

**Evaluation der Leistungsfähigkeit  
von gemischt-parallelen Programmen  
in homogenen und heterogenen Umgebungen  
unter Berücksichtigung effizienter Schedulingstrategien**

Von der Universität Bayreuth  
zur Erlangung des Grades eines  
Doktors der Naturwissenschaften (Dr. rer. nat.)  
genehmigte Abhandlung

von  
**Sascha Hunold**  
aus Merseburg

1. Gutachter: Prof. Dr. Thomas Rauber
2. Gutachter: Prof. Dr. Jens Gustedt

Tag der Einreichung: 22.10.2008  
Tag des Kolloquiums: 12.01.2009



*Für Petra und Jürgen*



## Kurzfassung

Die gemischt-parallele Formulierung von Programmen, welche aus kooperierenden Multi-prozessor-Tasks (M-Tasks) bestehen, erlaubt einen höheren Grad an Parallelität als gewöhnliche datenparallele Implementierungen. Um diesen höheren Parallelitätsgrad auszunutzen, bedarf es effizienter gemischt-paralleler Realisierungen von Algorithmen, einer guten Infrastruktur zur Ausführung der Programme und leistungsfähigen Scheduling-Algorithmen, die die einzelnen M-Tasks auf die bestmögliche Menge von Prozessoren abbilden.

Im ersten Teil der vorliegenden Arbeit werden exemplarisch verschiedene gemischt-parallele Realisierungen der Matrixmultiplikation zweier dicht besetzter Matrizen untersucht. Dazu werden Algorithmen (z. B. die Matrixmultiplikation nach Strassen) so umstrukturiert, dass die resultierenden Verfahren aus hierarchisch organisierten, datenparallelen Multiprozessor-Tasks bestehen. Durch die abstraktere Beschreibung von Problemen mittels kooperierender Tasks lassen sich Algorithmen einfacher miteinander kombinieren. In dieser Arbeit wurden verschiedene gemischt-parallele Algorithmen zu neuen Poly-Algorithmen zusammengesetzt, wobei die gemischt-parallele Variante von Strassens Algorithmus als Ausgangsalgorithmus gewählt wurde. Die so entstandenen Poly-Algorithmen zur Matrixmultiplikation wurden in einer Vielzahl von Experimenten mit der Leistung datenparalleler Implementierungen auf homogenen parallelen und verteilten Systemen verglichen. Dabei zeigte sich, dass die gemischt-parallelen Varianten für viele Konfigurationen kürzere Laufzeiten als die datenparallelen Algorithmen erreichen.

Gemischt-parallele Programme lassen sich als gerichteter azyklischer Graph (DAG) beschreiben. Diese Darstellung ist sehr gut für eine verteilte Abarbeitung der einzelnen Knoten (M-Tasks) über Clustergrenzen hinaus geeignet. Trotzdem benötigt man eine entsprechende Software-Infrastruktur, um gemischt-parallele Programme auf verschiedenen Clustern auszuführen. Aus diesem Grund wurde im Rahmen dieser Arbeit TGrid entwickelt, um gemischt-parallele Applikationen im Grid auszuführen. TGrid ist zum einen eine *Middleware*, die verschiedene heterogene Systeme zu einem kooperierenden System zusammenfügt. Zum anderen bietet TGrid eine Programmierschnittstelle, um gemischt-parallele Programme zu formulieren und diese mit Hilfe der Middleware auszuführen. Die TGrid-Middleware ermöglicht die *Co-Allokation von Ressourcen* für eine einzige gemischt-parallele Anwendung, d. h. ein einziges Programm kann durch mehrere Cluster parallel abgearbeitet werden. Eine weitere wichtige Eigenschaft ist die Unterstützung der automatischen Datenumverteilung zwischen M-Tasks. Der Programmierer muss dazu nur die Abbildung der Ausgangsdatenstrukturen auf die Eingangsdatenstrukturen zweier M-Tasks definieren. Die eigentliche Datenkommunikation übernimmt das TGrid-System.

Für eine effiziente Ausführung gemischt-paralleler Programme in Clustern und Multiclustern (Cluster aus Clustern) ist auch die Frage zu klären, in welcher Reihenfolge die ausführbaren Tasks abgearbeitet werden sollen. Das Ausführen von dynamisch erzeugten M-Taskgraphen in Multiclustern führt zu einer neuen Klasse von Scheduling-Problemen. Deshalb werden in der vorliegenden Arbeit zwei Algorithmen (RePA und DMHEFT) für das Scheduling von dynamisch generierten Taskgraphen entwickelt und deren Leistungsfähigkeit mit Compile-Zeit-Verfahren wie MHEFT verglichen.

Da TGrid auch für die Ausführung von statisch definierten Taskgraphen genutzt werden kann, wird ein neuer *zweistufiger Scheduling-Algorithmus* (RATS) vorgestellt, welcher zu Compile-Zeit arbeitet. Dieser Algorithmus versucht durch gezielte Änderung der Prozessor-Allokation von Tasks, die *Kosten* der *Datenumverteilung* zu *reduzieren*. Nach genauer Analyse mittels Grid-Simulationen konnte festgestellt werden, dass RATS deutlich kürzere Ablaufpläne als andere zweistufige Verfahren, wie z. B. HCPA, auf homogenen Clustern produziert.

Zusammenfassend zeigt die vorliegende Arbeit, wie gemischt-parallele Algorithmen entwickelt und effizient ausgeführt werden können. In homogenen parallelen Systemen ermöglichen diese gemischt-parallelen Anwendungen bessere Laufzeiten als datenparallele Implementierungen. Die Arbeit verdeutlicht auch, dass eine gemischt-parallele Formulierung von Algorithmen eine effiziente Ausführung von parallelen Verfahren in Grid-Umgebungen (Multiclustern) erlaubt, die mit anderen parallelen Programmiermodellen in diesen nicht zu erreichen ist.

# Evaluating the performance of mixed-parallel programs in homogeneous and heterogeneous environments using efficient scheduling strategies

## Abstract

The formulation of applications in a mixed-parallel way, in which a program is composed of concurrently executable multiprocessor tasks (M-tasks), can lead to a higher degree of parallelism than common data-parallel implementations. It is challenging to obtain efficiently performing mixed-parallel applications. Well-suited programming environments and well-performing scheduling algorithms should be available to exploit the full potential of this class of algorithms.

In the present work, several *mixed-parallel realizations of matrix multiplication* algorithms are developed to study the performance potential of mixed-parallel applications. Two new mixed-parallel realizations of a matrix multiplication for square dense matrices are introduced. The first algorithm is a mixed-parallel version of a panel-panel algorithm called tpMM. Secondly, a realization of *Strassen's algorithm* is presented using a recursive decomposition into a hierarchy of M-tasks. Since Strassen's algorithm is formulated hierarchically it is possible to use different building blocks to solve the remaining sub-problems at the cut-off level of Strassen's algorithm. This leads to a hierarchy of building blocks which can be combined to obtain different poly-algorithms, all starting with a mixed-parallel decomposition using Strassen's algorithm. The performance of these mixed-parallel combinations is compared with commonly used data-parallel implementations, e.g. PDGEMM of ScaLAPