werden. Diese Schedules können von der Suche ausgeschlossen werden ohne die Eigenschaft der Optimalität zu zerstören. Ein solches Eingrenzen des Suchraums kann den Speicherverbrauch von HP* deutlich senken. Ein weiterer Vorteil ist, dass weniger Schedules betrachtet werden müssen, was die Effizienz des Verfahrens verbessert. Zur Verdeutlichung ist in Abbildung 4.6 ein Teilgraph eines SDT für das Scheduling von zwei parallelen Tasks (gelb und grün) auf zwei Rechenknoten N_1 mit 2 Prozessorkernen und N_2 mit 1 Prozessorkern zu sehen. Kandidaten für den Ausschluss aus der Suche sind im Graphen farblich markiert.

In [45] und [70] wurden bereits Methoden für das Eingrenzen des Suchraums für das Scheduling voneinander abhängiger sequentieller Tasks vorgestellt. Während einige dieser Methoden speziell für Tasks mit Abhängigkeiten entworfen wurden, können andere auch auf das Scheduling unabhängiger paralleler Tasks übertragen werden. Die folgenden Methoden für das Eingrenzen des Suchraums entstammen den genannten Arbeiten und wurden für das Scheduling unabhängiger paralleler Tasks angepasst:

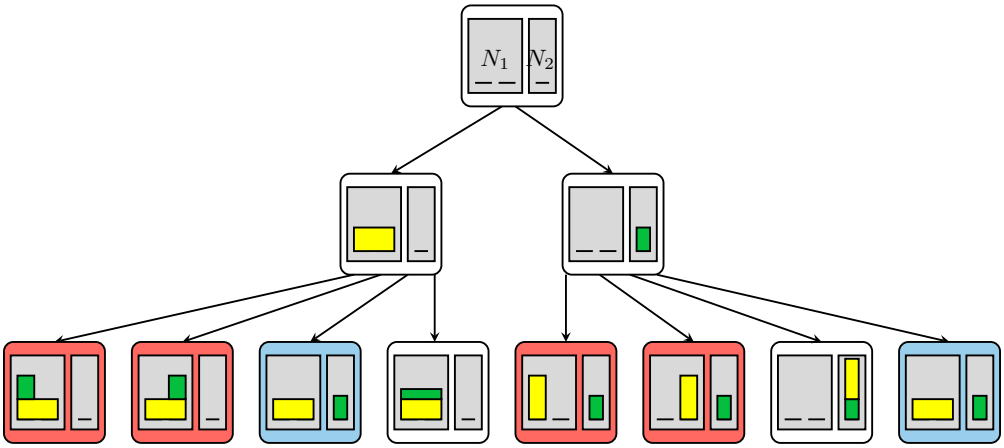


Abbildung 4.6: Teilgraph eines Scheduling Decision Tree für das Scheduling von zwei parallelen Tasks (gelb und grün) auf zwei Rechenknoten N_1 mit 2 Prozessorkernen und N_2 mit 1 Prozessorkern. Identische Schedules im Graphen sind blau und äquivalente Schedules rot markiert.

Ausschluss identischer Schedules

Da HP* alle möglichen Reihenfolgen und Zuordnungen der Tasks zu den Prozessorkernen betrachtet, können im Scheduling Decision Tree identische Schedules auftreten.

Definition 4.6 (Identische Schedules)**.** Zwei Schedules sind identisch, wenn denselben Prozessorkernen jeweils dieselben Tasks in derselben Reihenfolge zugewiesen wurden.

Abbildung 4.6 zeigt wie unterschiedliche Pfade im Scheduling Decision Tree zu identischen Schedules führen können. Im konkreten Beispiel weist HP* zuerst den gelben Task dem Rechenknoten N_1 und anschließend den grünen Task dem Rechenknoten N_2 zu. Im Algorithmus wird zusätzlich auch der umgekehrte Fall betrachtet, in dem zuerst der grüne Task dem Knoten N_2 und danach der gelbe Task dem Knoten N_1 zugewiesen wird. Beide Fälle führen zum identischen Schedule (blau markiert).

Der Ausschluss identischer Schedules in HP* erfolgt beim Besuch des Schedules S_{cur} (Zeile 4 in Algorithmus 2). Die Bearbeitung des Schedules S_{cur} (Zeilen 5–13) wird übersprungen, wenn bereits ein identischer Schedule besucht wurde. Auf diese Weise werden identische Schedules zwar von HP* erzeugt und den entdeckten Schedules zugeordnet, aber höchstens einmal bearbeitet.

Ausschluss äquivalenter Schedules

Neben identischen Schedules können im Scheduling Decision Tree (SDT) auch äquivalente Schedules auftreten. Da die Prozessorkerne eines Rechenknotens homogen

sind, unterscheiden sich diese nur im Namen. Somit unterscheiden sich auch die jeweiligen Tasklaufzeiten für die gleiche Anzahl an Kernen innerhalb eines Rechenknotens nicht, unabhängig von der konkreten Zuweisung. Das bedeutet, dass innerhalb eines Schedules jede Permutation der Prozessorkerne eines Rechenknotens einem äquivalenten Schedule entspricht.

Definition 4.7 (Äquivalente Schedules)**.** Zwei Schedules werden als äquivalent bezeichnet, wenn sie unabhängig von der Permutation der Prozessorkerne eines Rechenknotens dieselben Schedules repräsentieren. Präziser ausgedrückt sind zwei Schedules des SDT äquivalent, wenn die normalisierten Schedules identisch sind. Ein Schedule kann normalisiert werden, indem die Prozessorkerne jeweils eines Rechenknotens nach der maximalen Endezeit aller ihnen zugewiesenen Tasks sortiert werden.

Der SDT in Abbildung 4.6 enthält jeweils 2 äquivalente Schedules (rot markiert) in jedem Teilbaum. In HP* können äquivalente Schedules genau dann auftreten, wenn für jede Kombination von Prozessorkernen eines Rechenknotens ein neuer Schedule im SDT angelegt wird. Dies geschieht während der Erzeugung aller Nachfolger S des aktuellen Schedules S_{cur} (vgl. Zeile 10). An dieser Stelle des Algorithmus können äquivalente Schedules erkannt werden, indem alle Nachfolger S normalisiert und miteinander verglichen werden. Treten dabei identische Schedules auf, wird nur genau einer dieser Schedules den entdeckten Schedules hinzugefügt und die übrigen werden verworfen.

Verwendung einer heuristischen oberen Schranke

Für das Scheduling von parallelen Tasks auf heterogene Rechenressourcen existieren zahlreiche heuristische Schedulingverfahren. Einige dieser Verfahren werden in Abschnitt 2.3 vorgestellt. Diese Verfahren ermitteln einen Schedule, der zwar nicht zwangsläufig optimal ist, aber dessen Makespan M_{heur} dennoch möglichst gering ist. Da HP* immer eine optimale Lösung findet, befindet sich in jedem Fall ein optimaler Schedule im Scheduling Decision Tree, dessen Makespan M_{opt} kleiner gleich M_{heur} ist. Schedules mit höheren Kosten können vom Algorithmus nie ausgewählt werden, da in jedem Schritt der Schedule mit den geringsten Kosten gewählt wird. Somit können alle Schedules S_n von der Suche ausgeschlossen werden, deren Kosten $f(n)$ größer als M_{heur} sind. Auf diese Weise kann die Anzahl der Schedules im Scheduling Decision Tree deutlich reduziert werden ohne die Eigenschaft der Optimalität zu zerstören. Diese Methode reduziert zwar nicht die Anzahl von HP* betrachteter Schedules, verringert jedoch den Hauptspeicherbedarf enorm. Die Effizienz der Methode hängt allerdings von der Qualität der Lösung ab, die von der Heuristik gefunden wird.

Korollar 4.8. *Der Ausschluss identischer und äquivalenter Schedules sowie die Verwendung einer heuristischen oberen Schranke haben keinen Einfluss auf die Optimalität von HP*.*

Beweis. Identische Schedules gleichen sich sowohl in Bezug auf die zugewiesenen als auch in Bezug auf die verbleibenden Tasks. Im weiteren Verlauf weist HP* ausgehend von den identischen Schedules dieselben verbleibenden Tasks in allen möglichen Kombinationen den Prozessorkernen zu. Somit werden für identische Schedules auch identische nachfolgende Schedules erzeugt. Das führt zu den identischen Unterbäumen im Scheduling Decision Tree (SDT), so dass mögliche Lösungen ebenfalls mehrfach vorhanden sind. HP* überspringt einen Schedule nur dann, wenn ein identischer Schedule bereits bearbeitet wurde. Somit verbleibt einer der identischen Unterbäume mit all seinen möglichen Lösungen stets im SDT. Der Ausschluss identischer Schedules reduziert demnach den Suchraum, ohne optimale Lösungen zu eliminieren.

Äquivalente Schedules sind mit Ausnahme der Benennung der Prozessorkerne eines Rechenknotens identisch. Da es sich bei den Rechenknoten selbst um homogene parallele Systeme handelt, hat die Benennung der Prozessorkerne keinen Einfluss auf den Makespan. Ausgehend von äquivalenten Schedules weist HP* dieselben verbleibenden Tasks in allen möglichen Kombinationen den Prozessorkernen zu. Auf diese Weise entstehen äquivalente Unterbäume im SDT, die äquivalente Schedules mit dem gleichen Makespan enthalten. Da HP* einen der äquivalenten Schedules behält, wird keine mögliche Lösung verworfen. Auch ohne Verlust optimaler Lösungen können demnach äquivalente Schedules von der Suche ausgeschlossen werden.

Die Verwendung einer heuristischen oberen Schranke b in HP* führt zum Ausschluss eines Schedules S_n, wenn dessen erwartete Gesamtkosten größer als b sind. Die Restkostenschätzung $h(n)$ (Gleichung (4.4)) unterschätzt die exakten Kosten $h^*(n)$ für jeden Schedule S_n. Infolgedessen sind die erwarteten Gesamtkosten $f(n)$ für jeden Schedule S_n geringer als die exakten Kosten $f^*(n)$. Eine Lösung S_{opt} ist optimal, wenn keine andere Lösung mit geringeren Gesamtkosten existiert und somit $f^*(S_{opt}) \leq f^*(S_n)$ für alle Schedules S_n gilt. Die heuristische obere Schranke b muss also immer größer oder gleich $f^*(S_{opt})$ sein. Daher führt die Verwendung einer heuristischen oberen Schranke nicht zum Ausschluss optimaler Lösungen. □

4.4.5 Analyse der Komplexität

Das Scheduling paralleler Tasks ist ein NP-schweres Problem. Solange $P \neq NP$ gilt, kann daher eine optimale Lösung für das Scheduling paralleler Tasks auf heterogene parallele Systeme nicht in polynomieller Zeit gefunden werden. Auch HP* sucht nach einer optimalen Lösung und es kann gezeigt werden, dass die Laufzeit von HP* exponentiell mit der Taskanzahl steigt.

Lemma 4.9. *Die worst-case-Laufzeit von HP* verhält sich exponentiell zur Taskanzahl.*

Beweis. Um garantiert eine optimale Lösung zu finden, kann ein Durchsuchen des gesamten Scheduling Decision Tree (SDT) notwendig sein. Die worst-case-Laufzeit von HP* hängt demnach von der Größe des SDT ab. Wie in Abschnitt 4.4.1 beschrieben, hat der SDT eine Baumstruktur, dessen Wurzel ein leerer Schedule ist. Ein Knoten des Baums repräsentiert einen partiellen Schedule. Ausgehend von der Wurzel wird in jeder Ebene des Baums ein Task mehr im Vergleich zum jeweiligen Elternknoten im Schedule zugewiesen. Demnach entspricht die Tiefe eines Knotens der Anzahl zugewiesener Tasks und die maximale Tiefe des Baums entspricht n_T, das heißt der Gesamtanzahl an Tasks. Jeder innere Knoten des SDT hat einen Nachfolger für jede mögliche Zuweisung jedes noch nicht zugewiesenen Tasks. Diese Nachfolger werden gebildet, indem jeder dieser Tasks $p = 1, \ldots, p_j$ Prozessorkernen eines jeden Rechenknotens $N_j, j = 1, \ldots, n_N$ zugewiesen wird. Dabei wird für jede Anzahl p ein Nachfolger für jede Kombination dieser p Prozessorkerne eines jeden Rechenknotens N_j hinzugefügt. Daher entspricht die Anzahl der Nachfolger eines Knotens S_n im SDT der Summe aller Kombinationen (Binomialkoeffizienten) von 1 bis p_j Prozessorkernen über alle Rechenknoten $N_1, \ldots, N_{n_N}$ multipliziert mit r und kann wie folgt berechnet werden:

$$\#Nachfolger(S_n) = r \cdot \sum_{j=1}^{n_N} \sum_{p=1}^{p_j} \binom{p_j}{p} \tag{4.5}$$

Dabei bezeichnet S_n einen partiellen Schedule, r die Anzahl noch nicht zugewiesener Tasks, n_N die Anzahl vorhandener Rechenknoten und p_j die Anzahl verfügbarer Prozessorkerne auf dem Rechenknoten N_j. Die Anzahl noch nicht zugewiesener Tasks r entspricht für die Wurzel der Gesamtanzahl an Tasks n_T. Mit der Tiefe des Baums nimmt r jeweils genau um 1 ab, da in jeder Ebene genau ein Task mehr zugewiesen wird. Zur Erzeugung des kompletten SDT werden somit insgesamt $n_T \cdot (n_T - 1) \cdot \ldots \cdot 1 = n_T!$ Tasks zugewiesen. Die Anzahl der Nachfolger eines Knotens im SDT hängt zudem von der Anzahl Rechenknoten n_N sowie deren Anzahl Prozessorkerne p_j ab. Die Anzahl der Knoten und somit (partieller) Schedules im SDT steigt exponentiell mit Tiefe des Baums und kann daher wie folgt berechnet werden:

$$\#Schedules = n_T! \cdot \left(\sum_{j=1}^{n_N} \sum_{p=1}^{p_j} \binom{p_j}{p} \right)^{n_T} \tag{4.6}$$

Die Anzahl der Knoten im Scheduling Decision Tree und somit auch die Laufzeit von HP* verhält sich exponentiell zur Anzahl paralleler Tasks. □

Aus Gleichung (4.6) geht zusätzlich hervor, dass die Heterogenität des parallelen Systems einen entscheidenden Einfluss auf die Laufzeit von HP* hat. So fällt der

Rechen-knoten	Prozessor	Architektur	Erscheinungs-jahr	Anzahl Kerne	RAM-Größe	Frequenz in GHz
`skylake1`	Intel i7-6700	Skylake	2015	4	16 GB	3.40
`hw1`	Intel i7-4770K	Haswell	2013	4	16 GB	3.50
`sb1`	Intel Xeon E5-2650	Sandy Bridge	2012	8	32 GB	2.00

Tabelle 4.2: Liste der Rechenknoten des verwendeten heterogenen Rechenclusters.

Scheduling Decision Tree für ein homogenes paralleles System größer aus als für ein heterogenes paralleles System mit der gleichen Anzahl verfügbarer Prozessorkerne. Beispielsweise könnte ein heterogenes System aus 4 Prozessoren mit jeweils 8 Prozessorkernen bestehen, wohingegen ein entsprechendes homogenes System über 1 Prozessor mit 32 Kernen verfügen würde. Da ein paralleler Task nur auf genau einem Rechenknoten ausgeführt werden kann, werden von HP* auch nur Zuweisungen zu allen Kombination von Prozessorkernen eines jeden Rechenknotens betrachtet. Für einen großen Rechenknoten müssen demnach mehr Kombinationen betrachtet werden als für mehrere kleine Rechenknoten mit der gleichen Gesamtanzahl verfügbarer Prozessorkerne. Im genannten Beispiel würden pro Task für das heterogene System $4 \cdot 2^8$ Schedules und für das homogene System 2^{32} Schedules erzeugt. Als Konsequenz ergibt sich für heterogene Systeme eine deutlich kürzere Laufzeit des Verfahrens HP* als für vergleichbare homogene Systeme.

4.5 Experimentelle Auswertung

In diesem Abschnitt wird die Laufzeit der erzeugten Schedules des Schedulingverfahrens HP* evaluiert. Zunächst wird der Einfluss der in Abschnitt 4.4.4 vorgestellten Strategien zur Eingrenzung des Suchraums auf die Laufzeit des Verfahrens HP* untersucht. Anschließend wird HP* anhand von Laufzeitmessungen mit den in Abschnitt 2.3.1 vorgestellten List-Scheduling-Verfahren HCPA und Δ-CTS verglichen.

4.5.1 Zielplattform und betrachtete Anwendungen

Der verwendete heterogene Rechencluster besteht aus 3 Rechenknoten mit insgesamt 16 Prozessorkernen. In Tabelle 4.2 sind die Eigenschaften dieser Rechenknoten aufgelistet. Für die Bestimmung der Parameter des Kostenmodells (vgl. Anschnitt 3.2.1) wird der Rechenknoten `sb1` als Referenzrechenknoten verwendet, da dieser über die größte Anzahl Prozessorkerne verfügt. Die Schedulingverfahren werden auf einem separaten Front-End-Knoten ausgeführt, der mit einem Intel Xeon E5-2683-Prozessor und 128 GB Arbeitsspeicher ausgestattet ist. Dieser Rechenknoten startet

die Ausführung der Tasks gemäß der ermittelten Schedules über SSH-Verbindungen zu den Rechenknoten. Jeder Task wird gestartet, sobald alle seine Vorgänger im Schedule ihre Ausführung beendet haben. Während der Ausführung eines Schedules wird die Gesamtlaufzeit vom Start des ersten Tasks bis zur Beendigung des letzten Tasks gemessen. Um eventuelle Schwankungen der Laufzeit zu berücksichtigen, wird jeder Schedule fünfmal ausgeführt und die Durchschnittszeit angegeben. Als parallele Tasks werden die zwei Anwendungen BARNES und FMM sowie die zwei Kernel CHOLESKY und LU der SPLASH-3-Benchmark-Suite verwendet. Sofern nicht anders angegeben, werden die Standardparameter oder der Parametersatz „parsec-simlarge“ für die Benchmark-Tasks verwendet.

4.5.2 Analyse der Strategien zur Eingrenzung des Suchraums

Im Folgenden soll der Einfluss der in Abschnitt 4.4.4 vorgestellten Strategien zur Eingrenzung des Suchraums auf die Laufzeit des Verfahrens HP* untersucht werden. Da die Laufzeit von HP* stark von der Hardware des ausführenden Systems sowie der Implementierung abhängt, wird dazu jeweils die Anzahl der erzeugten und der vom Algorithmus besuchten Schedules ausgewertet. Zu den erzeugten Schedules zählen alle partiellen Schedules, die zur Durchsuchung der Nachbarschaft in Zeile 11 des Algorithmus 2 generiert und zu den entdeckten Schedules hinzugefügt werden. Ein partieller Schedule wird als besucht bezeichnet, wenn er im Verlauf des Algorithmus aufgrund seiner geringen Gesamtkostenschäzung ausgewählt wird (vgl. Zeile 4 in Algorithmus 2). Die Anzahl der Schedules ist unabhängig von der Hardware und der Implementation und verhält sich annähernd proportional zur Laufzeit von HP*. In dem für die Analyse betrachteten Schedulingproblem sollen 1 bis 16 BARNES-Anwendungstasks dem in Tabelle 4.2 beschriebenen heterogenen Cluster zugewiesen werden.

Tabelle 4.3 zeigt die Anzahl Schedules, die in Abhängigkeit von der Taskanzahl von HP* unter der Verwendung einzelner Strategien zur Eingrenzung des Suchraums jeweils erzeugt und besucht werden. Jede dieser Strategien wurde separat betrachtet und die entsprechenden Werte in der Tabelle aufgelistet. Sowohl die Anzahl der erzeugten als auch die Anzahl der besuchten Schedules beeinflussen zum einen die Laufzeit von HP* und zum anderen den Speicherbedarf. Da alle Schedules während der Ausführung von HP* gespeichert werden müssen und deren Anzahl exponentiell mit der Taskanzahl steigt, wird bereits für kleinere Schedulingprobleme eine große Menge an Speicher benötigt. Die Anwendung auf größere Schedulingprobleme führt daher häufig zu einem Abbruch des Schedulings, da die Speicherkapazität des ausführenden Systems überschritten wird. Die fehlenden Einträge in der Tabelle sind jeweils das Ergebnis eines solchen Abbruchs. Die Ergebnisse zeigen, dass HP* ohne Einschränkung des Suchraums nur auf kleine Schedulingprobleme anwendbar ist.

	keine Eingrenzung		ohne identische		ohne äquivalente		heuristische Schranke	
#Tasks	besucht	erzeugt	besucht	erzeugt	besucht	erzeugt	besucht	erzeugt
1	1	285	1	285	1	16	1	1
2	2	570	2	570	2	32	2	3
3	17	4.845	12	3.420	8	131	17	33
4	15.965	4.550.025	3.385	964.725	146	2.660	15.965	22.138
5	69.137	19.704.045	7.333	2.089.905	322	5.921	69.137	84.746
6	236.157	67.304.745	12.871	3.668.235	550	10.087	236.157	305.762
7	——	——	——	——	23.089	489.373	81.178.658	104.911.723
8	——	——	——	——	81.229	1.589.633	——	——
9	——	——	——	——	90.481	1.819.179	——	——
10	——	——	——	——	756.512	16.604.150	——	——
11	——	——	——	——	2.560.273	60.473.639	——	——
12	——	——	——	——	——	——	——	——

Tabelle 4.3: Anzahl von HP* erzeugter und besuchter Schedules in Abhängigkeit von der Taskanzahl unter Verwendung einzelner Strategien zur Eingrenzung des Suchraums.

Der Ausschluss identischer Schedules reduziert die Anzahl sowohl besuchter als auch erzeugter Schedules deutlich. So müssen ohne identische Schedules bis zu 94% weniger Schedules besucht und erzeugt werden, um eine Lösung zu finden. Dennoch kommt es für mehr als 6 Tasks zu einem Abbruch des Schedulings aufgrund eines zu hohen Speicherverbrauchs. Der größte Effekt der vorgestellten Strategien wird durch den Ausschluss äquivalenter Schedules erreicht. Mit dieser Strategie kann HP* eine Lösung für bis zu 11 Tasks finden, bevor der Speicherbedarf zu groß wird. Diese Verbesserung wird dadurch erreicht, dass im Vergleich zur version ohne Eingrenzung des Suchraums bis zu 99% weniger Schedules verarbeitet werden müssen. Zur Bestimmung der heuristischen oberen Schranke wurde das Water-Level-Search-Verfahren verwendet, das in Kapitel 7 vorgestellt wird. Wie bereits in Abschnitt 4.4.4 beschrieben, hat heuristische obere Schranke keinen Einfluss auf die Anzahl von HP* betrachteter Schedules. Jedoch werden mit dieser Strategie im Vergleich zur Version ohne Eingrenzung des Suchraums bis zu 99% weniger Schedules erzeugt. Allerdings ist auch diese signifikante Reduzierung nicht ausreichend, damit HP* eine Lösung für mehr als 7 Tasks finden kann.

Einzeln eingesetzt, ist jede der vorgestellten Strategien zur Eingrenzung des Suchraums bereits in der Lage, die Anzahl von HP* besuchter und erzeugter Schedules deutlich zu reduzieren. Jedoch ist der Speicherbedarf von HP* selbst für verhältnismäßig kleine Schedulingprobleme noch zu hoch. Da die Strategien unterschiedliche Teile des Suchraums ausschließen, kann eine Kombination der Strategien den Speicherbedarf möglicherweise weiter reduzieren. Aus diesem Grund wird im Folgenden eine Version von HP* betrachtet, in die alle vorgestellten Strategien integriert wurden. Ebenfalls untersucht werden soll der Einfluss der Restkostenschätzung (Gleichung 4.4 auf Seite 50) auf die Anzahl besuchter und erzeugter Schedules. Da-

	keine Eingrenzung des Suchraums		Kombination aller Strategien		Kombination aller Strategien ohne Restkostenschätzung	
#Tasks	besucht	erzeugt	besucht	erzeugt	besucht	erzeugt
1	1	285	1	1	1	1
2	2	570	2	3	2	3
3	17	4.845	6	13	6	13
4	15.965	4.550.025	46	112	46	112
5	69.137	19.704.045	76	186	76	186
6	236.157	67.304.745	106	280	106	280
7	——	——	936	2.827	936	2.827
8	——	——	1.824	5.724	1.824	5.724
9	——	——	1.900	5.734	1.929	5.789
10	——	——	7.272	22.771	7.479	23.244
11	——	——	15.416	49.352	16.335	53.922
12	——	——	62.370	220.145	62.999	222.054
13	——	——	534.397	2.195.234	534.397	2.195.234
14	——	——	1.274.832	4.518.356	1.275.255	4.519.432
15	——	——	4.957.128	20.415.293	4.957.318	20.415.054
16	——	——	12.637.246	49.721.482	12.638.539	49.727.127

Tabelle 4.4: Anzahl von HP* erzeugter und besuchter Schedules in Abhängigkeit von der Taskanzahl unter Verwendung einer Kombination aller Strategien zur Eingrenzung des Suchraums.

her wird eine Version von HP*, die eine Restkostenschätzung verwendet, mit einer Version ohne Restkostenschätzung verglichen, das heißt, in der $h(n) = 0$ gilt.

Tabelle 4.4 zeigt zeigt die Anzahl Schedules, die in Abhängigkeit von der Taskanzahl von HP* unter der Verwendung einer Kombination aller Strategien zur Eingrenzung des Suchraums jeweils erzeugt und besucht werden. Die Ergebnisse werden in der Tabelle zusammen mit den Resultaten der Version ohne Heuristikfunktion sowie der Version ohne Eingrenzung des Suchraums dargestellt. Die Kombination aller Strategien führt dazu, dass fast ausschließlich Schedules berücksichtigt werden, die für das Finden einer optimalen Lösung relevant sind. So werden im Vergleich zur Version von HP* ohne Eingrenzung des Suchraums bis zu 99,9% weniger Schedules betrachtet und erzeugt. Der Vergleich mit den Ergebnissen aus Tabelle 4.3 zeigt, dass die Kombination aller Strategien den Suchraum stärker eingrenzt als jede einzelne Strategie. Dadurch wird deutlich, dass es keine dominante Strategie gibt, sondern jede Strategie einen anderen Bereich des Suchraums eingrenzt. Die dargestellten Werte verdeutlichen, dass die Anzahl besuchter und erzeugter Schedules trotz der eingesetzten Strategien exponentiell mit der Taskanzahl steigt. Dennoch ist HP* mithilfe der Kombination aller Strategien auf kleine und mittlere Schedulingprobleme anwendbar. Im konkreten Anwendungsbeispiel können so auch für mehr als 32 Tasks Lösungen gefunden werden. Die gemessen Laufzeiten des Verfahrens HP*

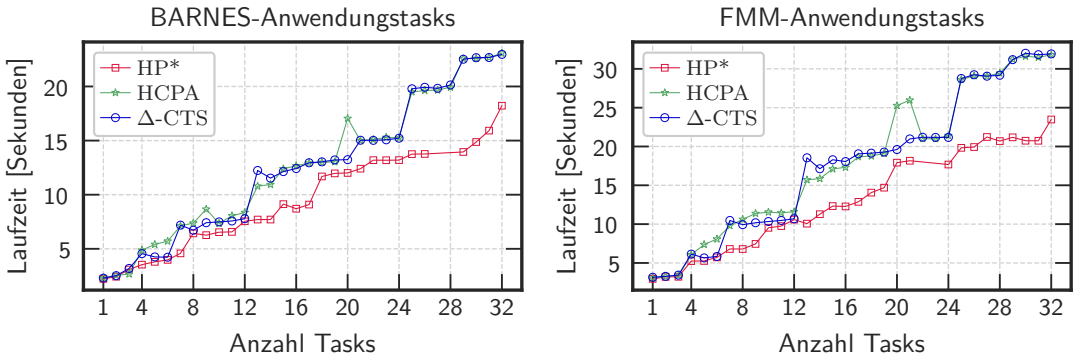


Abbildung 4.7: Gemessene Gesamtausführungszeit von BARNES- (links) und FMM-Anwendungstasks (rechts) in Abhängigkeit von der Taskanzahl auf dem in Tablle 4.2 beschriebenen heterogenen Cluster.

reichen dabei von wenigen Millisekunden bis zu 6,5 Stunden. Obwohl die Restkostenschätzung keine Strategie zur Eingrenzung des Suchraums im eigentlichen Sinne ist, beeinflusst sie die Laufzeit von HP*. Ohne eine Restkostenschätzung werden von HP* bis zu 9% mehr Schedules erzeugt und bis zu 6% mehr Schedules besucht.

4.5.3 Vergleich der Ausführungszeit der ermittelten Schedules

Im Folgenden soll die Qualität der gefundenen Lösungen des Verfahrens HP* hinsichtlich der Gesamtausführungszeit untersucht werden. Dazu werden HP* sowie die List-Scheduling-Verfahren HCPA und Δ-CTS verwendet, um jeweils 1 bis 32 SPLASH-3-Benchmark-Tasks auf dem in Tabelle 4.2 gezeigten heterogenen Cluster auszuführen. Für die Laufzeitmessungen werden in HP* alle Strategien zur Eingrenzung des Suchraums verwendet. Weitere Messergebnisse befinden sich im Anhang A.

In Abbildung 4.7 sind die gemessenen Gesamtlaufzeiten für die Ausführung der BARNES- (links) und der FMM-Anwendungstasks (rechts) in Abhängigkeit von der Taskanzahl dargestellt. Für beide Arten von Anwendungstasks liegen die Laufzeiten mit dem Verfahren HP* unter oder gleichauf mit den von HCPA und Δ-CTS erreichten Werten. Während die Ergebnisse aller Methoden für bis zu 12 Tasks nur wenig voneinander abweichen, kann HP* für größere Taskanzahlen eine signifikante Reduktion der Gesamtlaufzeit erzielen. Verglichen mit HCPA und Δ-CTS sind die Laufzeiten der von HP* ermittelten Schedules für beide Anwendungstasks bis zu 38% geringer. In vier Fällen (27 und 28 BARNES-Tasks und 22 und 23 FMM-Tasks) konnte HP* jedoch keinen Schedule bestimmen, da das Scheduling aufgrund eines zu hohen Speicherverbrauchs abgebrochen wurde. Ein Grund dafür könnte sein, dass die verwendete Restkostenschätzung (Gleichung 4.4 auf Seite 50) die tatsächlichen Kosten der verbleibenden Tasks zu stark unterschätzt. Folglich müssen von HP*

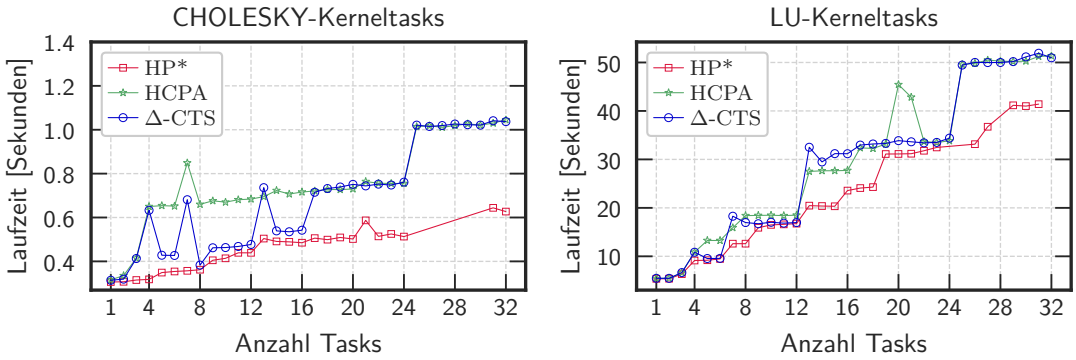


Abbildung 4.8: Gemessene Gesamtausführungszeit von CHOLESKY- (links) und LU-Kerneltasks (rechts) in Abhängigkeit von der Taskanzahl auf dem in Tabelle 4.2 beschriebenen heterogenen Cluster.

mehr partielle Schedules betrachtet werden, die als mögliche Lösungen infrage kommen.

Abbildung 4.8 zeigt die gemessenen Gesamtlaufzeiten für die Ausführung der CHOLESKY- (links) und der LU-Kerneltasks (rechts) in Abhängigkeit von der Anzahl paralleler Tasks. Auch für beide Arten von Kerneltasks sind die Laufzeiten der von HP* ermittelten Schedules geringer oder gleich den Resultaten der List-Scheduling-Verfahren HCPA und Δ-CTS. Für die CHOLESKY-Tasks erreicht HP* eine Reduktion der Gesamtlaufzeit von bis zu 58% gegenüber HCPA und von bis zu 50% gegenüber Δ-CTS. Auffällig ist auch, dass im Vergleich zu Δ-CTS die gemessene Laufzeit für 4 bis 16 CHOLESKY-Tasks bei der Verwendung von HCPA bis zu 72% höher ist. Begründen lassen sich diese großen Unterschiede damit, dass HCPA generell zu einer Ausführung der parallelen Tasks auf zu vielen Kernen tendiert. Die Ausführungszeit der einzelnen CHOLESKY-Tasks ist jedoch zu gering und profitiert daher kaum von einer höheren Anzahl von Prozessorkernen. Die zu kurze Ausführungszeit der CHOLESKY-Tasks führt ebenfalls zu einer zu starken Unterschätzung der Kosten der verbleibenden Tasks durch die Restkostenschätzung (Gleichung 4.4 auf Seite 50). Infolgedessen muss HP* mehr Schedules in die Suche einbeziehen, was zu einem erhöhten Speicherbedarf führt. Für 25 bis 30 CHOLESKY-Tasks und für 24, 25, 28 und 32 LU-Tasks führt der zu hohe Speicherverbrauch zu einem Abbruch des Schedulingverfahrens HP*. Für die LU-Tasks sind die Unterschiede zwischen den Ergebnissen der verwendeten Schedulingverfahren geringer. Dennoch führt die Verwendung von HP* zu Laufzeiten, die im Vergleich zu Δ-CTS bis zu 37% und verglichen mit HCPA bis zu 33% geringer sind.

4.6 Zusammenfassung

In diesem Kapitel wurde ein Verfahren für das Scheduling unabhängiger paralleler Tasks vorgestellt, das auf dem Ansatz des A*-Suchalgorithmus beruht. Es wurde gezeigt, dass dieses Heterogeneous Parallel task scheduling based on A* (HP*) genannte Schedulingverfahren einen optimalen Schedule findet. Des Weiteren wurde beschrieben, wie ein entsprechender Suchraum für das betrachtete Schedulingproblem definiert werden kann. Zur effizienteren Suche wurden neben einer spezifischen Restkostenschätzung drei Strategien zur Eingrenzung des Suchraums präsentiert, die keinen Einfluss auf die Optimalität von HP* haben. In Laufzeitexperimenten mit Benchmark-Tasks konnte gezeigt werden, dass der Einsatz von HP* im Vergleich zu häufig eingesetzten List-Scheduling-Verfahren zu einer deutlichen Reduktion der Ausführungszeit führt. Die Strategien zur Eingrenzung des Suchraums wurden sowohl einzeln als auch in Kombination experimentell untersucht. Dabei wurde deutlich, dass jede Strategie im Einzelnen die Anzahl der von HP* zu verarbeitenden Schedules verringert. Insbesondere jedoch die Kombination dieser Strategien führt zu einer deutlichen Reduktion der zu verarbeitenden Schedules führt. Somit ist es trotz der exponentiellen Laufzeit des Verfahrens möglich, eine optimale Lösung für kleine bis mittlere Schedulingprobleme zu ermitteln.

5 Suchbasiertes Scheduling mit der Tabu-Suche

Die Tabu-Suche ist eine Metaheuristik, die erstmals 1986 vorgestellt wurde [30]. Seitdem wurde sie bereits auf zahlreiche Optimierungsprobleme angewendet und hat sich zu einer der populärsten Lösungsmethoden entwickelt. Daher wurde im Rahmen dieser Arbeit ein auf der Tabu-Suche basierendes Verfahren für das Scheduling unabhängiger paralleler Tasks entwickelt. In diesem Kapitel wird dieses als Multiprocessor Task Tabu Search Scheduling (MTaSS) bezeichnete Verfahren vorgestellt. Zuvor wird das generelle Vorgehen einer Tabu-Suche zur Lösung von Optimierungsproblemen beschrieben. Abschließend wird in Experimenten die Leistungsfähigkeit von MTaSS hinsichtlich der Ausführungszeit der ermittelten Schedules untersucht.

5.1 Die Tabu-Suche zur Lösung von Optimierungsproblemen

Die Tabu-Suche ist ein metaheuristisches Verfahren, das zur Klasse der lokalen Suchverfahren gehört und auf Konzepten der Künstlichen Intelligenz beruht. Lokale Suchverfahren betrachten jeweils eine Lösung und deren Nachbarschaft und bewegen sich schrittweise von einer Lösung zu einer besseren Lösung im Hinblick auf eine Zielfunktion. Dieses Vorgehen kann jedoch dazu führen, dass sich die Suche in einem lokalen Optimum festsetzt. Durch die Möglichkeit auch schlechtere Lösungen zu akzeptieren, ist die Tabu-Suche in der Lage lokale Optima zu verlassen. Eine wichtige Rolle im Suchprozess spielt dabei die Speicherung besuchter Lösungen, wodurch eine Art Gedächtnis gebildet wird. Bereits besuchte Lösungen werden in eine Tabu-Liste eingefügt und somit eine Zeit lang von der Suche ausgeschlossen. Auf diese Weise kann sich die Suche von einem lokalen Optimum entfernen. Nach Erreichen eines bestimmten Abbruchkriteriums, z.B. einer definierten Anzahl Iterationen, stoppt die Tabu-Suche mit der bis dahin besten gefundenen Lösung. Die konkreten Parameter einer Tabu-Suche wie das Abbruchkriterium, die Länge der Tabu-Liste oder das Einschränken der Nachbarschaft können zwischen verschiedenen Optimierungsproblemen variieren. Vorschläge für derartige Eigenschaften werden in [31] gemacht.

Der Ablauf der Tabu-Suche ist in Abbildung 5.1 zu sehen. Die Tabu-Suche beginnt mit einer Anfangslösung, die von einem anderen Verfahren oder zufällig erzeugt wer-

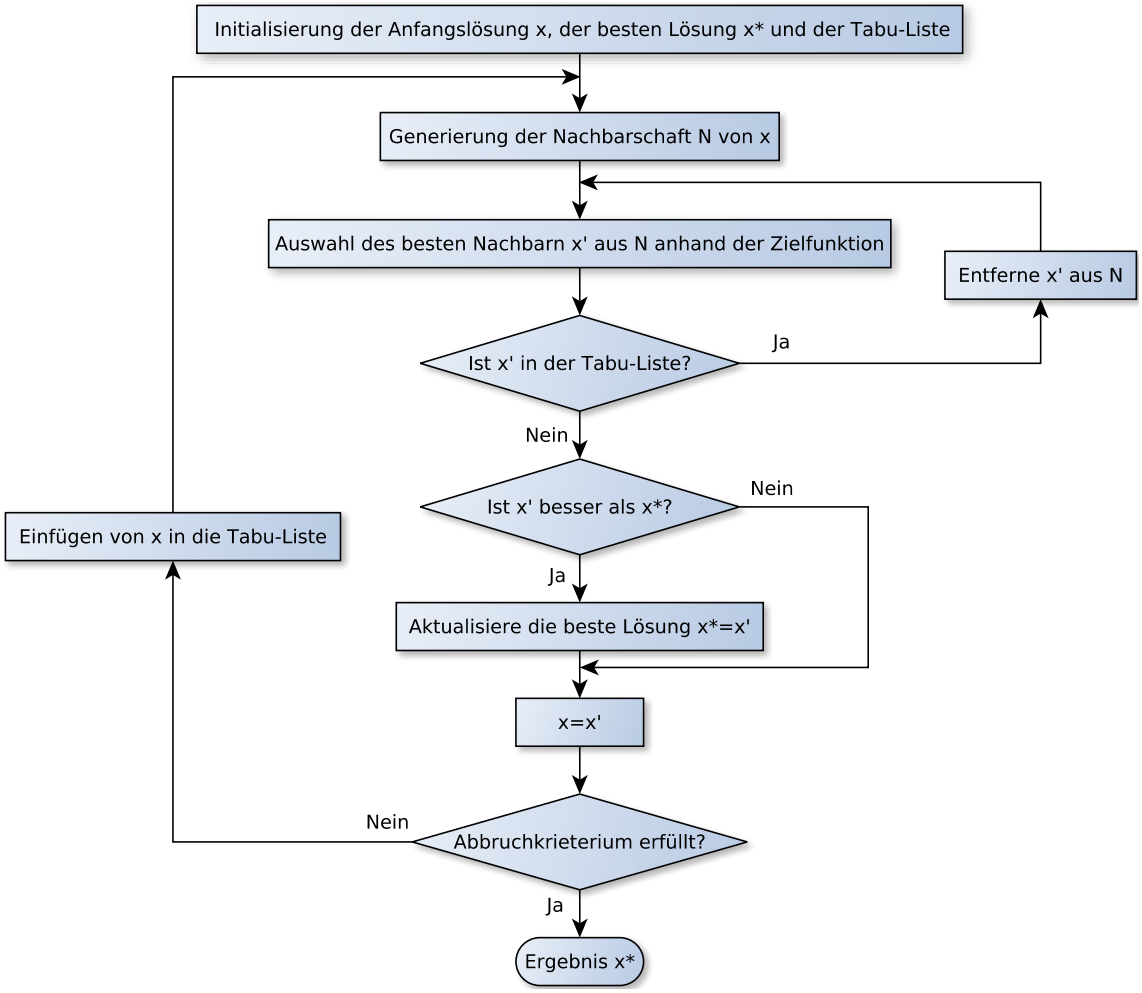


Abbildung 5.1: Programmablauf der Tabu-Suche nach [31].

den kann. In jeder Iteration wird die Nachbarschaft der aktuellen Lösung generiert. Anhand der Zielfunktion wird die beste Lösung der Nachbarschaft, die nicht in der Tabu-Liste ist, als neue aktuelle Lösung gewählt. Die neue Lösung muss dabei nicht zwangsläufig besser als die vorige Lösung sein. Die Tabu-Suche fährt mit der neuen aktuellen Lösung fort, bis das Abbruchkriterium erfüllt ist. Das Ergebnis der Tabu-Suche ist die beste gefundene Lösung aller Iterationen.

5.2 MTaSS – Multiprozessor-Task-Scheduling auf Basis einer Tabu-Suche

Die Tabu-Suche wurde als Ansatz für das Scheduling paralleler Tasks auf heterogene parallele Systeme verwendet. Zur Lösung des Schedulingproblems wurde im Rah-

Notation	Bedeutung
$T_1, \dots T_{n_T}$	Menge unabhängiger paralleler Tasks
$N_1, \dots N_{n_N}$	Menge der Rechenknoten der Zielplattform
p_j	Anzahl der Prozessorkerne des Rechenknotens N_j
K	Gesamtanzahl Prozessorkerne der heterogenen Plattform
S_{init}	Anfangsschedule
S_{best}	bester gefundener Schedule
S_{cur}	aktueller Schedule
S_N	benachbarter Schedule im Lösungsraum
$M(S_N)$	Makespan des Schedules S_N
I_{max}	maximale Anzahl Iterationen ohne Verbesserung der Lösung
G	maximale Größe der Nachbarschaft eines Schedules

Tabelle 5.1: Verwendete Notation zur Beschreibung des Schedulingproblems (oben) und des Schedulingverfahrens MTaSS (unten)

men dieser Arbeit das Schedulingverfahren Multiprocessor Task Tabu Search Scheduling (MTaSS) entwickelt, das im Folgenden vorgestellt wird. Zunächst beschrieben, wie das Schedulingproblem als Suchraum für die Tabu-Suche formuliert werden kann. Des Weiteren wird eine Strategie präsentiert, mit der die Anzahl von MTaSS zu durchsuchender Lösungen reduziert werden kann. Die in diesem Abschnitt verwendete Notation ist in Tabelle 5.1 aufgelistet.

5.2.1 Beschreibung des Suchraums

Die Tabu-Suche arbeitet wie die meisten Optimierungsverfahren auf einem Lösungsraum. Wie in Abschnitt 5.1 beschrieben, startet die Tabu-Suche ausgehend von einer Anfangslösung und sucht anschließend schrittweise nach einer besseren Lösung. In jedem Schritt werden dabei alle Lösungen in unmittelbarer Nachbarschaft zur aktuellen Lösung betrachtet.

Das Ziel des in dieser Arbeit beschriebenen Algorithmus MTaSS ist das Finden eines Schedules, in dem eine Menge paralleler Tasks einer Menge von Prozessorkernen zugewiesen wird. Das Optimierungskriterium ist dabei die Minimierung des Makespans. Der aus diesem Optimierungsproblem resultierende Lösungsraum entspricht der Menge aller Schedules. Der Aufbau eines solchen Lösungsraums wird in Abschnitt 2.4.2 beschrieben. Das Schedulingverfahren MTaSS betrachtet in jedem Schritt einen Schedule, dessen Nachbarn erzeugt und ausgewertet werden. Nach der Auswahl des besten Nachbarn wird mit diesem fortgefahren. MTaSS erzeugt und durchsucht so schrittweise einen Abschnitt des Lösungsraums, dessen Größe von Faktoren wie der Abbruchbedingung abhängt.

Algorithmus 3: Das Schedulingverfahren MTaSS.

Eingabe: • unabhängige parallele Tasks $T_1, \ldots, T_{n_T}$
• Rechenknoten $N_1, \ldots, N_{n_N}$

```
1  Erzeuge Anfangsschedule S_init
2  Initialisiere besten Schedule S_best = S_init
3  Initialisiere aktuellen Schedule S_cur = S_init
4  while (Abbruchkriterium nicht erreicht) do
5  |  Generiere Nachbarschaft von S_cur
6  |  Setze S_cur auf zufälligen Nachbarn
7  |  foreach (Nachbar S_N aus Nachbarschaft) do
8  |  |  if (S_N ist nicht in Tabu-Liste und M(S_N) < M(S_cur)) then S_cur = S_N
9  |  end
10 |  if (M(S_cur) < M(S_best)) then S_best = S_cur
11 |  Füge S_cur zur Tabu-Liste hinzu
12 end
```

5.2.2 Algorithmus des Schedulingverfahrens MTaSS

Algorithmus 3 beschreibt den Ablauf des Schedulingverfahrens MTaSS. Für die übergebenen Tasks und Rechenknoten wird zunächst ein Anfangsschedule erzeugt. Die Erzeugung kann mit einer einfachen und schnellen Heuristik oder zufällig erfolgen. In dieser Arbeit wird eine einfache Heuristik verwendet, die die Tasks nacheinander den Prozessorkernen zuweist. Dazu wird solange über die Prozessorkerne iteriert und jeder Task genau einem Kern zugewiesen, bis alle Tasks verteilt wurden. Mit dem resultierenden Schedule S_{init} werden sowohl der beste Schedule S_{best} als auch der aktuelle Schedule S_{cur} initialisiert. Nachfolgend durchsucht der Algorithmus wiederholt die Nachbarschaft des jeweils aktuellen Schedules S_{cur}. Aus der Nachbarschaft wird der Schedule S_N mit dem kleinsten Makespan $M(S_N)$ ausgewählt, der nicht in der Tabu-Liste ist. Dieser Schedule S_N wird somit der neue aktuelle Schedule S_{cur}. Verbessert der gefundene Schedule S_{cur} den besten bisher gefundenen Schedule S_{best} hinsichtlich des Makespans, dann wird S_{cur} als neuer Schedule S_{best} übernommen. Am Ende jeder Iteration wird der gefundene Schedule S_{cur} zur Tabu-Liste hinzugefügt. Sobald das Abbruchkriterium erreicht wurde, stoppt der Algorithmus und gibt den besten gefundenen Schedule S_{best} zurück.

Als Abbruchkriterium werden in [31] folgende Bedingungen vorgeschlagen:

- Eine optimale Lösung wurde gefunden.
- Die Nachbarschaft ist leer oder bereits komplett in der Tabu-Liste enthalten.
- Eine vordefinierte maximale Anzahl an Iterationen wurde überschritten.
- Seit der letzten Aktualisierung der besten gefundenen Lösung wurde eine vordefinierte maximale Anzahl an Iterationen überschritten.

Für die Umsetzung von MTaSS wurde der letzte dieser Vorschläge verwendet. Die Suche stoppt nach einer gewissen Anzahl aufeinanderfolgender Iterationen, in denen der beste bisher gefundene Schedule S_{best} nicht aktualisiert wurde. Die Anzahl Iterationen ohne Verbesserung I_{max} wird auf

$$I_{max} = n_T \cdot K \tag{5.1}$$

festgelegt, wobei n_T der Anzahl paralleler Tasks und K der Gesamtanzahl vorhandener Prozessorkerne entspricht.

5.2.3 Auswahl der vielversprechendsten Nachbarn

Bereits in [31] wird in bestimmten Fällen eine Einschränkung der Nachbarschaften empfohlen. Im Idealfall kann die Größe der zu durchsuchenden Nachbarschaft bis auf eine einzelne Lösung begrenzt werden. Auch für das Scheduling paralleler Tasks können die von MTaSS zu durchsuchenden Nachbarschaften innerhalb des Lösungsraums sehr groß werden. Um die von MTaSS für das Scheduling benötigte Zeit zu senken, wird eine Methode zur Eingrenzung der Suche auf bestimmte Nachbarn benötigt. Die Methode sollte dabei allerdings die Qualität der gefundenen Lösung nicht negativ beeinflussen.

Das Optimierungsziel der Methode MTaSS ist die Reduzierung des Makespans des jeweils aktuellen Schedules. Daher ist es ausreichend in jedem Schritt nur die Tasks zu betrachten, die den Makespan beeinflussen, um die Größe der Nachbarschaft zu reduzieren. Bei der Generierung der Nachbarschaft eines Schedules werden in dieser Strategie nun nicht mehr alle Tasks neu zugewiesen, sondern nur diese ausgewählten Tasks. Zur Auswahl der Tasks gehört der Task mit der größten Endezeit und alle Tasks die denselben Prozessorkernen zugewiesen wurden. Mit diesen Tasks wird die Nachbarschaft dann, wie in Abschnitt 5.2.1 beschrieben, erzeugt.

Beispiel für die Auswahl der vielversprechendsten Nachbarn

Abbildung 5.2 zeigt die Strategie zur Eingrenzung der Nachbarschaft beispielhaft für das Scheduling fünf paralleler Tasks auf einen Rechenknoten mit 4 Prozessorkernen. Dargestellt ist ein Teil der Nachbarschaft des Schedules S_1. Im Schedule S_1 wird der Makespan durch die gelben Tasks auf Kern 1 bestimmt. Gemäß der vorgestellten Strategie werden zur Generierung der Nachbarn von S_1 nur diese beiden Tasks separat jeder Kombination von Prozessorkernen neu zugewiesen. Zur besseren Übersichtlichkeit sind nur einige wenige dieser Nachbarn dargestellt. Für die Generierung der kompletten Nachbarschaft des Schedules S_1 müssten auch die grauen Tasks neu zugewiesen werden. Diese komplette Nachbarschaft würde aus 75 Schedules bestehen, wohingegen die vorgestellte Strategie zu einer Reduktion auf 30 Schedules führt.

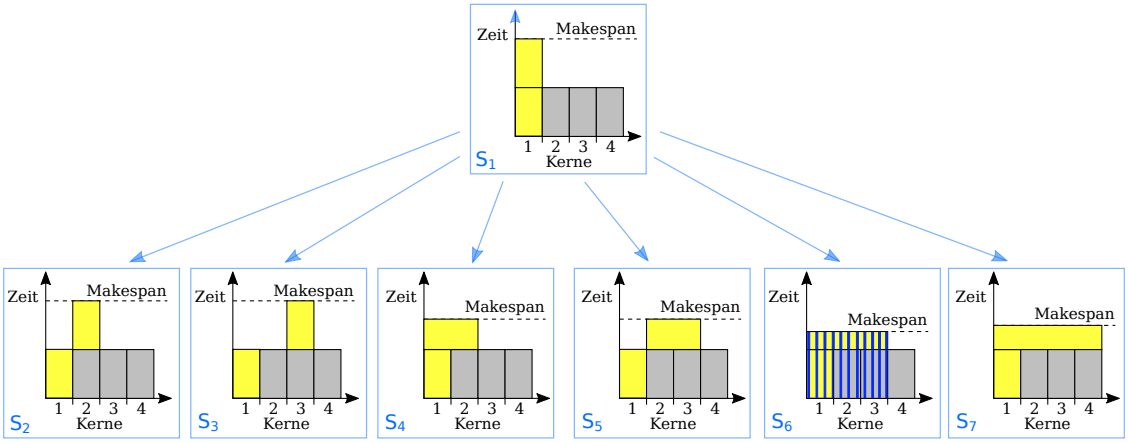


Abbildung 5.2: Teil der Nachbarschaft des Schedules S_1 für das Scheduling fünf paralleler Tasks auf einen Rechenknoten mit 4 Prozessorkernen. Die ausgewählten Tasks zur Generierung der vielversprechendsten Nachbarn von S_1 sind gelb und von S_6 schraffiert dargestellt.

Im nächsten Schritt würde MTASS mit dem benachbarten Schedule mit dem geringsten Makespan (hier S_6) fortfahren. Im Schedule S_6 bestimmt der gelbe Task den Makespan, der den Kernen 1, 2 und 3 zugewiesen wurde sowie alle Tasks die ebenfalls diesen Kernen zugewiesen wurden. Diese blau schraffierten Tasks würden dann zur Generierung der Nachbarschaft von S_6 neu zugewiesen, während der graue Task auf Kern 4 nicht mit einbezogen wird.

5.2.4 Analyse der Komplexität

Die Laufzeit des Schedulingverfahrens MTASS hängt direkt von der Anzahl zu betrachtender Schedules ab. In jeder Iteration von MTASS wird die Nachbarschaft des jeweils aktuellen Schedules erzeugt und über alle diese Nachbarn iteriert. Zur Generierung der Nachbarschaft wird jeder der n_T Tasks jeder Kombination aus 1 bis p_j Prozessorkernen eines jeden Rechenknotens $N_j, j = 1, \ldots, n_N$ neu zugewiesen. Ohne die Verwendung einer Reduktionsstrategie ergibt sich die Größe G einer jeden Nachbarschaft aus der Taskanzahl n_T multipliziert mit der Summe aller Binomialkoeffizienten für p_j über p aufsummiert über die Anzahl an Rechenknoten. Die Größe G der Nachbarschaft eines Schedules kann mithilfe der Gleichung

$$G = n_T \cdot \sum_{j=1}^{n_N} \sum_{p=1}^{p_j} \binom{p_j}{p} = n_T \cdot \sum_{j=1}^{n_N} (2^{p_j} - 1) \tag{5.2}$$

berechnet werden, wobei n_T die Taskanzahl, n_N die Anzahl an Rechenknoten und p_j die Anzahl Prozessorkerne des Rechenknotens N_j bezeichnen. Die Anzahl von

MTASS zu betrachtender Schedules ergibt sich demnach aus der Anzahl Iterationen I multipliziert mit der Größe G der Nachbarschaft und kann wie folgt berechnet werden:

$$\#Schedules = I \cdot G = I \cdot n_T \cdot \sum_{j=1}^{n_N} (2^{p_j} - 1) \tag{5.3}$$

Die Anzahl zu betrachtender Schedules und somit auch die Laufzeit von MTASS steigt demnach exponentiell mit der Anzahl Prozessorkerne der Rechenknoten. Wie in Abschnitt 5.2.2 beschrieben, stoppt der Algorithmus nach I_{max} (Gleichung (5.1), Seite 69) Iterationen ohne Verbesserung der besten bisher gefundenen Lösung. Daraus folgt, dass MTASS mindestens I_{max} Iterationen durchführt, in denen jeweils der Makespan von G Nachbarn ausgewertet werden muss. Die genaue Anzahl Iterationen I hängt vom konkreten Schedulingproblem ab. Einen Einfluss hat dabei nicht nur die Anzahl der Tasks und Rechenknoten, sondern unter anderem auch das Laufzeitverhalten der Tasks und die Zusammensetzung des heterogenen parallelen Systems.

5.3 Experimentelle Auswertung

In diesem Abschnitt wird das Schedulingverfahren MTASS mit den in Abschnitt 2.3.1 vorgestellten List-Scheduling-Verfahren HCPA und Δ-CTS verglichen. Die Verfahren werden verwendet, um Schedules für die Ausführung paralleler SPLASH-3-Benchmark-Tasks auf einem heterogenen parallelen System zu erstellen. Die Tasks werden anhand der erzeugten Schedules ausgeführt und die Gesamtlaufzeit gemessen, die als Basis für den Vergleich genutzt wird. Anschließend wird der Einfluss der in Abschnitt 5.2.3 vorgestellten Strategie zur Auswahl der vielversprechendsten Nachbarn auf die Laufzeit des Verfahrens MTASS untersucht.

5.3.1 Zielplattform und betrachtete Anwendungen

Der verwendete heterogene Rechencluster besteht aus 3 Rechenknoten mit insgesamt 16 Prozessorkernen. In Tabelle 5.2 sind die Eigenschaften dieser Rechenknoten aufgelistet. Als Referenzrechenknoten wird der Rechenknoten `sb1` verwendet, da dieser über die größte Anzahl Prozessorkerne verfügt. Auf diesem Rechenknoten wird somit die Bestimmung der Parameter des Kostenmodells (vgl. Anschnitt 3.2.1) durchgeführt. Sowohl für die Ausführung der Schedulingverfahren als auch das Starten der Tasks gemäß der ermittelten Schedules wird ein separater Front-End-Knoten eingesetzt. Dieser ist mit einem Intel Xeon E5-2683-Prozessor und 128 GB Arbeitsspeicher ausgestattet und über SSH-Verbindungen mit den Rechenknoten verbunden. Der Start eines Tasks erfolgt, sobald alle seine Vorgänger im Schedule ihre Ausführung beendet haben. Während der Ausführung eines Schedules wird die Gesamtlaufzeit vom Start des ersten Tasks bis zur Beendigung des letzten Tasks

Rechen-knoten	Prozessor	Architektur	Erscheinungs-jahr	Anzahl Kerne	RAM-Größe	Frequenz in GHz
`skylake1`	Intel i7-6700	Skylake	2015	4	16 GB	3.40
`hw1`	Intel i7-4770K	Haswell	2013	4	16 GB	3.50
`sb1`	Intel Xeon E5-2650	Sandy Bridge	2012	8	32 GB	2.00

Tabelle 5.2: Liste der Rechenknoten des verwendeten heterogenen Rechenclusters.

gemessen. Eventuelle Schwankungen der Gesamtlaufzeit werden berücksichtigt, indem jeder Schedule zehnmal ausgeführt und die Durchschnittszeit angegeben wird. Die in den Laufzeitmessungen verwendeten parallelen Tasks sind jeweils die zwei Anwendungen BARNES und FMM sowie die zwei Kernel CHOLESKY und LU der SPLASH-3-Benchmark-Suite. Diese werden, sofern nicht anders angegeben, mit den Standardparametern oder dem Parametersatz „parsec-simlarge“ ausgeführt.

5.3.2 Vergleich der Gesamtausführungszeit der ermittelten Schedules

Im Folgenden sollen die vom Verfahren MTASS ermittelten Schedules hinsichtlich ihrer Gesamtausführungszeit untersucht werden. Dazu werden MTASS sowie die List-Scheduling-Verfahren HCPA und Δ-CTS verwendet, um jeweils 1 bis 32 SPLASH-3-Benchmark-Tasks auf dem in Tabelle 5.2 gezeigten heterogenen Cluster auszuführen. Weitere Messergebnisse befinden sich im Anhang A.

In Abbildung 5.3 (oben) sind die gemessenen Gesamtlaufzeiten für die Ausführung der BARNES- (links) und der FMM-Anwendungstasks (rechts) in Abhängigkeit von der Taskanzahl dargestellt. Für beide Arten von Anwendungstasks führt MTASS zu geringeren oder ähnlichen Laufzeiten verglichen mit den Ergebnissen der beiden List-Scheduling-Verfahren HCPA und Δ-CTS. Besonders für eine größere Anzahl Tasks werden von MTASS deutlich geringere Laufzeiten erreicht. Die maximale Reduktion der Gesamtlaufzeit gegenüber HCPA und Δ-CTS entspricht dabei 40% für die BARNES-Tasks und 38% für die FMM-Tasks. Eine weitere Beobachtung bezüglich MTASS ist der gleichmäßigere Anstieg der Gesamtlaufzeit mit steigender Taskanzahl. Dagegen führt die Verwendung von HCPA und Δ-CTS teilweise zu einem sprunghaften Anstieg der Laufzeit. Solche Anstiege können insbesondere für 13 und 25 Tasks beobachtet werden, zu deren Ausführung ein leistungsschwächerer Rechenknoten hinzugezogen wird.

Abbildung 5.3 (unten) zeigt die gemessenen Gesamtlaufzeiten für die Ausführung der CHOLESKY- (links) und der LU-Kerneltasks (rechts) in Abhängigkeit von der Anzahl paralleler Tasks. Analog zu den Anwendungstasks führt MTASS im Vergleich zu HCPA und Δ-CTS auch für beide Arten von Kerneltasks zu geringeren

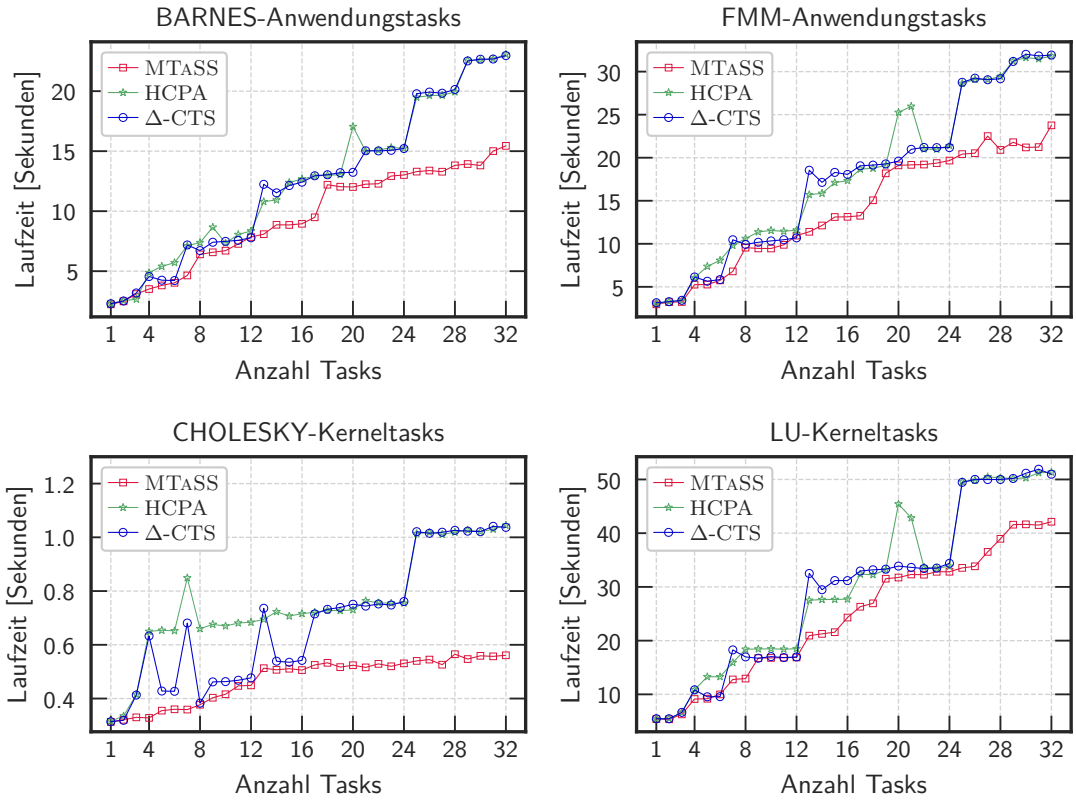


Abbildung 5.3: Oben: Gemessene Gesamtausführungszeit von BARNES- (links) und FMM-Anwendungstasks (rechts) in Abhängigkeit von der Taskanzahl auf dem in Tablle 5.2 beschriebenen heterogenen Cluster. Unten: Gemessene Gesamtausführungszeit von CHOLESKY- (links) und LU-Kerneltasks (rechts) in Abhängigkeit von der Taskanzahl auf dem in Tablle 5.2 beschriebenen heterogenen Cluster.

oder ähnlichen Laufzeiten. Für die CHOLESKY-Tasks erreicht MTaSS eine bis zu 31% geringere Gesamtlaufzeit als beide List-Scheduling-Verfahren. Eine mögliche Ursache für diese signifikanten Unterschiede könnte die kurze Ausführungszeit der CHOLESKY-Tasks sein. Die Tasks profitieren dadurch kaum von einer jeweiligen parallelen Ausführung auf einer größeren Anzahl Prozessorkernen. Insbesondere HCPA aber auch Δ-CTS tendieren jedoch dazu, den Tasks viele Kerne zuzuweisen. Daher erzielen beide Verfahren auch ähnliche Ergebnisse, mit Ausnahme des Bereichs von 1 bis 16 Tasks, in dem HCPA bis zu 87% höhere Laufzeiten liefert. MTaSS hingegen entdeckt bei dessen Suche auch Schedules, in denen den Tasks weniger Prozessorkerne zugewiesen werden. Für die LU-Tasks lassen sich deutlich geringere Unterschiede zwischen den drei verwendeten Verfahren feststellen. Besonders auffällig sind allerdings die großen Anstiege der Laufzeit der von HCPA und Δ-CTS produzierten Schedules für 13 und 25 Tasks, die durch die Zuweisung einiger Tasks zu einem leistungsschwächeren Rechenknoten entstehen. An diesen Stellen erreicht MTaSS eine bis zu 32% geringere Laufzeit im Vergleich zu HCPA und eine bis zu 36% geringere Laufzeit verglichen mit Δ-CTS.

5.3.3 Vergleich der Laufzeit der Schedulingverfahren

Auf der einen Seite ermöglicht eine längere Suchzeit dem Schedulingverfahren MTaSS mehr Schedules zu betrachten und damit eine bessere Lösung zu finden. Andererseits kann der für die Ausführung des Schedules erreichte Zeitgewinn gegenüber einer schnelleren Heuristik durch die lange Laufzeit des Schedulingverfahrens konterkariert werden. Daher wird für den Vergleich der Laufzeit der Schedulingverfahren die Summe aus der Zeit für das Scheduling und der Ausführungszeit des ermittelten Schedules betrachtet. Zusätzlich wird der Einfluss der in Abschnitt 5.2.3 vorgestellten Strategie zur Auswahl der vielversprechendsten Nachbarn auf die Laufzeit von MTaSS untersucht. Für beide Analysen werden jeweils die beiden Kerneltasks LU und CHOLESKY verwendet, die in den vorigen Laufzeitmessungen zu den längsten und kürzesten Schedules geführt haben. Neben MTaSS werden die beiden List-Scheduling-Verfahren HCPA und Δ-CTS verwendet, um jeweils 1 bis 32 Tasks auf dem in Tabelle 5.2 gezeigten heterogenen Cluster auszuführen.

Abbildung 5.4 zeigt die Laufzeit des Verfahrens MTaSS für das Scheduling von LU- (links) und CHOLESKY-Kerneltasks (rechts) auf den in Tabelle 5.2 beschriebenen heterogenen Cluster. Die Laufzeit wurde jeweils mit und ohne implementierte Strategie zur Auswahl der vielversprechendsten Nachbarn gemessen. Für beide Varianten ist erkennbar, dass die Auswahl der Tasks nur einen geringen Einfluss auf die Laufzeit des Schedulingvorgangs hat. Die Einschränkung der Nachbarschaft führt für beide Kerneltasks zu einer deutlichen Reduktion der Laufzeit von bis zu 83%. Zusätzlich verläuft der Anstieg der Laufzeit mit zunehmender Taskanzahl unter Ver-

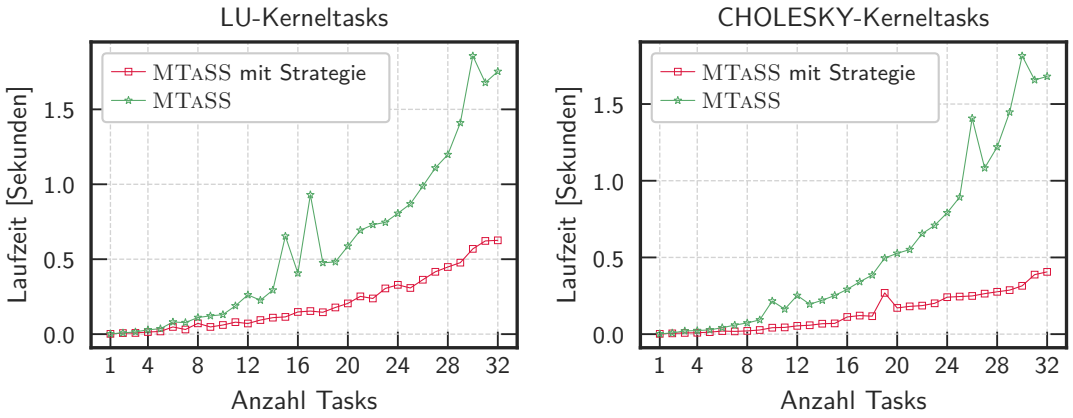


Abbildung 5.4: Ausführungszeit des Verfahrens MTASS mit und ohne Strategie zur Auswahl der vielversprechendsten Nachbarn für das Scheduling von LU- (links) und CHOLESKY-Kerneltasks (rechts) auf den in Tabelle 5.2 beschriebenen heterogenen Cluster.

wendung der Strategie gleichmäßiger und weniger steil verglichen mit MTASS ohne Strategie.

Die List-Scheduling-Verfahren HCPA und Δ-CTS laufen deutlich schneller als das suchbasierte Verfahren MTASS. Für das Scheduling von 32 LU-Kerneltasks auf den betrachteten Cluster beispielsweise benötigen HCPA und Δ-CTS etwa 7 ms, MTASS hingegen etwa 620 ms. Wird die gesamte Zeit für das Scheduling und die Ausführung des ermittelten Schedules betrachtet, so kann MTASS die Verfahren HCPA und Δ-CTS zumeist dennoch übertreffen. In Abbildung 5.5 ist die prozentuale Differenz dieser gesamten Zeit zwischen MTASS und den beiden List-Scheduling-Verfahren zu sehen. Betrachtet wird dabei das Scheduling und die entsprechende Ausführung von LU- (links) und CHOLESKY-Kerneltasks (rechts) auf den in Tabelle 5.2 gezeigten heterogenen Cluster. Aufgrund der geringeren Ausführungszeit der erzeugten Schedules fällt die Gesamtzeit für Scheduling und Ausführung der LU-Tasks für MTASS verglichen mit HCPA und Δ-CTS zumeist deutlich niedriger aus. Im Durchschnitt liegt Gesamtzeit für HCPA etwa 21% und für Δ-CTS etwa 18% höher als für MTASS. In einigen wenigen Fällen liegen die Gesamtzeiten HCPA und Δ-CTS auf dem Niveau von MTASS oder leicht darunter. Für die CHOLESKY-Tasks ergibt sich ein etwas anderes Bild. Aufgrund der sehr kurzen Laufzeit der einzelnen Tasks fällt der Unterschied zwischen den Ausführungszeiten der erzeugten Schedules für die drei Verfahren geringer aus. Dadurch fällt insbesondere für eine größere Anzahl Tasks die längere Schedulingzeit von MTASS stärker ins Gewicht. Für kleinere Taskanzahlen sind die Gesamtlaufzeiten von HCPA bis zu 126% und von Δ-CTS bis zu 88% höher. Für einige wenige Taskanzahlen unterbieten die Gesamtlaufzeiten von Δ-CTS und HCPA die Ergebnisse von MTASS um bis zu 12%.

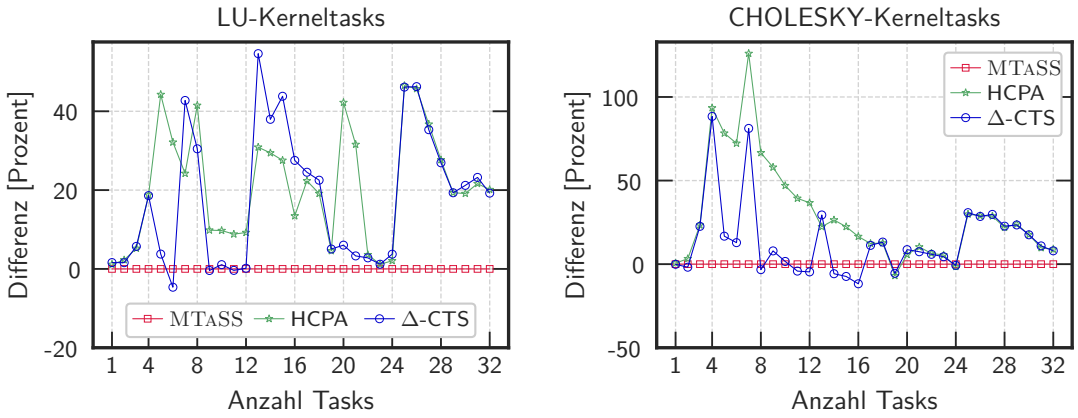


Abbildung 5.5: Differenz der Gesamtzeit für das Scheduling und die Ausführung des ermittelten Schedules zwischen MTASS und den beiden List-Scheduling-Verfahren HCPA und Δ-CTS für das Scheduling von LU- (links) und CHOLESKY-Kerneltasks (rechts) auf den in Tabelle 5.2 beschriebenen heterogenen Cluster.

5.4 Zusammenfassung

Mit Multiprocessor Task Tabu Search Scheduling (MTASS) wurde in diesem Kapitel ein neues Schedulingverfahren für parallele Tasks vorgestellt, das auf der Tabu-Suche basiert. Neben dem Verfahren wurde der Aufbau des Suchraums beschrieben und wie dieser von MTASS durchsucht wird. Zudem wurde eine Strategie zur weiteren Eingrenzung dieses Suchraums vorgestellt, mit der die Laufzeit von MTASS um bis zu 83% reduziert werden konnte. Die Komplexität des Verfahrens wurde sowohl formal als auch in Laufzeitmessungen untersucht. Dabei zeigte sich, dass die Laufzeit von MTASS trotz der Eingrenzung des Suchraums etwas länger als bei den zwei List-Scheduling-Heuristiken HCPA und Δ-CTS ist. Jedoch konnte MTASS im Vergleich zu den beiden anderen Verfahren eine Reduktion der Ausführungszeit um bis zu 40% erreichen. Insgesamt ist der Gewinn bei der Ausführungszeit der erzeugten Schedules größer als der Laufzeitunterschied der Verfahren, wodurch sich der Einsatz von MTASS durchaus lohnen kann.

6 Scheduling paralleler Tasks basierend auf Simulated Annealing

Simulated Annealing [42] ist eine probabilistische Suchmethode zur näherungsweisen Lösung komplexer Optimierungsprobleme. Im Rahmen dieser Arbeit wurde mit Simulated Annealing Multiprocessor Task Scheduling (SAMT) ein Verfahren entwickelt, das Simulated Annealing als Ansatz für das Scheduling unabhängiger paralleler Tasks verwendet. Dieses Kapitel stellt dieses Verfahren im Anschluss an den generellen Ablauf von Simulated Annealing zur Lösung von Optimierungsproblemen vor. Den Abschluss des Kapitels bildet die Auswertung von Experimenten, in denen die Leistungsfähigkeit von SAMT in Bezug auf die Ausführungszeit der ermittelten Schedules untersucht wird.

6.1 Das Verfahren der simulierten Abkühlung

Simulated Annealing (simulierte Abkühlung) ist eine Metaheuristik zur näherungsweisen Lösung komplexer Optimierungsprobleme. Das Verfahren ist inspiriert vom Abkühlen und Erhärten geschmolzener Metalle. Mit sinkender Temperatur nimmt auch die Beweglichkeit der Atome ab, bis diese sich zu einem stabilen Kristallgitter geordnet haben. Übertragen auf Optimierungsprobleme entsprechen die Atomzustände den Lösungen des Lösungsraums, die Energie des Zustands dem Wert der Zielfunktion und die Temperatur einer Wahrscheinlichkeit, mit der auch schlechtere Lösungen akzeptiert werden.

Simulated Annealing ist ein probabilistischer Algorithmus mit der Konsequenz, dass sich die Qualität der gefundenen Lösungen zwischen einzelnen Läufen unterscheiden kann. Der Algorithmus beginnt mit einer Anfangslösung bei einer Temperatur, die mit der Anzahl Iterationen abnimmt. In jeder Iteration wird eine zufällige Lösung aus der unmittelbaren Nachbarschaft gewählt. Als Nachbarn werden Lösungen bezeichnet, die sich nur geringfügig unterscheiden. Die gewählte Lösung wird als neue aktuelle Lösung akzeptiert, wenn sie eine Verbesserung im Sinne der Zielfunktion darstellt. Um lokalen Optima zu entkommen, werden schlechtere Lösungen mit einer temperaturabhängigen Wahrscheinlichkeit ebenfalls akzeptiert. Da mit abnehmender Temperatur auch diese Wahrscheinlichkeit sinkt, wird Simulated Annealing im Verlauf mehr und mehr zu einer lokalen Suche. Der Algorithmus stoppt mit der besten bis dahin gefundenen Lösung, wenn eine vordefinierte Abbruchbedingung erfüllt ist. Das kann beispielsweise ein bestimmter Temperaturwert oder

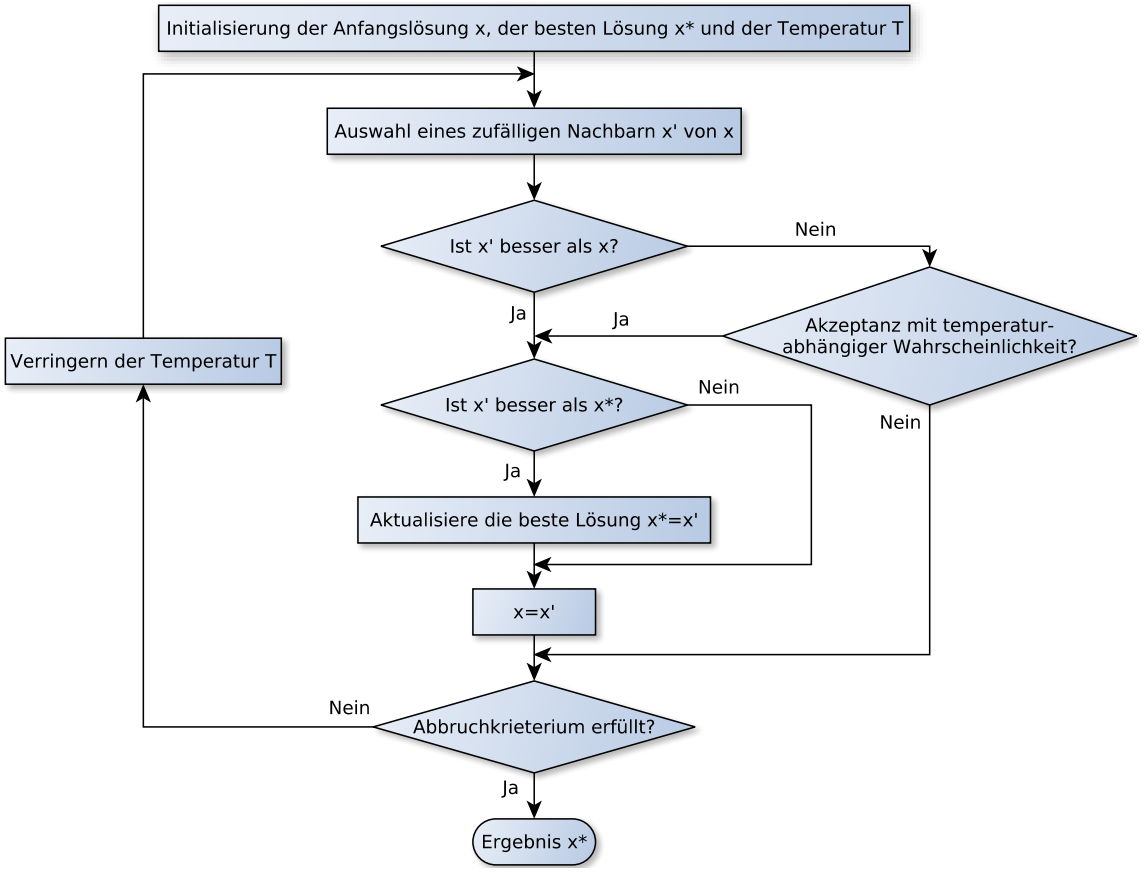


Abbildung 6.1: Ablauf der simulierten Abkühlung.

eine bestimmte Anzahl Iterationen sein. Zur Anwendung von Simulated Annealing auf ein konkretes Optimierungsproblem müssen insbesondere der Lösungsraum, die Akzeptanz schlechterer Lösungen und die Temperatur sowie die Abkühlungsrate problemspezifisch festgelegt werden.

Abbildung 6.1 zeigt den Ablauf des Verfahrens der simulierten Abkühlung. Das Verfahren startet bei einem vordefinierten Temperaturwert T und mit einer Anfangslösung x. Diese Anfangslösung kann entweder zufällig oder mithilfe eines anderen Verfahrens erzeugt werden. In jeder Iteration wird ein zufälliger Nachbar x' der aktuellen Lösung x gewählt. Eine bessere Lösung im Hinblick auf die Zielfunktion wird immer akzeptiert, während schlechtere Lösungen nur mit einer temperaturabhängigen Wahrscheinlichkeit akzeptiert werden. Falls die akzeptierte Lösung x' die beste bisher gefundene Lösung x^* verbessert, wird x^* durch x' ersetzt. Anschließend wird die akzeptierte Lösung x' zur neuen aktuellen Lösung x. Falls die Abbruchbedingung noch nicht erfüllt ist, beginnt eine neue Iteration mit verringerter Temperatur T. Bei erfüllter Abbruchbedingung terminiert das Verfahren mit der besten gefundenen Lösung x^*.

6.2 SAMT - Simulated Annealing für das Multiprozessor-Task-Scheduling

In diesem Abschnitt wird beschrieben, wie Simulated Annealing als Ansatz für das Scheduling paralleler Tasks auf heterogene parallele Systeme verwendet werden kann. Zunächst wird eine Möglichkeit gezeigt, ein solches Schedulingproblem als Suchraum für die Anwendung von Simulated Annealing zu formulieren. Anschließend wird das Schedulingverfahren Simulated Annealing Multiprocessor Task Scheduling (SAMT) vorgestellt und analysiert, welches zur Lösung des Schedulingproblems entwickelt wurde. Tabelle 6.1 listet die in diesem Abschnitt verwendete Notation auf.

6.2.1 Beschreibung des Suchraums

Wie die Tabu-Suche arbeitet auch Simulated Annealing auf einem Lösungsraum. Die Suche startet bei einer Anfangslösung und wählt in jedem Schritt eine neue Lösung aus dessen Nachbarschaft aus. Im Gegensatz zur Tabu-Suche bestimmt Simulated Annealing allerdings nicht den besten Nachbarn, sondern wählt diesen zufällig aus. Das hat den entscheidenden Vorteil, dass anstelle der gesamten Nachbarschaft nur ein einziger Nachbar generiert werden muss.

Bei der Anwendung von Simulated Annealing auf das Scheduling paralleler Tasks entspricht jede Lösung im Lösungsraum einem Schedule. In jedem Schritt des Algorithmus SAMT wird ein aktueller Schedule sowie ein zufällig gewählter Nachbar dieses Schedules betrachtet. Da die Bestimmung des Nachbarn zufällig erfolgt, ist es nicht notwendig, die komplette Nachbarschaft zu erzeugen und daraus anschließend einen Nachbarn zufällig auszuwählen. Vielmehr kann ausgehend vom aktuellen Schedule ein einzelner Nachbar zufällig erzeugt werden. Die Generierung eines Nachbarn verläuft dabei wie in Abschnitt 2.4.2 beschrieben. Zunächst wird ein Task zufällig ausgewählt, aus dem aktuellen Schedule entfernt und anschließend neu zugewiesen. Die Zuweisung erfolgt zu einer zufälligen Auswahl von Prozessorkernen auf einem ebenfalls zufällig bestimmten Rechenknoten.

6.2.2 Algorithmus der Schedulingmethode SAMT

Algorithmus 4 beschreibt den Ablauf des Schedulingverfahrens SAMT. Zu Beginn wird die Temperatur mit einem Startwert initialisiert und für die übergebenen Tasks und Rechenknoten ein Anfangsschedule erzeugt. Zur Erzeugung eines Anfangsschedules iteriert SAMT solange über die Prozessorkerne, bis alle Tasks verteilt wurden und weist dabei jeden Task genau einem Kern zu. Die anschließende Hauptschleife des Algorithmus wird solange durchlaufen, bis die Temperatur einen definierten Minimalwert erreicht hat. Innerhalb dieser Schleife wird ein zufälliger Nachbar S_N des

Notation	Bedeutung
$T_1, \ldots T_{n_T}$	Menge unabhängiger paralleler Tasks
$N_1, \ldots N_{n_N}$	Menge der Rechenknoten der Zielplattform
p_j	Anzahl der Prozessorkerne des Rechenknotens N_j
K	Gesamtanzahl Prozessorkerne der heterogenen Plattform
S_{init}	Anfangsschedule
S_{best}	bester gefundener Schedule
S_{cur}	aktueller Schedule
S_N	benachbarter Schedule im Lösungsraum
$M(S_N)$	Makespan des Schedules S_N
T	aktuelle Temperatur
T_{min}	minimale Temperatur
L	Anzahl Iterationen pro Temperaturwert

Tabelle 6.1: Verwendete Notation zur Beschreibung des Schedulingproblems (oben) und des Schedulingverfahrens SAMT (unten)

aktuellen Schedules S_{cur} bestimmt. Der Schedule S_N wird als neuer aktueller Schedule S_{cur} akzeptiert, wenn dessen Makespan $M(S_N)$ geringer ist oder die Bedingungen für die Akzeptanz einer schlechteren Lösung erfüllt sind. Das Verfahren speichert und aktualisiert die beste bisher gefundene Lösung S_{best} in jedem Schritt. Am Ende jeder Iteration wird die Temperatur mit einer festgelegten Abkühlgeschwindigkeit reduziert. Wenn die Minimaltemperatur T_{min} erreicht wurde, terminiert der Algorithmus mit dem besten gefundenen Schedule S_{best}. Im Folgenden werden weitere Details zu Temperatur, Abkühlgeschwindigkeit und Akzeptanz schlechterer Lösungen beschrieben. Bewährte Ansätze zur Umsetzung dieser Parameter werden in [57] vorgestellt. Zudem werden spezielle Empfehlungen für das Scheduling sequentieller Tasks gegeben. Diese Empfehlungen wurden beim Entwurf des Verfahrens SAMT berücksichtigt.

Temperaturverlauf

In SAMT wird eine geometrische Abkühlgeschwindigkeit verwendet. Dabei handelt es sich um den gängigsten und in [57] empfohlenen Ansatz für das Task-Scheduling mittels simulierter Abkühlung. Die Temperatur wird auf den Bereich zwischen 0 und 1 beschränkt und in jedem Schritt um einen konstanten Faktor verringert. Die konkreten Werte, die in SAMT verwendet werden, wurden anhand von Messungen ermittelt. Im Falle von SAMT liegt die Starttemperatur bei 1, die Minimaltemperatur T_{min} bei 10^{-10} und der konstante Faktor q beträgt 0,95. Zur Verringerung der

Algorithmus 4: Das Schedulingverfahren SAMT.

Eingabe: • unabhängige parallele Tasks $T_1, \ldots, T_{n_T}$
• Rechenknoten $N_1, \ldots, N_{n_N}$

```
1  Erzeuge Anfangsschedule S_init
2  Initialisiere besten Schedule S_best = S_init
3  Initialisiere aktuellen Schedule S_cur = S_init
4  Initialisiere Temperatur T
5  while (T > T_min) do
6  |  Bestimme zufälligen Nachbarn S_N des aktuellen Schedules S_cur
7  |  if ((M(S_N) < M(S_cur)) oder temperaturabhängige Akzeptanz) then
8  |  |  if (M(S_N) < M(S_best)) then S_best = S_N
9  |  |  S_N wird zum neuen aktuellen Schedule S_cur
10 |  end
11 |  Verringere Temperatur T
12 end
```

Temperatur wird diese nach einer gewissen Anzahl Iterationen mit q multipliziert. Diese Anzahl Iterationen L wird auf

$$L = n_T \cdot K^2 \tag{6.1}$$

festgelegt, wobei n_T die Anzahl paralleler Tasks und K die Gesamtanzahl der Prozessorkerne des heterogenen parallelen Systems bezeichnen. Die Temperatur T in Iteration i kann demnach wie folgt berechnet werden:

$$T = 0.95^{\lfloor \frac{i}{L} \rfloor} \tag{6.2}$$

Akzeptanz schlechterer Lösungen

Die Akzeptanz schlechterer Lösungen ermöglicht dem Algorithmus das Entkommen aus lokalen Optima. Nach der Empfehlung von [57] wird eine normierte, inverse Exponentialfunktion zur Ermittlung der Akzeptanz verwendet. Die Wahrscheinlichkeit, mit der eine schlechtere Lösung akzeptiert wird, hängt vom Qualitätsunterschied zur aktuellen Lösung und der Temperatur ab. In SAMT werden die Makespans des aktuellen Schedules S_{cur}, des zufällig bestimmten Nachbarn S_N und des Anfangsschedules S_{init} sowie die Temperatur T in die Berechnung einbezogen. Eine schlechtere Lösung S_N wird akzeptiert, wenn ein zufälliger Wert zwischen 0 und 1 kleiner als der ermittelte Funktionswert ist:

$$\text{Akzeptanz von } S_N \Longleftrightarrow \text{Zufallswert}(0,1) < \frac{1}{1 + \exp(\frac{M(S_N) - M(S_{cur})}{M(S_{init}) \cdot T})} \tag{6.3}$$

6.2.3 Analyse der Komplexität

Die Laufzeit eines Schedulingverfahrens ist ein entscheidendes Kriterium für dessen Praktikabilität. Daher wird im Folgenden die Laufzeitkomplexität des Verfahrens SAMT analysiert.

Lemma 6.1. *Die Laufzeit von SAMT verhält sich quadratisch zur Anzahl Prozessorkerne des heterogenen parallelen Systems.*

Beweis. Für eine festgelegte Starttemperatur T_0, einen konstanten Faktor q und eine Minimaltemperatur T_{min} verwendet SAMT $\log_q \frac{T_{min}}{T_0}$ verschiedene Temperaturen. Die Anzahl Iterationen L, die für jeden Temperaturwert durchgeführt werden, ergibt sich aus Gleichung (6.1) (Seite 81). Für n_T parallele Tasks und ein paralleles System mit insgesamt K Prozessorkernen kann daher für SAMT die Gesamtanzahl Iterationen wie folgt berechnet werden:

$$\#Iterationen = \log_q \frac{T_{min}}{T_0} \cdot n_T \cdot K^2 \tag{6.4}$$

□

Unabhängig von den gefundenen Lösungen führt SAMT eine feste Anzahl an Iterationen aus, die quadratisch mit der Gesamtanzahl Prozessorkerne des heterogenen parallelen Systems wächst.

6.3 Experimentelle Auswertung

Zur Auswertung des Schedulingverfahrens SAMT werden Laufzeitmessungen für die Ausführung paralleler SPLASH-3-Benchmark-Tasks auf einem heterogenen parallelen System durchgeführt. Für einen Vergleich werden zudem die in Abschnitt 2.3.1 vorgestellten List-Scheduling-Verfahren HCPA und Δ-CTS betrachtet. Die Tasks werden anhand der erzeugten Schedules ausgeführt und die Gesamtlaufzeit gemessen, die als Basis für den Vergleich genutzt wird.

6.3.1 Zielplattform und betrachtete Anwendungen

Für die Messungen wird ein heterogener Rechencluster bestehend aus 3 Rechenknoten mit insgesamt 16 Prozessorkernen verwendet. Die Eigenschaften dieser Rechenknoten sind in Tabelle 6.2 aufgelistet. Der Rechenknoten `sb1` verfügt über die größte Anzahl Prozessorkerne innerhalb des Clusters und wird daher als Referenzrechenknoten für die Bestimmung der Parameter des Kostenmodells (vgl. Anschnitt 3.2.1) verwendet. Ein separater Front-End-Knoten, der mit einem Intel Xeon E5-2683-Prozessor und 128 GB Arbeitsspeicher ausgestattet ist, übernimmt die Ausführung

Rechen-knoten	Prozessor	Architektur	Erscheinungs-jahr	Anzahl Kerne	RAM-Größe	Frequenz in GHz
`skylake1`	Intel i7-6700	Skylake	2015	4	16 GB	3.40
`hw1`	Intel i7-4770K	Haswell	2013	4	16 GB	3.50
`sb1`	Intel Xeon E5-2650	Sandy Bridge	2012	8	32 GB	2.00

Tabelle 6.2: Liste der Rechenknoten des verwendeten heterogenen Rechenclusters.

der Schedulingverfahren. Von diesem Knoten aus werden anschließend die Tasks gemäß der ermittelten Schedules über SSH-Verbindungen auf den jeweiligen Rechenknoten gestartet. Dabei erfolgt der Start eines jeden Tasks erst, nachdem alle seine Vorgänger im Schedule ihre Ausführung beendet haben. Die Messung der Gesamtlaufzeit eines Schedules erfolgt vom Start des ersten Tasks bis zur Beendigung des letzten Tasks. Eventuelle Schwankungen der Gesamtlaufzeit werden berücksichtigt, indem jeder Schedule zehnmal ausgeführt und die Durchschnittszeit angegeben wird. Als parallele Tasks werden für die Laufzeitmessungen jeweils die zwei Anwendungen BARNES und FMM sowie die zwei Kernel CHOLESKY und LU der SPLASH-3-Benchmark-Suite verwendet. Die Tasks werden, sofern nicht anders angegeben, unter Verwendung der Standardparameter oder des Parametersatzes „parsec-simlarge“ gestartet.

6.3.2 Vergleich der Gesamtausführungszeit der ermittelten Schedules

Die vom Verfahren MTASS sowie den List-Scheduling-Verfahren HCPA und Δ-CTS ermittelten Schedules sollen hinsichtlich ihrer Gesamtausführungszeit miteinander verglichen werden. Dazu werden die Verfahren für das Scheduling von jeweils 1 bis 32 SPLASH-3-Benchmark-Tasks auf dem in Tabelle 5.2 gezeigten heterogenen Cluster verwendet und die Gesamtausführungszeit der Tasks gemessen. Weitere Messergebnisse befinden sich im Anhang A.

In Abbildung 6.2 (oben) sind die gemessenen Gesamtlaufzeiten für die Ausführung der BARNES- (links) und der FMM-Anwendungstasks (rechts) in Abhängigkeit von der Taskanzahl dargestellt. Bei beiden Arten von Anwendungstasks führt die Verwendung von SAMT für die meisten Taskanzahlen zu geringeren oder ähnlichen Laufzeiten verglichen mit HCPA und Δ-CTS. Im Durchschnitt sind die Gesamtlaufzeiten beider Anwendungstasks mit Δ-CTS etwa 15% höher und mit HCPA etwa 18% höher. Insbesondere für eine größere Anzahl Tasks erreicht SAMT bessere Ergebnisse als die beiden List-Scheduling-Verfahren. Gegenüber beiden Verfahren reduziert SAMT die Gesamtlaufzeit der BARNES-Tasks um bis zu 32%. Für 6, 12 und 18 BARNES-Tasks liegen die für SAMT gemessenen Laufzeiten jedoch um bis

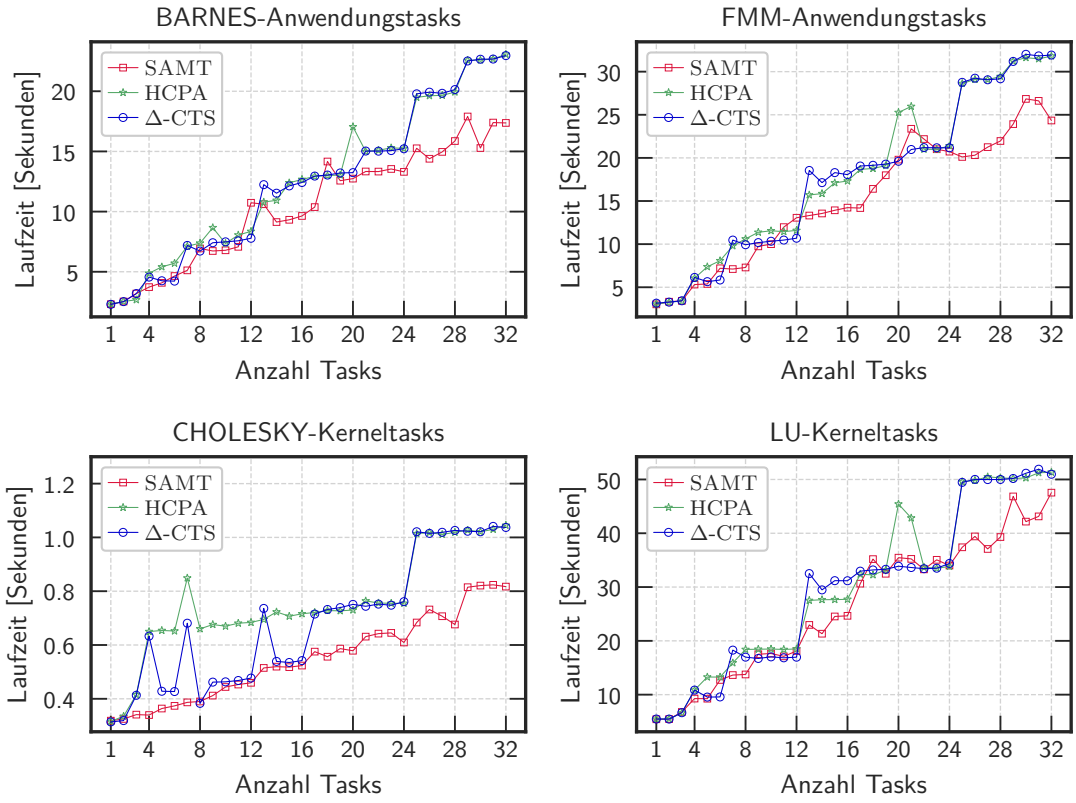


Abbildung 6.2: Oben: Gemessene Gesamtausführungszeit von BARNES- (links) und FMM-Anwendungstasks (rechts) in Abhängigkeit von der Taskanzahl auf dem in Tablle 5.2 beschriebenen heterogenen Cluster. Unten: Gemessene Gesamtausführungszeit von CHOLESKY- (links) und LU-Kerneltasks (rechts) in Abhängigkeit von der Taskanzahl auf dem in Tablle 5.2 beschriebenen heterogenen Cluster.

zu 38% höher als für Δ-CTS und um bis zu 31% höher als für HCPA. Auch für 6, 11, 12, 21, 22 FMM-Tasks führt SAMT zu höheren Gesamtlaufzeiten. Diese sind verglichen mit Δ-CTS bis zu 23% höher und im Vergleich zu HCPA bis zu 14% höher. Die größte von SAMT erreichte Reduktion der Gesamtlaufzeit der FMM-Tasks beträgt 47% gegenüber Δ-CTS und 45% gegenüber HCPA.

Abbildung 6.2 (unten) zeigt die gemessenen Gesamtlaufzeiten für die Ausführung der CHOLESKY- (links) und der LU-Kerneltasks (rechts) in Abhängigkeit von der Anzahl paralleler Tasks. Auch für die Kerneltasks erreicht SAMT im Vergleich zu HCPA und Δ-CTS für die meisten Taskanzahlen geringere oder ähnliche Laufzeiten. Allerdings fällt der Unterschied für die CHOLESKY-Tasks etwas deutlicher aus als für die LU-Tasks. So liegen Gesamtlaufzeiten der CHOLESKY-Tasks im Durchschnitt etwa 24% höher für Δ-CTS und etwa 40% höher für HCPA. Für die LU-Tasks beträgt dieser Unterschied etwa 11% für Δ-CTS und 13% für HCPA.

Auffällig sind die starken Schwankungen bei den gemessenen Laufzeiten für SAMT, die für alle vier Arten von Tasks auftreten. Begründen lässt sich dieses Verhalten mit der zufallsbasierten Suchstrategie, die SAMT durchführt.

6.4 Zusammenfassung

Basierend auf der Metaheuristik Simulated Annealing wurde im Rahmen dieser Arbeit ein neues Schedulingverfahren für parallele Tasks entwickelt. In diesem Kapitel wurde dieses als Simulated Annealing Multiprocessor Task Scheduling (SAMT) bezeichnete Verfahren vorgestellt. Eine Besonderheit dieses Verfahrens ist die zufallsbasierte Vorgehensweise bei der Suche. Zwar kann dadurch die Qualität der gefundenen Lösungen zwischen den Ausführungen variieren, allerdings ist der Suchaufwand geringer. So muss in jedem Schritt von SAMT nur genau ein neuer Schedule erzeugt werden, während andere Verfahren eine Vielzahl möglicher neuer Schedules betrachten. Bestandteil der Beschreibung des Verfahrens ist auch die Wahl der Parameter zum Temperaturverlauf, der Akzeptanz von Lösungen und der Abbruchbedingung, die sowohl die Laufzeit des Verfahrens als auch die Qualität der gefundenen Lösungen beeinflussen. Die Komplexität von SAMT wurde ebenfalls analysiert und ergibt, dass die Laufzeit quadratisch mit der Anzahl der Prozessorkerne des parallelen Systems und linear in Bezug auf die Taskanzahl steigt. In weiteren Untersuchungen wurden die Ausführungszeiten der erzeugten Schedules gemessen. Verglichen mit den List-Scheduling-Verfahren HCPA und Δ-CTS erreichte SAMT für die meisten Taskanzahlen deutlich geringere Laufzeiten. So konnte im Durchschnitt eine Reduktion von bis zu 40% der Laufzeit erzielt werden.

7 Das Water-Level-Search-Verfahren

Mit dem Water-Level-Search-Verfahren wurde im Rahmen dieser Arbeit ein Schedulingverfahren für unabhängige parallele Tasks entwickelt, das eine Kombination aus List-Scheduling und Suchverfahren darstellt. Einen ersten Ansatz für diesen Algorithmus liefert das Water-Level-Verfahren, welches zunächst präsentiert wird. Anschließend wird dann das Water-Level-Search-Verfahren vorgestellt, bevor beide Algorithmen in Experimenten mit den in Abschnitt 2.3.1 beschriebenen List-Scheduling-Verfahren HCPA und Δ-CTS verglichen werden.

7.1 Das Water-Level-Verfahren

Beim List-Scheduling werden die Tasks nach einer bestimmten Ordnung sortiert und anschließend in dieser Reihenfolge jeweils der Rechenressource zugewiesen, die für den jeweiligen Task das bestmögliche Ergebnis liefert. Da die Tasks schrittweise zugewiesen werden, sind die Auswirkungen einer Zuweisung auf den Makespan des resultierenden Schedules erst am Ende des Schedulings sichtbar. Somit ist ein entscheidender Nachteil dieser Vorgehensweise, dass bei der Zuweisung eines Tasks die verbleibenden Tasks nicht berücksichtigt werden. Im Ergebnis führt das häufig zu Schedules mit einem zu großem Makespan.

In [15] wurde das Water-Level-Verfahren vorgestellt, um dem Nachteil gebräuchlicher List-Scheduling-Verfahren entgegenzuwirken. Beim Water-Level-Verfahren handelt es sich zwar auch um ein List-Scheduling-Verfahren, allerdings werden bei der Zuweisung eines Tasks die verbleibenden Tasks mit berücksichtigt. Für die verbleibenden Tasks wird eine optimale Verteilung angenommen, bei der sich diese Tasks wie „Wasser über der Tasklandschaft“ verteilen. Eine solche Verteilung wurde bereits beim Verfahren HP* in Abschnitt 4.4.2 betrachtet. Diese Tasks werden dazu als ein malleable task aufgefasst, der übergreifend die verfügbaren Prozessorkerne aller Rechenknoten nutzt. Die sequentielle Laufzeit dieses malleable tasks ergibt sich aus der Summe der sequentiellen Laufzeiten aller verbleibender Tasks. Die parallele Laufzeit entspricht der sequentiellen Laufzeit geteilt durch die Anzahl verwendeter Prozessorkerne. Mithilfe dessen Endezeit, also der Höhe des „Wasserstands“, im Folgenden als Water-Level bezeichnet, kann bei jeder Zuweisung eines Tasks ein ungefährer Wert für den Makespan des resultierenden Schedules ermittelt werden. Dieser Wert wird im Folgenden als angenommener Makespan bezeichnet. Jeder Task wird dann den Prozessorkernen zugewiesen, für die der angenommene Makespan am geringsten ausfällt.

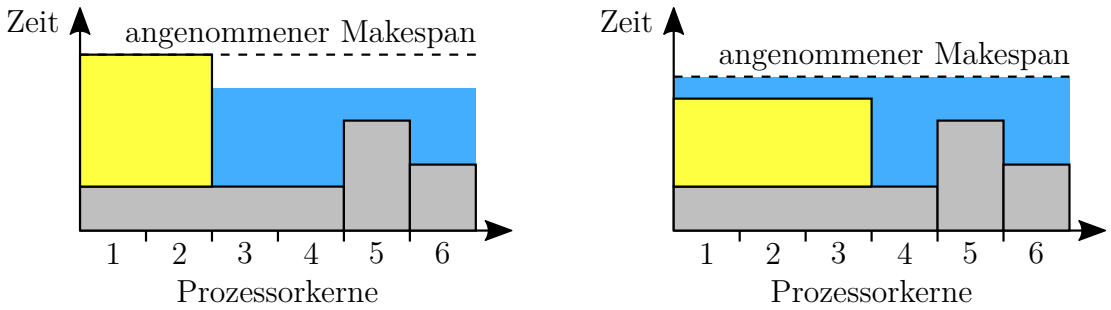


Abbildung 7.1: Beispiel für Zuweisung eines Tasks (gelb) zu einem Rechenknoten mit 6 Prozessorkernen in einem Schritt des Water-Level-Verfahrens. Darstellung der Schedules für eine Zuweisung auf zwei Prozessorkerne (links) und drei Prozessorkerne (rechts) mit bereits zugewiesenen Tasks (grau) und optimaler Verteilung der verbleibenden Tasks (blau).

Beispiel für die Zuweisung eines Tasks mit dem Water-Level-Verfahren

In Abbildung 7.1 ist das Vorgehen in einem Schritt des Water-Level-Verfahrens dargestellt. Zu sehen ist die potentielle Zuweisung des aktuellen Tasks (gelb) zu 2 (links) bzw. 3 (rechts) Prozessorkernen und der jeweils resultierende angenommene Makespan. Die Darstellung beschränkt sich auf diese beiden Fälle, während im Water-Level-Verfahren alle möglichen Anzahlen von Prozessorkernen betrachtet werden. Die bereits zugewiesenen Tasks sind grau und die optimale Verteilung der verbleibenden Tasks blau dargestellt. Der angenommene Makespan wird entweder durch die größte Endezeit der zugewiesenen Tasks (links) oder durch das Water-Level der angenommenen optimalen Verteilung der verbleibenden Tasks (rechts) bestimmt. Im gezeigten Beispiel würde der aktuelle Task (gelb) drei Kernen zugewiesen werden, da der angenommene Makespan dadurch geringer ausfällt.

Algorithmus des Water-Level-Verfahrens

Die zur Beschreibung des Water-Level-Verfahrens verwendete Notation wird in Tabelle 7.1 aufgelistet. Zur Beschreibung des Algorithmus wird der angenommene Makespan m benötigt, die wie folgt berechnet werden kann. Da das Scheduling im Water-Level-Verfahren inkrementell erfolgt, wird in jedem Schritt des Verfahrens ein partieller Schedule S_n betrachtet. Für diesen Schedule S_n lassen sich die verbleibende Rechenzeit R_n und die verfügbare Prozessorzeit $P(S_n)$ (Definitionen 2.1 und 2.2 auf Seite 22) berechnen. Ist die verfügbare Prozessorzeit $P(S_n)$ größer als die verbleibende Rechenzeit R_n, so wird der aktuelle Makespan $M(S_n)$ durch die verbleibenden Tasks nicht erhöht. Übersteigt dagegen die verbleibende Rechenzeit R_n die verfügbare Prozessorzeit $P(S_n)$, dann wird deren Differenz gleichmäßig und

Notation	Bedeutung
$T_1, \dots T_{n_T}$	Menge unabhängiger paralleler Tasks
$N_1, \dots N_{n_N}$	Menge der Rechenknoten der Zielplattform
p_j	Anzahl der Prozessorkerne des Rechenknotens N_j
f_j	Leistungsfaktor des Rechenknotens N_j
K	Gesamtanzahl Prozessorkerne der heterogenen Plattform
R_n	verbleibende Rechenzeit für einen Schedule S_nt
$P(S_n)$	verfügbare Prozessorzeit für einen Schedule S_n
$t_{i,r}(1)$	sequentielle Laufzeit eines Tasks T_i auf dem Referenzrechenknoten N_r
m	angenommener Makespan
*	ermittelte Werte für den geringsten angenommener Makespan

Tabelle 7.1: Verwendete Notation zur Beschreibung des Schedulingproblems (oben) und des Water-Level-Verfahrens (unten)

unter Berücksichtigung des Leistungsfaktors f_j auf alle Prozessorkerne verteilt. Die Berechnung des angenommenen Makespans m für einen Schedule S_n lässt sich somit wie folgt ausdrücken:

$$m = \begin{cases} M(S_n) + {}^{(R_n - P(S_n))}\Big/\sum_{j=1}^{n_N} (p_j \cdot f_j), & \text{falls } R_n > P(S_n) \\ M(S_n), & \text{andernfalls} \end{cases} \tag{7.1}$$

Algorithmus 5 zeigt den Ablauf des Water-Level-Verfahrens. Die Tasks werden zu Beginn absteigend nach ihrer sequentiellen Laufzeit sortiert und anschließend in dieser Reihenfolge bearbeitet. Für jeden Task soll jeweils die Zuweisung zu den Prozessorkernen ermittelt werden, die den geringsten angenommenen Makespan m aufweist. Der aktuell geringste angenommene Makespan m^* wird daher zunächst mit einem möglichst großen Wert initialisiert. Danach wird über alle Rechenknoten $N_j, j \in 1, \dots, n_N$ und deren Anzahl an Prozessorkernen $p = 1, \dots, p_j$ iteriert. In jeder Iteration wird der angenommene Makespan m für eine potentielle Zuweisung des aktuellen Tasks T_i zu p Prozessorkernen des Rechenknotens N_j ermittelt. Falls der ermittelte Wert für m kleiner ist als der bislang beste Wert m^*, dann werden die ermittelten Werte für m, j und p als neue beste Werte gespeichert. Nach der Iteration über alle Rechenknoten wird der Task T_i p^* Kernen des Rechenknotens N_{j^*} zur frühestmöglichen Startzeit zugewiesen, bevor mit dem nächsten Task fortgefahren wird.

Komplexität des Water-Level-Verfahrens

Das Water-Level-Verfahren iteriert über jeden der n_T Tasks und jede Anzahl verfügbarer Prozessorkerne der n_N Rechenknoten. In jeder der Iterationen erfolgt

Algorithmus 5: Das WATER-LEVEL-Verfahren.

Eingabe: • unabhängige parallele Tasks $T_1, \ldots, T_{n_T}$
• Rechenknoten $N_1, \ldots, N_{n_N}$

```
1 Sortiere die Tasks T_i, i = 1,...,n_T absteigend nach t_{i,r}(1)
2 foreach Task T_i der sortierten Menge do
3 |   geringster angenommener Makespan m* = ∞
4 |   foreach Rechenknoten N_j und Anzahl Prozessorkerne p = 1,...,p_j do
5 |   |   Ermittle den angenommenen Makespan m für die Zuweisung von T_i zu
  |   |    p Kernen des Knotens N_j
6 |   |   if (m < m*) then {(m*, j*, p*) = (m, j, p)}
7 |   end
8 |   Weise Task T_i p* Kernen des Rechenknotens N_{j*} zu
9 end
```

die Berechnung des angenommenen Makespans m in konstanter Zeit. Daher steigt die Laufzeit des WATER-LEVEL-Verfahrens linear zur Anzahl an Tasks n_T und zur Gesamtanzahl Prozessorkerne K und kann wie folgt ausgedrückt werden:

$$\mathcal{O}(n_T \cdot K) \tag{7.2}$$

7.2 Das Water-Level-Search-Verfahren

Ein Nachteil des WATER-LEVEL-Verfahrens ist die zu ungenaue Abschätzung der Verteilung der verbleibenden Rechenzeit. Der daraus resultierende angenommene Makespan wird häufig unterschätzt, insbesondere wenn das Laufzeitverhalten der Tasks stark von den Annahmen dieser Abschätzung abweicht. Da für die Abschätzung eine Effizienz der parallelen Tasks von 1 angenommen wird, ist das beispielsweise der Fall für Tasks mit zusätzlichen Kosten, etwa für Synchronisierung und Kommunikation. Das WATER-LEVEL-Verfahren tendiert dann dazu den Tasks zu viele Prozessorkerne zuzuweisen. In derartigen Fällen werden Schedules mit einem zu großen Makespan erzeugt. Das suchbasierte Schedulingverfahren WATER-LEVEL-SEARCH wurde entwickelt, um solche ungenauen Abschätzungen zu vermeiden.

Das Ziel des betrachteten Schedulingproblems ist das Finden eines Schedules mit möglichst geringem Makespan. Per Definition (Gleichung (2.4) auf Seite 16) entspricht der Makespan der größten Endezeit aller Tasks. Somit muss sich der geringste Makespan in der Menge aller potentieller Endezeiten sämtlicher Tasks befinden. Die Idee des WATER-LEVEL-SEARCH-Verfahrens ist daher ein Aufsammeln der aufgetretenen Endezeiten und eine anschließende Suche innerhalb dieser Endezeiten. Gesucht wird dabei die geringste Endezeit, die gleichzeitig ein Makespan eines vollständigen Schedules ist.

Notation	Bedeutung
$T_1, \dots T_{n_T}$	Menge unabhängiger paralleler Tasks
$N_1, \dots N_{n_N}$	Menge der Rechenknoten der Zielplattform
p_j	Anzahl der Prozessorkerne des Rechenknotens N_j
K	Gesamtanzahl Prozessorkerne der heterogenen Plattform
$t_{i,r}(1)$	sequentielle Laufzeit eines Tasks T_i auf dem Referenzrechenknoten N_r
$\hat{m}$	Limit für den Makespan
L	Liste potentieller Limits

Tabelle 7.2: Verwendete Notation zur Beschreibung des Schedulingproblems (oben) und des Water-Level-Search-Verfahrens (unten)

Anstelle einer Abschätzung des Makespans nutzt das Water-Level-Search-Verfahren ein Limit $\hat{m}$ für den Makespan. Die Tasks werden nacheinander derart auf die Prozessorkerne der Rechenknoten verteilt, dass dieses Limit $\hat{m}$ nicht überschritten wird. Dazu wird über die Rechenknoten iteriert und jeweils die Anzahl zu verwendender Prozessorkerne inkrementiert, bis der aktuelle Task zugewiesen werden kann, ohne das Limit $\hat{m}$ zu überschreiten. Ist eine Zuweisung mit dem aktuellen Limit $\hat{m}$ nicht möglich, dann wird $\hat{m}$ erhöht. Danach wird der aktuelle Schedule verworfen und das Scheduling aller Tasks mit dem erhöhten Limit erneut begonnen. Auf diese Weise wird ein Limit gefunden, bei dem alle Tasks zugewiesen werden können. Das gefundene Limit kann allerdings zu groß sein. Daher erfolgt im Anschluss eine binäre Suche in den aufgetretenen Endezeiten der Tasks nach einem kleineren Limit. Ziel des Verfahrens ist es ein möglichst kleines Limit $\hat{m}$ für den Makespan zu finden, das die Zuweisung aller Tasks ermöglicht.

7.2.1 Algorithmus

Der Ablauf des Water-Level-Search-Verfahrens ist in Algorithmus 6 und die verwendete Notation in Tabelle 7.2 zu sehen. Das Verfahren lässt sich in zwei Phasen unterteilen, die nach einer Sortierung der Tasks ausgeführt werden. Die Tasks werden absteigend nach ihrer sequentiellen Laufzeit sortiert und in beiden Phasen in dieser Reihenfolge bearbeitet.

Phase 1 (Zeilen 3-14): Phase 1 wird solange wiederholt bis alle Tasks zugewiesen wurden. Bei jedem Neustart dieser Phase wird die Liste potentieller Limits L geleert und der aktuelle Schedule verworfen. Anschließend wird für jeden Task T_i über alle Rechenknoten und deren Anzahl an Prozessorkernen iteriert. In jeder Iteration wird die früheste Endezeit e_i einer möglichen Zuweisung des aktuellen Tasks T_i zu p Kernen des jeweiligen Rechenknotens ermittelt. Die ermittelte Endezeit e_i wird dann der Liste L hinzugefügt. Die Iteration stoppt sobald e_i kleinergleich dem

aktuellen Limit $\hat{m}$ ist. In diesem Fall wird T_i p Prozessorkernen des Rechenknotens N_j zugewiesen und mit dem nächsten Task fortgefahren (Zeile 10).

Algorithmus 6: Das Water-Level-Search-Verfahren.

Eingabe: unabhängige parallele Tasks $T_1, \ldots, T_{n_T}$
Rechenknoten $N_1, \ldots, N_{n_N}$

```
1  Initialisiere das Limit m̂
2  Sortiere die Tasks absteigend nach ihrer sequentiellen Laufzeit
3  repeat
4      Lösche die Liste potentieller Limits L
5      Verwerfe den aktuellen Schedule
6      foreach Task T_i der sortierten Menge do
7          foreach Rechenknoten N_j und Anzahl Kerne p = 1, ..., p_j do
8              Bestimme die Endezeit e_i einer Zuweisung zu p Kernen des Knotens N_j
9              Füge e_i zur Liste potentieller Limits L hinzu
10             if (e_i ≤ m̂) then Weise T_i p Kernen des Rechenknotens N_j zu und beende
                 die Schleife
11         end
12         if (Task T_i konnte nicht zugewiesen werden) then Setze m̂ gleich der kleinsten
             Endezeit e_i, die für Task T_i ermittelt wurde und beende die Schleife falls
             restart(i) gilt
13     end
14 until alle Tasks wurden zugewiesen
15 repeat
16     Setze Limit m̂ = Median aller potentiellen Limits L
17     Verwerfe den aktuellen Schedule
18     foreach Task T_i der sortierten Menge do
19         foreach Rechenknoten N_j und Anzahl Kerne p = 1, ..., p_j do
20             Bestimme die Endezeit e_i einer Zuweisung zu p Kernen des Knotens N_j
21             if (e_i ≤ m̂) then Weise T_i p Kernen des Rechenknotens N_j zu und beende
                 die Schleife
22         end
23     end
24     if (alle Tasks wurden zugewiesen) then Entferne alle Werte größer m̂ aus L
25     else Entferne alle Werte kleiner gleich m̂ aus L
26 until |L| ≤ 1
```

Falls das Limit $\hat{m}$ für jede potentielle Endezeit überschritten wird, wird $\hat{m}$ auf die kleinste dieser Endezeiten gesetzt. Anschließend wird das Scheduling mit dem neuen Limit $\hat{m}$ neu gestartet (Zeile 12). Die Neustarts werden mithilfe der Funktion `restart()` jeweils nach $n_T/2, 3n_T/4, 7n_T/8, \ldots$ Tasks durchgeführt, um die Menge an Neustarts auf $\mathcal{O}(log n_T)$ zu begrenzen. Auf diese Weise wird das Limit $\hat{m}$ sukzessive erhöht, sodass letztendlich alle Tasks zugewiesen werden können.

Phase 2 (Zeilen 15-26): Das gefundene Limit $\hat{m}$ kann aufgrund der Vorgehensweise zu groß gewählt sein. Daher erfolgt in der zweiten Phase eine Suche nach einem möglichst kleinen Wert für $\hat{m}$. Als Kandidaten für $\hat{m}$ werden die Endezeiten betrachtet, die in der ersten Phase in der Liste L gesammelt wurden. Darunter befindet sich auch die größte Endezeit aller Tasks, genauer gesagt der Makespan des in Phase 1 gefundenen Schedules. Damit ist sichergestellt, dass die zweite Phase ein Limit $\hat{m}$ für alle Tasks findet und terminiert. In jedem Durchlauf wird $\hat{m}$ auf den Median der Liste L gesetzt (Zeile 16) und der bisherige Schedule verworfen. Anschließend wird für jeden der sortierten Tasks T_i über alle Rechenknoten und deren Anzahl an Prozessorkernen iteriert. Analog zu Phase 1 wird die früheste Endezeit e_i des aktuellen Tasks T_i für eine mögliche Zuweisung zu p Kernen des jeweiligen Rechenknotens ermittelt. Falls die ermittelte Endezeit e_i kleinergleich dem aktuellen Limit $\hat{m}$ ist, wird T_i p Prozessorkernen des Rechenknotens zugewiesen und mit dem nächsten Task fortgefahren (Zeile 21). Konnten am Ende eines Durchlaufs alle Tasks zugewiesen werden, dann wurde ein kleinerer Wert für $\hat{m}$ gefunden. In diesem Fall können alle Werte, die größer als $\hat{m}$ sind aus der Liste L entfernt werden. Andernfalls war der Wert für $\hat{m}$ zu niedrig gewählt und es werden alle Werte kleiner gleich $\hat{m}$ aus L entfernt. In beiden Fällen wird anschließend die zweite Phase mit der verkleinerten Liste L wiederholt falls L mehr als einen Wert enthält. Am Schluss enthält L nur noch das kleinste gefundene Limit $\hat{m}$. Der durch das Water-Level-Search-Verfahren ermittelte Schedule ergibt sich aus den gemachten Zuweisungen der Tasks im letzten Durchlauf.

7.2.2 Analyse der Komplexität

In beiden Phasen des Water-Level-Search-Verfahrens wird wie bereits beim Water-Level-Verfahren über jeden der n_T Tasks und jede Anzahl verfügbarer Prozessorkerne der n_N Rechenknoten iteriert. Die Berechnungen in jeder Iteration lassen sich in $\mathcal{O}(1)$ durchführen. Die Laufzeit für einen Durchlauf jeder Phase kann somit durch

$$\mathcal{O}(n_T \cdot K) \tag{7.3}$$

beschrieben werden, wobei n_T der Anzahl paralleler Tasks und K der Gesamtanzahl verfügbarer Prozessorkerne entspricht. Phase 1 wird wiederholt bis alle Tasks zugewiesen wurden. Die Anzahl Wiederholungen ist jedoch auf $\mathcal{O}(log n_T)$ begrenzt. Phase 2 wird wiederholt bis die Liste L genau 1 Element besitzt. Die maximale Anzahl an Endezeiten, die in Phase 1 zu L hinzugefügt werden, entspricht $n_T \cdot K$. In jeder Wiederholung wird der Median von L gewählt und es werden alle kleineren oder größeren Werte entfernt. Dadurch verhält sich die Anzahl Wiederholungen logarithmisch zur maximalen Länge von L.

Rechen-knoten	Prozessor	Architektur	Erscheinungs-jahr	Anzahl Kerne	RAM-Größe	Frequenz in GHz
`sb1`	Intel Xeon E5-2650	Sandy Bridge	2012	16	32 GB	2.00
`ws1,...,ws5`	Intel Xeon X5650	Westmere	2010	5×12	12 GB	2.66
`cs1, cs2`	Intel Xeon E5345	Clovertown	2007	2×8	8 GB	2.33

Tabelle 7.3: Liste der Rechenknoten des verwendeten heterogenen Rechenclusters.

Für ein Schedulingproblem mit n_T parallelen Tasks und einer Gesamtanzahl von K verfügbaren Prozessorkernen ergibt sich folgende Laufzeitkomplexität für beide Phasen und somit für das Water-Level-Search-Verfahren insgesamt:

$$\mathcal{O}((\log n_T) \cdot n_T \cdot K + \log(n_T \cdot K) \cdot n_T \cdot K) \tag{7.4}$$

7.3 Experimentelle Auswertung

In diesem Abschnitt werden die Schedulingverfahren Water-Level (WaterL) und Water-Level-Search (WLS) mit den in Abschnitt 2.3.1 vorgestellten List-Scheduling-Verfahren HCPA und Δ-CTS verglichen. Als Basis für diesen Vergleich dienen Laufzeitmessungen mit Benchmark-Tasks sowie FEM-Simulationstasks. Dazu werden die Tasks anhand der erzeugten Schedules auf einem heterogenen Cluster ausgeführt und die Gesamtlaufzeit gemessen. Des Weiteren wird untersucht, in welchem Maß die Laufzeit sinkt, wenn ein bestehender Cluster um zusätzliche Rechenknoten erweitert wird.

7.3.1 Zielplattform und betrachtete Anwendungen

Der verwendete heterogene Rechencluster besteht aus 8 Rechenknoten mit insgesamt 92 Prozessorkernen. In Tabelle 7.3 sind die Eigenschaften dieser Rechenknoten aufgelistet. Als Referenzrechenknoten und für die Bestimmung der Parameter des Kostenmodells (vgl. Anschnitt 3.2.1) wird der Rechenknoten `cs1` verwendet. Für die Ausführung der Schedulingverfahren wird ein separater Front-End-Knoten genutzt, der mit einem Intel Xeon E5-2683-Prozessor und 128 GB Arbeitsspeicher ausgestattet ist. Dieser Rechenknoten ist ebenfalls für das Starten der Taskausführung anhand der ermittelten Schedules verantwortlich. Dazu werden SSH-Verbindungen zu den Rechenknoten aufgebaut und jeder Task gestartet, sobald alle seine Vorgänger im Schedule ihre Ausführung beendet haben. Während der Ausführung eines Schedules wird die Gesamtlaufzeit vom Start des ersten Tasks bis zur Beendigung des letzten Tasks gemessen. Um eventuelle Schwankungen der Laufzeit zu berücksichtigen, wird jeder Schedule fünfmal ausgeführt und die Durchschnittszeit angegeben. Als

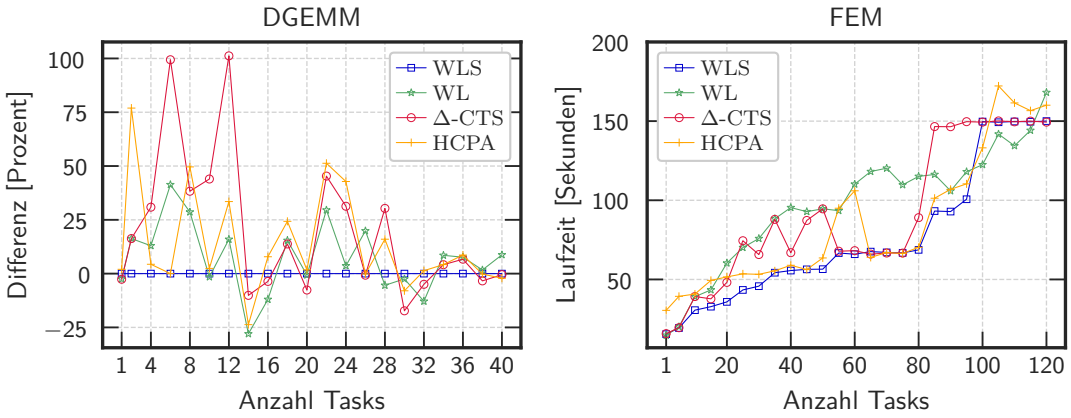


Abbildung 7.2: Links: Prozentualer Unterschied der gemessenen Gesamtlaufzeit der DGEMM-Tasks im Vergleich zu WLS in Abhängigkeit von der Taskanzahl auf den Rechenknoten `cs1` und `ws1`. Rechts: Gemessene Gesamtausführungszeit der FEM-Simulationstasks in Abhängigkeit von der Taskanzahl auf dem in Tablle 4.2 beschriebenen heterogenen Cluster.

parallele Tasks werden die in Abschnitt 3.1.1 beschriebenen FEM-Simulationstasks und die DGEMM-Anwendung aus der OpenBLAS-Bibliothek [54] verwendet. Bei DGEMM handelt es sich um eine Benchmark-Anwendung der numerischen linearen Algebra, die eine Multiplikation zweier Matrizen der Größe 4000×4000 durchführt.

7.3.2 Vergleich der Gesamtausführungszeit der erzeugten Schedules

Im Folgenden soll WLS hinsichtlich der Gesamtausführungszeit der ermittelten Schedules und des erreichten Speedups untersucht werden. Für einen Vergleich der Ergebnisse werden WaterL sowie die List-Scheduling-Verfahren HCPA und Δ-CTS betrachtet. Die vier genannten Verfahren werden zum einen verwendet, um jeweils 1 bis 40 DGEMM-Tasks auf den beiden Rechenknoten `cs1` und `ws1` mit insgesamt 20 Prozessorkernen auszuführen. Zum anderen sollen die Verfahren eingesetzt werden, um jeweils 1 bis 120 FEM-Simulationstasks auf dem kompletten in Tabelle 4.2 gezeigten heterogenen Cluster auszuführen. Weitere Messergebnisse befinden sich im Anhang A.

Abbildung 7.2 (links) zeigt die Ergebnisse für die Ausführung der DGEMM-Tasks auf den Rechenknoten `cs1` und `ws1` anhand der jeweils ermittelten Schedules an. Dargestellt ist die prozentuale Abweichung der von WLS erreichten Gesamtlaufzeit in Abhängigkeit von der Taskanzahl. Insbesondere für kleine Taskanzahlen führen

die übrigen Verfahren zu deutlich höheren Gesamtlaufzeiten im Vergleich zu WLS. Diese sind verglichen mit WLS bei einer Verwendung von Δ-CTS bis zu doppelt so hoch, unter HCPA bis zu 77% höher und mit WaterL bis zu 41% höher. Für einige wenige Taskanzahlen werden die von WLS erreichten Gesamtlaufzeiten von den übrigen Verfahren unterboten. So liegt die größte Verbesserung der Ergebnisse für WaterL bei 28%, für HCPA bei 24% und für Δ-CTS bei 17%. Dennoch wird die Mehrheit der kleinsten Gesamtlaufzeiten von WLS erreicht. Im Durchschnitt unterbieten diese Laufzeiten die Ergebnisse von WaterL um 7%, von HCPA um 14% und von Δ-CTS um 20%.

Abbildung 7.2 (rechts) zeigt die gemessene Gesamtlaufzeit der FEM-Simulationstasks in Abhängigkeit von der Taskanzahl auf dem in Tablle 4.2 beschriebenen heterogenen Cluster. Für die überwiegende Mehrheit der Taskanzahlen werden mit WLS die geringsten Gesamtlaufzeiten der betrachteten Verfahren erzielt. WaterL führt in Bezug auf WLS im Durchschnitt zu 38% höheren Laufzeiten. Begründen lässt sich dies mit dem Laufzeitverhalten der einzelnen FEM-Tasks. Diese skalieren deutlich schlechter als die durch WaterL angenommene optimale Verteilung der verbleibenden Tasks. Dadurch wird der angenommene Makespan zu stark unterschätzt, was zu den erhöhten Gesamtlaufzeiten führt. Die Ergebnisse von HCPA und Δ-CTS liegen mit durchschnittlich 22% und 24% etwas näher an für WLS gemessenen Laufzeiten, schwanken jedoch deutlich stärker. Auf diese Weise treten für HCPA bis zu 61% und für Δ-CTS bis zu 72% höhere Laufzeiten verglichen mit WLS auf. Auffällig sind auch die starken Anstiege der Laufzeiten, die für Δ-CTS zwischen 75 und 85 Tasks und für WLS bei 100 Tasks beobachtet werden können. Diese Anstiege treten auf, wenn viele FEM-Tasks gleichzeitig auf einem einzelnen Rechenknoten ausgeführt werden. Ein Grund dafür könnte die begrenzte Speicherbandbreite der Rechenknoten `cs1` und `cs2` sein, auf denen sich die Laufzeit der einzelnen FEM-Tasks in solchen Fällen fast verdoppelt. Für WaterL und HCPA sind diese Effekte geringer, da beide Verfahren zu einer Ausführung mit weniger Tasks pro Rechenknoten tendieren. Weitere Untersuchungen und Lösungsansätze zu dieser Thematik werden in [17] beschrieben.

7.3.3 Vergleich des Speedups der erzeugten Schedules

In Abbildung 7.3 ist der parallele Speedup für die Ausführung von 16 (links) und 92 (rechts) FEM-Simulationstasks in Abhängigkeit von der Gesamtanzahl verfügbarer Prozessorkerne zu sehen. Der Speedup T_s/T_p gibt den relativen Geschwindigkeitsgewinn der parallelen Ausführung T_p auf p Prozessorkernen gegenüber der sequentiellen Ausführung T_s auf einem Prozessorkern an. Die sequentielle Laufzeit T_s entspricht der Ausführung aller FEM-Tasks auf einem Prozessorkern des Rechenknotens `sb1`, da dort die kürzesten Laufzeiten der FEM-Tasks gemessen wurden. Die parallele Laufzeit T_p entspricht der Laufzeit der Ausführung aller FEM-Tasks anhand eines

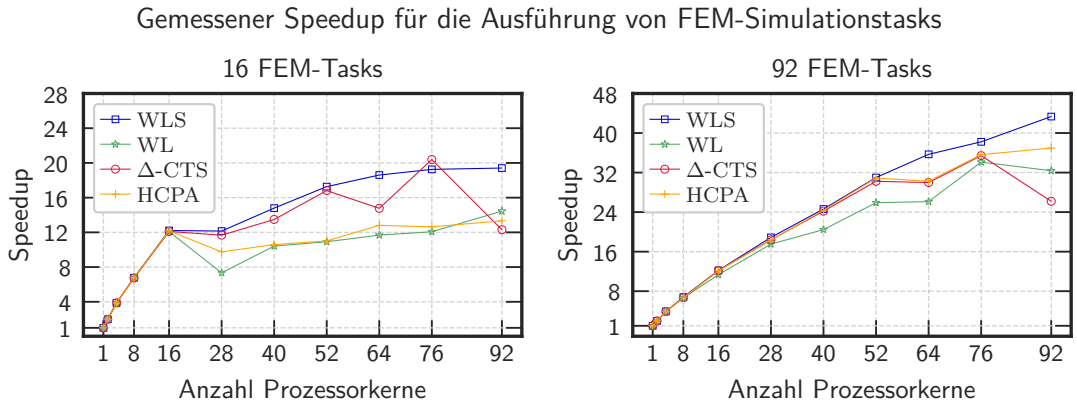


Abbildung 7.3: Paralleler Speedup für die Ausführung von 16 (links) und 92 (rechts) FEM-Simulationstasks in Abhängigkeit von der Gesamtanzahl verfügbarer Prozessorkerne.

ermittelten Schedules auf einer Menge von Rechenknoten, die insgesamt über p Prozessorkerne verfügen. Zur Steigerung der Kernanzahl werden die Rechenknoten in absteigender Reihenfolge ihrer Leistungsfaktoren in die Ausführung einbezogen. Begonnen wird für 1 bis 16 Prozessorkerne mit dem Rechenknoten `sb1`. Für 28 bis 76 Kerne werden zusätzlich die Rechenknoten `ws1` bis `ws5` betrachtet. Die Messung für 92 Prozessorkerne findet auf allen in Tabelle 7.3 genannten Rechenknoten statt.

Abbildung 7.3 (links) zeigt den parallelen Speedup für die Ausführung von 16 FEM-Tasks in Abhängigkeit von der Anzahl verfügbarer Prozessorkerne unter Verwendung der verschiedenen Schedulingverfahren. Für bis zu 16 Prozessorkerne gleichen sich die Ergebnisse, da alle Verfahren die Tasks gleichmäßig den Kernen des Rechenknotens `sb1` zuweisen. Der von HCPA und WaterL erreichte Speedup nimmt für 28 Kerne ab und steigt mit der anschließenden Verwendung weiterer Kerne nur langsam an. Beide Verfahren weisen den einzelnen Tasks jeweils zu viele Kerne zu, wodurch die Tasklaufzeit nur unwesentlich abnimmt und der Speedup nur langsam ansteigt. Das Verfahren Δ-CTS weist dagegen die Tasks gleichmäßiger den Prozessorkernen zu und erreicht so höhere Speedup-Werte. Allerdings wird dabei die Heterogenität des Clusters nicht ausreichend berücksichtigt, wodurch in einigen wenigen Fällen ein sinkender Speedup auftritt. Dies ist insbesondere für 92 Kerne der Fall, da dort die älteren und leistungsschwächeren Rechenknoten `cs1` und `cs2` einbezogen werden. Im Gegensatz zu den anderen Verfahren steigt mit der Verwendung von WLS der Speedup kontinuierlich an.

In Abbildung 7.3 (rechts) ist der parallele Speedup für die Ausführung von 92 FEM-Tasks in Abhängigkeit von der Anzahl verfügbarer Prozessorkerne unter Verwendung der verschiedenen Schedulingverfahren dargestellt. Der Vergleich zu den Ergebnissen für die Ausführung von 16 FEM-Tasks zeigt, dass alle Verfahren einen

ähnlichen und deutlich stärkeren Anstieg des Speedups für mehr als 16 Prozessorkerne erreichen. Diese Ergebnisse sind nicht überraschend, da weniger Kerne als Tasks zur Verfügung stehen. In solchen Fällen werden von allen Verfahren so viele Tasks wie möglich parallel zueinander ausgeführt und zusätzliche Kerne werden für weitere Tasks verwendet. Erst mit steigender Kernanzahl werden die Unterschiede zwischen den Verfahren deutlich. Während der Speedup für WLS kontinuierlich ansteigt, fällt insbesondere für Δ-CTS der Speedup für 92 Kerne wie auch zuvor für 16 Tasks signifikant.

7.3.4 Zusammenfassung

Das in diesem Kapitel vorgestellte Verfahren Water-Level-Search (WLS) wurde im Rahmen dieser Arbeit entwickelt. Es beschreibt eine Kombination aus List-Scheduling-Verfahren und binärer Suche. Auf diese Weise benötigt WLS eine für Suchverfahren besonders geringe Laufzeit, wie auch eine Analyse der Laufzeitkomplexität ergibt. Für die von WLS erzeugten Schedules wurde die Gesamtausführungszeit gemessen und der parallele Speedup ermittelt. Die Ergebnisse wurden mit den beiden List-Scheduling-Verfahren HCPA und Δ-CTS verglichen. Dabei erreichte WLS eine Reduktion der Ausführungszeit von durchschnittlich bis zu 20% für eine parallele Multiplikation zweier Matrizen und von bis zu 72% für die Tasks einer Bauteilsimulation. Zudem zeigte sich ein weniger sprunghafter Anstieg der Laufzeiten in Abhängigkeit von der Taskanzahl. Auch beim Speedup führte Water-Level-Search zu einem kontinuierlicheren Anstieg des Speedups und insgesamt höheren Speedup-Werten.

8 Zusammenfassung

Die vorliegende Dissertation beschäftigt sich mit dem Scheduling unabhängiger paralleler Tasks. Wie in der Einleitung erwähnt, ermöglicht die Verwendung eines gemischt parallelen Programmiermodells in parallelen Anwendungen die Identifikation unabhängiger Tasks, die selbst eine interne Parallelität aufweisen. Diese parallelen Tasks können daher sowohl parallel zueinander als auch jeweils selbst parallel ausgeführt werden, um die Ausführungszeit der Anwendung zu reduzieren. Die Ausführungszeit hängt dabei maßgeblich von der effizienten Ausnutzung der vorhandenen Rechenressourcen ab. Zur zeitlichen und räumlichen Zuordnung der Tasks zu den Prozessoren des parallelen Systems werden Schedulingverfahren eingesetzt. Ein zentrales Problem bei der Bestimmung eines Schedules für parallele Tasks besteht darin, dass das Scheduling bereits für Single-Prozessor-Tasks und parallele Systeme mit zwei Prozessoren NP-schwer ist. Das Scheduling unabhängiger paralleler Tasks ist dahingegen komplexer, als dass zusätzlich für jeden Task die Anzahl zu verwendender Prozessoren bestimmt werden muss. Für die Berechnung eines Schedules werden daher zumeist Approximationsalgorithmen und Heuristiken eingesetzt.

Das Ziel dieser Arbeit ist es, Suchverfahren für das Scheduling unabhängiger paralleler Tasks zu verwenden. Dazu wurden vier Schedulingverfahren entwickelt, die auf speziellen Suchstrategien, unter anderem aus dem Bereich der Künstlichen Intelligenz, basieren. Die Verfahren wurden in Laufzeitmessungen sowohl untereinander als auch mit existierenden List-Scheduling-Heuristiken verglichen. Die Messungen wurden mit Anwendungen aus dem naturwissenschaftlichen Bereich und dem Maschinenbau durchgeführt. So wurden parallele Berechnungen der SPLASH-3-Benchmark-Suite sowie eine praxisnahe Simulationsanwendung zur Bauteilbelastung als parallele Tasks verwendet. In den Untersuchungen wurden die parallelen Tasks anhand der erzeugten Schedules ausgeführt und deren Gesamtlaufzeiten gemessen. Neben der Ausführungszeit der Schedules ist die Laufzeit der Verfahren selbst ein wichtiges Kriterium für Schedulingverfahren. Daher wurden die Verfahren auch hinsichtlich der Dauer des Schedulingvorgangs untersucht.

Da die exakten Tasklaufzeiten in praktischen Einsatzgebieten oft nicht zur Verfügung stehen, basieren die von Schedulingverfahren getroffenen Entscheidungen zumeist auf Kostenmodellen. In der vorliegenden Arbeit werden ebenfalls modellierte Tasklaufzeiten verwendet. Zu diesem Zweck wird eine Modellierung der Kosten für parallele Tasks auf heterogenen parallelen Systemen vorgestellt. Das präsentierte Kostenmodell entspricht einer parametrisierten Laufzeitformel, die das Laufzeitverhalten gewöhnlicher paralleler Anwendungen abbildet. Dazu setzt sich die Laufzeit-

formel aus einem parallelen und einem sequentiellen Anteil sowie einem logarithmischen Anteil für den Overhead der parallelen Ausführung zusammen. Die Genauigkeit des Kostenmodells wurde für parallele Tasks der SPLASH-3-Benchmark-Suite untersucht. Die Ergebnisse zeigen für alle Tasks eine parallele Laufzeit, die mit steigender Threadanzahl abnimmt. Dieses Laufzeitverhalten der Tasks wird durch die gewählte Laufzeitformel sehr gut abgebildet. Größere Abweichungen traten lediglich für einzelne Threadanzahlen bestimmter Anwendungstasks auf und sind auf eine ungünstige Aufteilung der Daten insbesondere bei Gitter-basierten Berechnungen zurückzuführen.

Da parallele Tasks in der Literatur unter verschiedenen Begriffen verwendet werden, wurde eine Klassifizierung der verschiedenen Arten von parallelen Tasks durchgeführt und das in der vorliegenden Arbeit betrachtete Schedulingproblem eingeordnet. Neben aktuellen Schedulingverfahren aus Literatur werden zwei Ansätze zur Konstruktion von Schedules für parallele Tasks vorgestellt. Dies ist zum einen der inkrementelle Aufbau eines Schedules, bei dem mit einem leeren Schedule begonnen wird und die Tasks schrittweise zugewiesen werden, bis schlussendlich ein vollständiger Schedule konstruiert worden ist. Eine andere Möglichkeit ist die Modifikation eines bestehenden Schedules, wobei schrittweise einzelne Tasks eines Ausgangsschedules neu zugewiesen werden, bis ein Abbruchkriterium erreicht wird. Mithilfe derartiger Modifikationen lassen sich Lösungsräume konstruieren, in denen nach einer optimalen Lösung gesucht werden kann. Für jeden der beiden Ansätze wurden zwei spezielle Suchverfahren repräsentativ ausgewählt und in Schedulingverfahren umgesetzt. HP* und Water-Level-Search lassen sich der Klasse der inkrementellen suchbasierten Schedulingverfahren zuordnen, wohingegen MTaSS und SAMT den modifizierenden suchbasierten Schedulingverfahren angehören.

Mit dem Verfahren Heterogeneous Parallel task scheduling based on A* (HP*) wurde ein Schedulingverfahren basierend auf der A*-Suche entwickelt. HP* weist ausgehend von einem leeren Schedule die Tasks nacheinander den Prozessorkernen der Rechenknoten zu. Für jede Zuweisung wird dabei jede Reihenfolge der Tasks und jede mögliche Kombination von Prozessorkernen betrachtet. Es wurde gezeigt, dass HP* auf diese Weise eine optimale Lösung finden kann. Des Weiteren wurden eine Restkostenschätzung zur effizienteren Suche entworfen und drei Strategien zur Eingrenzung des Suchraums präsentiert. Durch die Kombination dieser drei Strategien konnten die Anzahl von HP* zu verarbeitenden Schedules um bis zu 99,9% reduziert werden. In der Folge sinken die Laufzeit und der Hauptspeicherbedarf von HP* deutlich und gleichzeitig wird das Lösen größerer Schedulingprobleme ermöglicht. Die Resultate zeigen zudem, dass die Verwendung der Restkostenschätzung zu einer weiteren Reduktion der zu verarbeitenden Schedules beiträgt.

Die Schedulingverfahren Multiprocessor Task Tabu Search Scheduling (MTaSS) und Simulated Annealing Multiprocessor Task Scheduling (SAMT) beruhen auf den Metaheuristiken Tabu-Suche und der Methode der si-

mulierten Abkühlung (Simulated Annealing). Beide Verfahren transformieren einen Anfangsschedule schrittweise, um einen Schedule mit geringerem Makespan zu finden. Das Verfahren MTaSS führt in jedem Schritt die Transformation durch, die den geringsten Makespan zur Folge hat. Kreisläufe und ein Feststecken in lokalen Optima werden vermieden, indem besuchte Schedules in einer Tabu-Liste gespeichert und so von der Suche ausgeschlossen werden. Zudem wurde eine Strategie entworfen, mit der die Laufzeit von MTaSS um etwa ein Drittel gesenkt werden kann. In jedem Durchlauf des Verfahrens SAMT wird eine zufällige Transformation am aktuellen Schedule durchgeführt und ein resultierender Schedule mit geringerem Makespan als neue Lösung akzeptiert. Schlechtere Lösungen werden mit einer Wahrscheinlichkeit ebenfalls akzeptiert, die zusammen mit der Temperatur in jedem Durchlauf abnimmt. Im Vergleich zu den List-Scheduling-Heuristiken konnte MTaSS die Gesamtausführungszeit der Tasks um bis zu 40% und SAMT um bis zu 47% reduzieren.

Als viertes Schedulingverfahren wurde das Water-Level-Search-Verfahren entwickelt. Zunächst werden in wiederholten List-Scheduling-Vorgängen die aufgetretenen Endezeiten der Tasks aufgesammelt. Anschließend wird eine binäre Suche innerhalb dieser Endezeiten durchgeführt. Gesucht wird die geringste der Endezeiten, die das Scheduling aller Tasks ermöglicht. Verglichen mit den List-Scheduling-Heuristiken erreicht Water-Level-Search nicht nur eine geringere Gesamtausführungszeit der Tasks sondern auch höhere Werte beim parallelen Speedup.

Die Optimalität von HP* spiegelt sich auch in den Laufzeitmessungen wider. Sowohl für die Tasks der SPLASH-3-Benchmark-Suite als auch für die FEM-Tasks der Simulationsanwendung erreichte HP* die geringsten Ausführungszeiten der Schedules. So liegen die gemessenen Ausführungszeiten der erzeugten Schedules bis zu 20% unter den von SAMT erzielten Ergebnissen und bis zu 5% unter den von MTaSS und Water-Level-Search erreichten Werten. Allerdings ist das Scheduling mit HP* vergleichsweise zeitintensiv und aufgrund des sehr hohen Speicherbedarfs ist es nur für kleinere und mittelgroße Schedulingprobleme geeignet. Von den verbleibenden Verfahren erzeugte MTaSS die Schedules mit den im Durchschnitt geringsten Laufzeiten. So wurden für SAMT im Durchschnitt 46% höhere Laufzeiten und für Water-Level-Search durchschnittlich 19% höhere Laufzeiten gemessen. Für das Verfahren SAMT ergaben sich mit steigender Taskanzahl zudem stark schwankende Laufzeiten, während die übrigen drei Verfahren einen gleichmäßigen Anstieg zeigten. Diese Ergebnisse lassen sich auf die verwendete Suchstrategie zurückführen, die stark vom Zufall abhängt. Dennoch konnten alle vier suchbasierten Verfahren mit den erzeugten Schedules die Gesamtausführungszeiten der Tasks im Vergleich zu existierenden List-Scheduling-Heuristiken zum Teil deutlich reduzieren.

In Bezug auf die Laufzeit der Schedulingverfahren erzielte Water-Level-Search die geringsten Werte der suchbasierten Verfahren, gefolgt von MTaSS, SAMT und HP*. Eine intensivere Suche und die damit einhergehende längere Dauer des Sche-

dulings kann sich dennoch lohnen, wenn dieser Mehraufwand die Reduktion der Laufzeit des Schedules nicht übersteigt. Die Ergebnisse zeigen, dass MTaSS zwar eine längere Zeit für das Scheduling benötigt als Water-Level-Search, diese Zeit aber häufig geringer ist als der Laufzeitunterschied der erzeugten Schedules. Für parallele Tasks mit kürzeren Laufzeiten fallen die Laufzeitunterschiede bei den erzeugten Schedules geringer aus. Dennoch konnte gezeigt werden, dass sich auch in diesen Fällen der Einsatz der zeitaufwendigeren suchbasierten Schedulingverfahren lohnt.

Abschließend bleibt festzuhalten, dass alle vier Schedulingverfahren im Vergleich zu existierenden List-Scheduling-Heuristiken eine signifikante Reduktion der Ausführungszeit erreichen können. Diese Ergebnisse sind unabhängig davon, ob eine inkrementelle oder eine modifizierende Konstruktion von Schedules verwendet wird. Daran zeigt sich, dass das komplexe Scheduling paralleler Tasks vom Einsatz von Suchverfahren profitiert, wie sie auch in modernen KI-Anwendungen zum Einsatz kommen.

A Ergebnisse weiterer Laufzeitmessungen

A.1 Zielplattform und Parallele Anwendungsprogramme

Für die Laufzeitmessungen wurde ein Rechencluster bestehend aus 4 Rechenknoten mit insgesamt 32 Prozessorkernen verwendet. Die Rechenknoten sowie deren Eigenschaften sind in Tabelle A.1 aufgelistet. Innerhalb des Clusters verfügt der Rechenknoten `matisse1` über die größte Anzahl an Prozessorkernen. Daher wurde dieser Knoten als Referenzrechenknoten für die Parameterbestimmung des Kostenmodells gewählt. Die Schedulingverfahren selbst werden auf einem separaten Front-End-Knoten ausgeführt, der über einen Intel Xeon E5-2683-Prozessor und 128 GB Arbeitsspeicher verfügt. Dieser Knoten ist jeweils direkt mit den Knoten des Clusters verbunden und ist daher auch für das Starten der Tasks auf den zugewiesenen Rechenknoten verantwortlich. Die Ausführung eines jeden Tasks wird dabei über eine SSH-Verbindung gestartet, sobald dessen Vorgänger im Schedule ihre Ausführung beendet haben. Zur Berücksichtigung von Laufzeitschwankungen wird jeder Schedule zehnmal ausgeführt und die Durchschnittszeit angegeben.

Als parallele Tasks wurde eine Auswahl der in Abschnitt 3.1.1 beschriebenen parallelen Anwendungsprogramme verwendet. Aus der Gruppe der SPLASH-3-Benchmark-Tasks gehören dazu die beiden Anwendungen BARNES und FMM sowie der Kernel LU. Die Ausführung der Tasks erfolgt, sofern nicht anders angegeben, unter Verwendung der Standardparameter oder des Parametersatzes „parsec-simlarge“. Eine weitere Gruppe von parallelen Tasks bilden die FEM-Simulationstasks. Bei diesen Tasks handelt es sich um Anwendungen einer Finite-Elemente-Methode (FEM), die mit OpenMP parallelisiert wurden. Die Anzahl zu verwendender Prozessorkerne wird daher über die Umgebungsvariable `OMP_NUM_THREADS` gesteuert.

A.2 Vergleich der Gesamtausführungszeit der ermittelten Schedules

Rechen-knoten	Prozessor	Architektur	Erscheinungs-jahr	Anzahl Kerne	RAM-Größe	Frequenz in GHz
`skylake1`	Intel i7-6700	Skylake	2015	4	16 GB	3.40
`hw1`	Intel i7-4770K	Haswell	2013	4	16 GB	3.50
`coffeelake1`	Intel i7-9700	Coffee Lake	2020	8	32 GB	3.00
`matisse1`	AMD Ryzen 9 3950X	Matisse	2020	16	64 GB	3.50

Tabelle A.1: Liste der Rechenknoten des verwendeten heterogenen Rechenclusters.

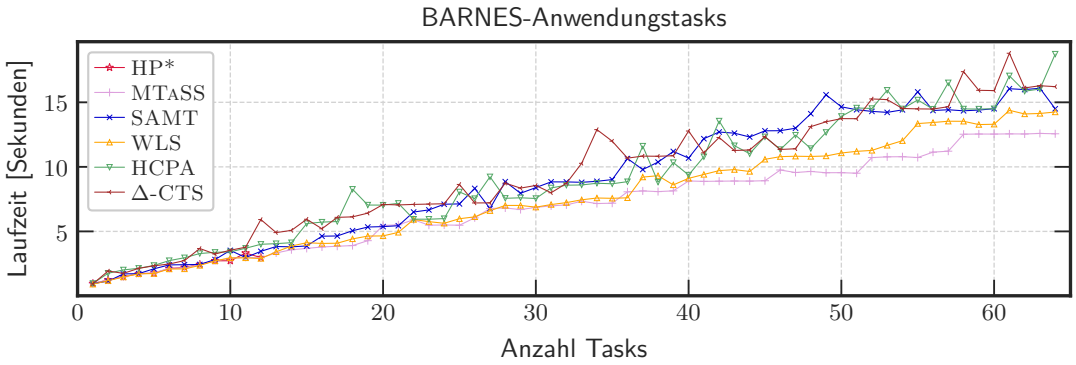


Abbildung A.1: Gemessene Gesamtausführungszeit von BARNES-Anwendungstasks in Abhängigkeit von der Taskanzahl auf dem in Tablle A.1 beschriebenen heterogenen Cluster.

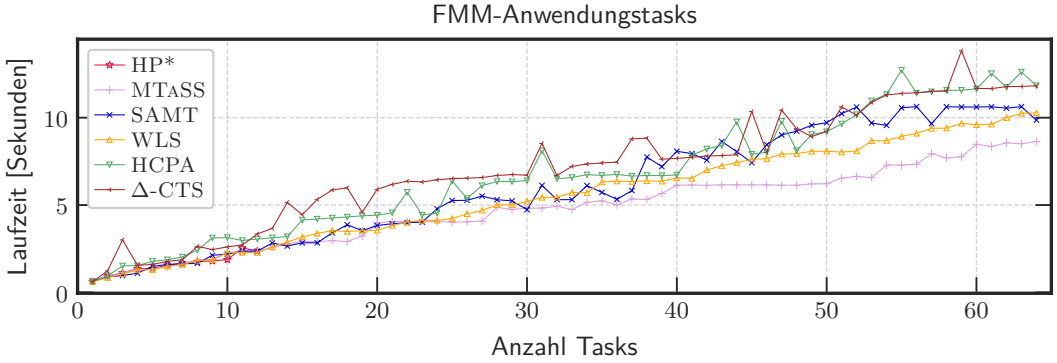


Abbildung A.2: Gemessene Gesamtausführungszeit von FMM-Anwendungstasks in Abhängigkeit von der Taskanzahl auf dem in Tablle A.1 beschriebenen heterogenen Cluster.

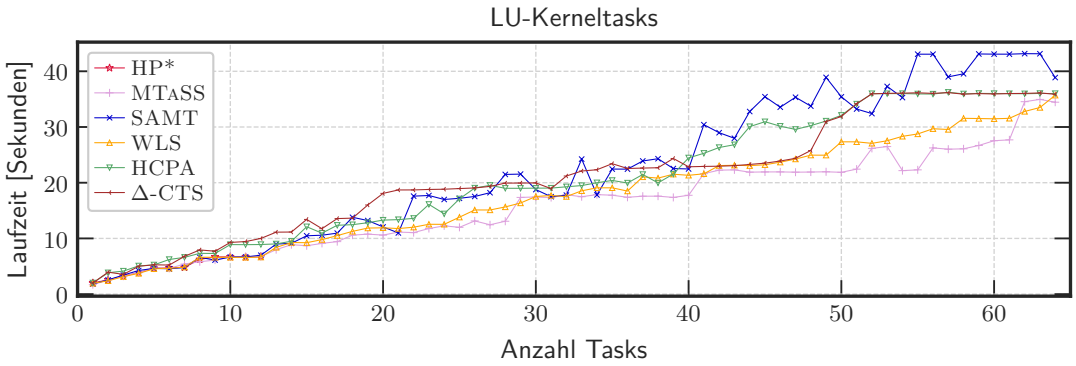


Abbildung A.3: Gemessene Gesamtausführungszeit von LU-Kerneltasks in Abhängigkeit von der Taskanzahl auf dem in Tablle A.1 beschriebenen heterogenen Cluster.

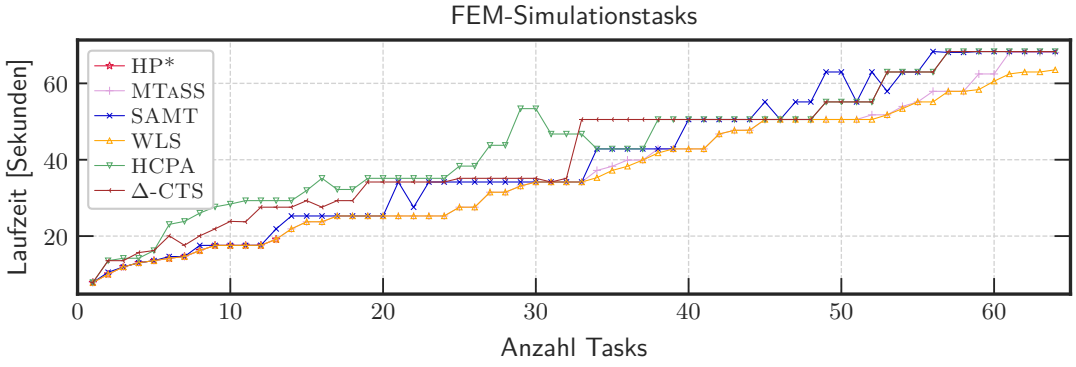


Abbildung A.4: Gemessene Gesamtausführungszeit von FEM-Simulationstasks in Abhängigkeit von der Taskanzahl auf dem in Tablle A.1 beschriebenen heterogenen Cluster.

Literaturverzeichnis

[1] Akbari, M. ; Rashidi, H. ; Alizadeh, S. H.: An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems. In: *Engineering Applications of Artificial Intelligence* 61 (2017), S. 35–46. http://dx.doi.org/10.1016/j.engappai.2017.02.013. – DOI 10.1016/j.engappai.2017.02.013. – ISSN 0952–1976

[2] Benoit, A. ; Fèvre, V. L. ; Perotin, L. ; Raghavan, P. ; Robert, Y. ; Sun, H. : Resilient Scheduling of Moldable Jobs on Failure-Prone Platforms. In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, S. 81–91

[3] Beuchler, S. ; Meyer, A. ; Pester, M. : SPC-PM3AdH v1.0 - Programmer's Manual. In: *Preprint SFB/393 01-08, TU-Chemnitz* (2001)

[4] Biswas, S. K. ; Jagdev, R. ; Muhuri, P. K.: Energy Efficient Scheduling in Multiprocessor Systems Using Archived Multi-objective Simulated Annealing. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018, S. 1–8

[5] Blazewicz ; Drabowski ; Weglarz: Scheduling Multiprocessor Tasks to Minimize Schedule Length. In: *IEEE Transactions on Computers* C-35 (1986), Nr. 5, S. 389–393. http://dx.doi.org/10.1109/TC.1986.1676781. – DOI 10.1109/TC.1986.1676781

[6] Blazewicz, J. ; Kovalyov, M. Y. ; Machowiak, M. ; Trystram, D. ; Weglarz, J. : Preemptable malleable task scheduling problem. In: *IEEE Transactions on Computers* 55 (2006), Nr. 4, S. 486–490. http://dx.doi.org/10.1109/TC.2006.58. – DOI 10.1109/TC.2006.58

[7] Bleuse, R. ; Hunold, S. ; Kedad-Sidhoum, S. ; Monna, F. ; Mounie, G. ; Trystram, D. : Scheduling Independent Moldable Tasks on Multi-Cores with GPUs. In: *IEEE Transactions on Parallel and Distributed Systems* 28 (2017), Nr. 9, S. 2689–2702. http://dx.doi.org/10.1109/TPDS.2017.2675891. – DOI 10.1109/TPDS.2017.2675891

[8] Boveiri, H. R.: ACO-MTS: A new approach for multiprocessor task scheduling based on ant colony optimization. In: *2010 International Conference on Intelligent and Advanced Systems*, 2010, S. 1–5

[9] Bożejko, W. ; Nadybski, P. ; Wodecki, M. : Two level algorithm with Tabu Search optimization for task scheduling problem in computing cluster environment. In: *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*, 2017, S. 238–242

[10] Braun, T. D. ; Siegel, H. J. ; Beck, N. ; Bölöni, L. L. ; Maheswaran, M. ; Reuther, A. I. ; Robertson, J. P. ; Theys, M. D. ; Yao, B. ; Hensgen, D. ; Freund, R. F.: A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. In: *J. Parallel Distrib. Comput.* 61 (2001), Jun., Nr. 6, S. 810–837. http://dx.doi.org/10.1006/jpdc.2000.1714. – DOI 10.1006/jpdc.2000.1714. – ISSN 0743–7315

[11] Corberán, A. ; Fernández, E. ; Laguna, M. ; Martí, R. : Heuristic solutions to the problem of routing school buses with multiple objectives. In: *Journal of the Operational Research Society* 53 (2002), Nr. 4, S. 427–435. http://dx.doi.org/10.1057/palgrave.jors.2601324. – DOI 10.1057/palgrave.jors.2601324

[12] *Kapitel* 17. In: Crainic, T. G. ; Toulouse, M. : *Parallel Strategies for Meta-Heuristics*. Boston, MA : Springer US, 2003. – ISBN 978–0–306–48056–0, S. 475–513

[13] Culler, D. ; Karp, R. ; Patterson, D. ; Sahay, A. ; Schauser, K. E. ; Santos, E. ; Subramonian, R. ; Eicken, T. von: LogP: Towards a Realistic Model of Parallel Computation. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA : Association for Computing Machinery, 1993 (PPOPP '93). – ISBN 0897915895, S. 1–12

[14] Dechter, R. ; Pearl, J. : Generalized Best-first Search Strategies and the Optimality of A*. In: *J. ACM* 32 (1985), Jul., Nr. 3, S. 505–536. – ISSN 0004–5411

[15] Dietze, R. : *Effiziente Ausführung paralleler Simulationstasks auf verteilten Rechenressourcen*, TU Chemnitz, Masterarbeit, 2015

[16] Dietze, R. ; Hofmann, M. ; Rünger, G. : Water-Level scheduling for parallel tasks in compute-intensive application components. In: *The Journal of Supercomputing, Special Issue on Sustainability on Ultrascale Computing Systems and Applications* 72 (2016), Nr. 11, S. 4047–4068. http://dx.doi.org/10.1007/s11227-016-1711-1. – DOI 10.1007/s11227–016–1711–1. – ISSN 1573–0484

[17] Dietze, R. ; Hofmann, M. ; Rünger, G. : Analysis and Modeling of Resource Contention Effects based on Benchmark Applications. In: *International*

Conference on High Performance Computing & Simulation (HPCS 2018), Special Session on High Performance Computing Benchmarking and Optimization (HPBench 2018), IEEE, July 2018. – ISBN 978–1–5386–7879–4, S. 330–337

[18] Dietze, R. ; Rünger, G. : Search-based Scheduling for Parallel Tasks on Heterogenous Platforms. In: *Euro-Par 2019: Parallel Processing Workshops*, Springer, August 2019. – ISBN 978–3–030–48340–1, S. 333–344

[19] Dietze, R. ; Rünger, G. : The Search-based Scheduling Algorithm HP* for Parallel Tasks on Heterogeneous Platforms. In: *Concurrency and Computation: Practice and Experience* (2020). http://dx.doi.org/10.1002/cpe.5898. – DOI 10.1002/cpe.5898

[20] Dowdy, L. W.: On the partitioning of multiprocessor systems / Vanderbilt University. 1988 (88-06). – Forschungsbericht

[21] Drozdowski, M. : *Scheduling for Parallel Processing*. 1st. Springer Publishing Company, Incorporated, 2009. http://dx.doi.org/10.5555/1628958. http://dx.doi.org/10.5555/1628958. – ISBN 1848823096

[22] Du, J. ; Leung, J. Y.-T. : Complexity of Scheduling Parallel Task Systems. In: *SIAM Journal on Discrete Mathematics* 2 (1989), Nr. 4, S. 473–487. http://dx.doi.org/10.1137/0402042. – DOI 10.1137/0402042

[23] Dümmler, J. ; Rauber, T. ; Rünger, G. : Mixed Programming Models using Parallel Tasks. In: Dongarra, J. (Hrsg.) ; Hsu, C.-H. (Hrsg.) ; Li, K.-C. (Hrsg.) ; Yang, L. T. (Hrsg.) ; Zima, H. (Hrsg.): *Handbook of Research on Scalable Computing Technologies*. Information Science Reference, 2009. – ISBN 978–1–60566–661–7, S. 246–275

[24] Dümmler, J. ; Kunis, R. ; Rünger, G. : A Scheduling Toolkit for Multiprocessor-Task Programming with Dependencies. In: Kermarrec, A.-M. (Hrsg.) ; Bougé, L. (Hrsg.) ; Priol, T. (Hrsg.): *Euro-Par 2007 Parallel Processing*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007. – ISBN 978–3–540–74466–5, S. 23–32

[25] Dümmler, J. ; Rauber, T. ; Rünger, G. : Scheduling Support for Communicating Parallel Tasks. In: Rajopadhye, S. (Hrsg.) ; Mills Strout, M. (Hrsg.): *Languages and Compilers for Parallel Computing*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–36036–7, S. 252–267

[26] Dutot, P.-F. ; Mounié, G. ; Trystram, D. : Scheduling Parallel Tasks: Approximation Algorithms. In: Leung, J. T. (Hrsg.): *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004 (chapter 26), S. 26–1 – 26–24

[27] Fortune, S. ; Wyllie, J. : Parallelism in Random Access Machines. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. New York, NY, USA : Association for Computing Machinery, 1978 (STOC '78). – ISBN 9781450374378, S. 114–118

[28] Geiger, M. J.: *Lokale Suchheuristiken*. Wiesbaden : Deutscher Universitätsverlag, 2005. – 47–111 S. http://dx.doi.org/10.1007/978-3-322-82174-4_4. http://dx.doi.org/10.1007/978-3-322-82174-4_4. – ISBN 978–3–322–82174–4

[29] Gemund, A. C. V. ; Radulescu, A. : A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In: *2013 42nd International Conference on Parallel Processing*. Los Alamitos, CA, USA : IEEE Computer Society, sep 2001, S. 69–76

[30] Glover, F. : Future Paths for Integer Programming and Links to Artificial Intelligence. In: *Comput. Oper. Res.* 13 (1986), Mai, Nr. 5, S. 533–549. http://dx.doi.org/10.1016/0305-0548(86)90048-1. – DOI 10.1016/0305–0548(86)90048–1. – ISSN 0305–0548

[31] Glover, F. ; Taillard, E. : A user's guide to tabu search. In: *Annals of Operations Research* 41 (1993), Mar, Nr. 1, S. 1–28. http://dx.doi.org/10.1007/BF02078647. – DOI 10.1007/BF02078647. – ISSN 1572–9338

[32] Glover, F. W. ; Kochenberger, G. A.: *Handbook of Metaheuristics*. Springer, 2003 (International Series in Operations Research & Management Science). – ISBN 978–0–306–48056–0

[33] Gupta, S. ; Kumar, V. ; Agarwal, G. : Task Scheduling in Multiprocessor System Using Genetic Algorithm. In: *2010 Second International Conference on Machine Learning and Computing*, 2010, S. 267–271

[34] Hanti, T. ; Frey, A. ; Hardt, W. : Reconfigurable Multi-Core scheduling for real-time functions in avionic mission systems. In: *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015, S. 7A3–1–7A3–11

[35] Hart, P. E. ; Nilsson, N. J. ; Raphael, B. : A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics* 4 (1968), July, Nr. 2, S. 100–107. – ISSN 0536–1567

[36] Iliopoulou, C. ; Kepaptsoglou, K. ; Vlahogianni, E. : Metaheuristics for the transit route network design problem: a review and comparative analysis. In: *Public Transport* 11 (2019), Oct, Nr. 3, S. 487–521. http://dx.doi.org/10.1007/s12469-019-00211-2. – DOI 10.1007/s12469–019–00211–2. – ISSN 1613–7159

[37] Jansen, K. ; Zhang, H. : Scheduling malleable tasks with precedence constraints. In: *Journal of Computer and System Sciences* 78 (2012), Nr. 1, S. 245–259. http://dx.doi.org/10.1016/j.jcss.2011.04.003. – DOI 10.1016/j.jcss.2011.04.003. – ISSN 0022–0000. – JCSS Knowledge Representation and Reasoning

[38] Jin, S. ; Schiavone, G. ; Turgut, D. : A Performance Study of Multiprocessor Task Scheduling Algorithms. In: *J. Supercomput.* 43 (2008), Jan., Nr. 1, S. 77–97. http://dx.doi.org/10.1007/s11227-007-0139-z. – DOI 10.1007/s11227–007–0139–z. – ISSN 0920–8542

[39] Kafil, M. ; Ahmad, I. : Optimal task assignment in heterogeneous computing systems. In: *Proceedings Sixth Heterogeneous Computing Workshop (HCW'97)*, 1997, S. 135–146

[40] Kalra, M. ; Singh, S. : A review of metaheuristic scheduling techniques in cloud computing. In: *Egyptian Informatics Journal* 16 (2015), Nr. 3, S. 275–295. http://dx.doi.org/https://doi.org/10.1016/j.eij.2015.07.001. – DOI https://doi.org/10.1016/j.eij.2015.07.001. – ISSN 1110–8665

[41] *Kapitel* 1. In: Khamsi, M. A. ; Kirk, W. A.: *Introduction*. John Wiley & Sons, Ltd, 2001. – ISBN 9781118033074, S. 1–11

[42] Kirkpatrick, S. ; Gelatt, C. ; Vecchi, M. : Optimization by Simulated Annealing. Version: 1987. http://dx.doi.org/https://doi.org/10.1016/B978-0-08-051581-6.50059-3. In: Fischler, M. A. (Hrsg.) ; Firschein, O. (Hrsg.): *Readings in Computer Vision*. San Francisco (CA) : Morgan Kaufmann, 1987. – DOI https://doi.org/10.1016/B978–0–08–051581–6.50059–3. – ISBN 978–0–08–051581–6, S. 606–615

[43] Klein, D. ; Manning, C. D.: A* Parsing: Fast Exact Viterbi Parse Selection. In: *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2003, S. 119–126

[44] Kroll, L. (Hrsg.): *Technologiefusion für multifunktionale Leichtbaustrukturen*. Springer Berlin Heidelberg, 2019. – 598–608 S. http://dx.doi.org/10.1007/978-3-662-54734-2. http://dx.doi.org/10.1007/978-3-662-54734-2

[45] Kwok, Y.-K. ; Ahmad, I. : On Multiprocessor Task Scheduling Using Efficient State Space Search Approaches. In: *J. Parallel Distrib. Comput.* 65 (2005), Dez., Nr. 12, S. 1515–1532. – ISSN 0743–7315

[46] Marquardt, D. W.: An Algorithm for Least-Squares Estimation of Nonlinear Parameters. In: *Journal of the Society for Industrial and Applied Mathematics* 11 (1963), Nr. 2, S. 431–441. http://dx.doi.org/10.1137/0111030. – DOI 10.1137/0111030

[47] Forschungscluster MERGE: *Technologiefusion für multifunktionale Leichtbaustrukturen.* – https://www.tu-chemnitz.de/MERGE/, letzter Zugriff am 17.10.2021

[48] Mounie, G. ; Rapine, C. ; Trystram, D. : A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks. In: *SIAM Journal on Computing* 37 (2007), Nr. 2, S. 401–412. http://dx.doi.org/10.1137/S0097539701385995. – DOI 10.1137/S0097539701385995

[49] Nivetha, S. K. ; Asokan, R. ; Senthilkumaran, N. : Metaheuristics in Mobile AdHoc Network Route Optimization. In: *2019 TEQIP III Sponsored International Conference on Microwave Integrated Circuits, Photonics and Wireless Networks (IMICPW)*, 2019, S. 414–418

[50] N'Takpé, T. ; Suter, F. : Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms. In: *Proceedings of the 12th International Conference on Parallel and Distributed Systems - Volume 1*, IEEE Computer Society, 2006 (ICPADS '06). – ISBN 0769526128, S. 3–10

[51] N'Takpe, T. ; Suter, F. ; Casanova, H. : A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In: *Sixth International Symposium on Parallel and Distributed Computing (ISPDC'07)*, 2007, S. 35–35

[52] Ohlmann, J. W. ; Thomas, B. W.: A Compressed-Annealing Heuristic for the Traveling Salesman Problem with Time Windows. In: *INFORMS J. Comput.* 19 (2007), S. 80–90

[53] Omara, F. A. ; Arafa, M. M.: Genetic algorithms for task scheduling problem. In: *Journal of Parallel and Distributed Computing* 70 (2010), Nr. 1, S. 13–22. http://dx.doi.org/10.1016/j.jpdc.2009.09.009. – DOI 10.1016/j.jpdc.2009.09.009. – ISSN 0743–7315

[54] OpenBLAS: *An optimized BLAS library.* – http://www.openblas.net/, letzter Zugriff am 17.10.2021

[55] Orr, M. ; Sinnen, O. : Integrating Task Duplication in Optimal Task Scheduling With Communication Delays. In: *IEEE Transactions on Parallel and Distributed Systems* 31 (2020), Nr. 10, S. 2277–2288. http://dx.doi.org/10.1109/TPDS.2020.2989767. – DOI 10.1109/TPDS.2020.2989767

[56] Orr, M. ; Sinnen, O. : Optimal task scheduling benefits from a duplicate-free state-space. In: *Journal of Parallel and Distributed Computing* 146 (2020), S. 158–174. http://dx.doi.org/10.1016/j.jpdc.2020.07.005. – DOI 10.1016/j.jpdc.2020.07.005. – ISSN 0743–7315

[57] Orsila, H. ; Salminen, E. ; Hämäläinen, T. D.: Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems. Version: 2008. http://dx.doi.org/10.5772/5559. In: Tan, C. M. (Hrsg.): *Simulated Annealing*. Rijeka : IntechOpen, 2008. – DOI 10.5772/5559, Kapitel 16

[58] Pearl, J. ; Kim, J. : Studies in Semi-Admissible Heuristics. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-4 (1982), S. 392–399

[59] Radulescu, A. ; Nicolescu, C. ; van Gemund, A. J. C. ; Jonker, P. P.: CPR: mixed task and data parallel scheduling for distributed systems. In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, 2001, S. 9 pp.–

[60] Ramaswamy, S. ; Sapatnekar, S. ; Banerjee, P. : A framework for exploiting task and data parallelism on distributed memory multicomputers. In: *IEEE Transactions on Parallel and Distributed Systems* 8 (1997), Nr. 11, S. 1098–1116. http://dx.doi.org/10.1109/71.642945. – DOI 10.1109/71.642945

[61] Rauber, T. ; Rünger, G. : A Scheduling Selection Process for Energy-efficient Task Execution on DVFS Processors. In: *Concurrency and Computation: Practice and Experience* 31 (2019), Juni. http://dx.doi.org/10.1002/cpe.5043. – DOI 10.1002/cpe.5043. – ISSN 1532–0626

[62] Rauber, T. ; Rünger, G. : Multiprocessor Task Programming and Flexible Load Balancing for Time-Stepping Methods on Heterogeneous Cloud Infrastructures. In: *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2019, S. 1537–1544

[63] Rauber, T. ; Rünger, G. : Compiler support for task scheduling in hierarchical execution models. In: *Journal of Systems Architecture* 45 (1999), Nr. 6, S. 483–503. http://dx.doi.org/10.1016/S1383-7621(98)00019-8. – DOI 10.1016/S1383–7621(98)00019–8. – ISSN 1383–7621

[64] Rauber, T. ; Rünger, G. : *Parallele Programmierung, 3. Auflage*. Springer, 2012 (eXamen.press). – ISBN 978–3–642–13603–0

[65] Reiter, R. : Scheduling Parallel Computations. In: *J. ACM* 15 (1968), Okt., Nr. 4, S. 590–599. http://dx.doi.org/10.1145/321479.321485. – DOI 10.1145/321479.321485. – ISSN 0004–5411

[66] Russell, S. ; Norvig, P. : *Artificial Intelligence: A Modern Approach*. 3rd. Prentice Hall Press, 2016. – ISBN 9781292153964

[67] Sahoo, R. M. ; Padhy, S. K.: Elephant Herding Optimization for Multiprocessor Task Scheduling in Heterogeneous Environment. In: Das, A. K. (Hrsg.) ; Nayak, J. (Hrsg.) ; Naik, B. (Hrsg.) ; Dutta, S. (Hrsg.) ; Pelusi, D. (Hrsg.): *Computational Intelligence in Pattern Recognition*. Singapore : Springer Singapore, 2020. – ISBN 978–981–15–2449–3, S. 217–229

[68] Sahoo, R. M. ; Padhy, S. K.: Improved Crow Search Optimization for Multiprocessor Task Scheduling: A Novel Approach. In: Nayak, J. (Hrsg.) ; Balas, V. E. (Hrsg.) ; Favorskaya, M. N. (Hrsg.) ; Choudhury, B. B. (Hrsg.) ; Rao, S. K. M. (Hrsg.) ; Naik, B. (Hrsg.): *Applications of Robotics in Industry Using Advanced Mechanisms*. Cham : Springer International Publishing, 2020. – ISBN 978–3–030–30271–9, S. 1–13

[69] Sakalis, C. ; Leonardsson, C. ; Kaxiras, S. ; Ros, A. : Splash-3: A properly synchronized benchmark suite for contemporary research. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2016, S. 101–111

[70] Semar Shahul, A. Z. ; Sinnen, O. : Scheduling task graphs optimally with A*. In: *The Journal of Supercomputing* 51 (2010), Mar, Nr. 3, S. 310–332. http://dx.doi.org/10.1007/s11227-010-0395-1. – DOI 10.1007/s11227–010–0395–1. – ISSN 1573–0484

[71] Singh, J. ; Singh, G. : Improved Task Scheduling on Parallel System using Genetic Algorithm. In: *International Journal of Computer Applications* 39 (2012), S. 17–22

[72] Sinnen, O. : Reducing the solution space of optimal task scheduling. In: *Computers & Operations Research* 43 (2014), S. 201 – 214. – ISSN 0305–0548

[73] Skillicorn, D. B. ; Hill, J. ; McColl, W. : Questions and answers about BSP. In: *Scientific Programming* 6 (1997), Nr. 3, S. 249–274. http://dx.doi.org/10.1155/1997/532130. – DOI 10.1155/1997/532130

[74] Suter, F. : Scheduling Δ-Critical Tasks in mixed-parallel applications on a national grid. In: *2007 8th IEEE/ACM International Conference on Grid Computing*, 2007, S. 2–9

[75] Teh, E. K. ; Zawawi, M. A. M. ; Mohamed, M. F. P. ; Isa, N. A. M.: Practical Full Chip Clock Distribution Design With a Flexible Topology and Hybrid Metaheuristic Technique. In: *IEEE Access* 9 (2021), S. 14816–14835. http://dx.doi.org/10.1109/ACCESS.2021.3053052. – DOI 10.1109/ACCESS.2021.3053052

[76] Topcuoglu, H. ; Wu, M. you ; al. et: Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. In: *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS* 13 (2002), Nr. 3, S. 260–274

[77] Turner, H. ; White, J. : Multi-core Deployment Optimization Using Simulated Annealing and Ant Colony Optimization. In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013, S. 1216–1223

[78] Vodel, M. ; Lippmann, M. ; Caspar, M. ; Hardt, W. : A capable, high-level scheduling concept for application-specific wireless sensor networks. In: *2010 International Symposium on Information Technology* Bd. 2, 2010, S. 914–919

[79] Wang, Y.-R. ; Huang, K.-C. ; Wang, F.-J. : Scheduling online mixed-parallel workflows of rigid tasks in heterogeneous multi-cluster environments. In: *Future Generation Computer Systems* 60 (2016), S. 35–47. http://dx.doi.org/10.1016/j.future.2016.01.013. – DOI 10.1016/j.future.2016.01.013. – ISSN 0167–739X

[80] Watanabe, T. ; Nishikawa, H. ; Tomiyama, H. : Scheduling of Rigid Tasks on Heterogeneous Multicores. In: *2020 International SoC Design Conference (ISOCC)*, 2020, S. 330–331

[81] Wong, Y. W. ; Goh, R. S. M. ; Kuo, S.-H. ; Low, M. Y. H.: A Tabu Search for the Heterogeneous DAG Scheduling Problem. In: *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*. USA : IEEE Computer Society, 2009 (ICPADS '09). – ISBN 9780769539003, S. 663–670

Thesen zur Dissertation

(1) Das Scheduling unabhängiger paralleler Tasks ist komplexer als das Scheduling von Single-Prozessor-Tasks, unabhängig davon, ob zwischen diesen Tasks Abhängigkeiten vorliegen oder nicht.

(2) Das Laufzeitverhalten gewöhnlicher paralleler Anwendungen kann mithilfe einer parametrisierten Laufzeitformel, bestehend aus einem parallelen und einem sequentiellen Anteil sowie einem logarithmischen Anteil, sehr gut abgebildet werden.

(3) Das Schedulingproblem für parallele Tasks und Cluster von Multicore-Systemen lässt sich als kombinatorisches Optimierungsproblem formulieren, auf das sich lokale und globale Suchverfahren anwenden lassen.

(4) Das komplexe Scheduling paralleler Tasks profitiert dahingehend vom Einsatz der entwickelten suchbasierten Schedulingverfahren HP*, MTaSS, SAMT und WLS, dass diese Verfahren Schedules mit deutlich geringeren Laufzeiten erzeugen als die bestehenden List-Scheduling-Heuristiken HCPA und Δ-CTS.

(5) Das Verfahren MTaSS benötigt eine geringfügig längere Zeit für das Scheduling als die List-Scheduling-Heuristiken HCPA und Δ-CTS. Die damit ermöglichte Reduzierung der Laufzeit der erzeugten Schedules übersteigt jedoch diese Mehrkosten deutlich, sodass durch MTaSS insgesamt ein Laufzeitgewinn erreicht wird.

(6) Der in einem Schedulingverfahren verwendete Ansatz zur Konstruktion eines Schedules für parallele Tasks, das heißt entweder eine inkrementelle oder eine modifizierende Konstruktion, hat weder einen direkten Einfluss auf die Laufzeit der erzeugten Schedules noch auf die Laufzeit des Verfahrens.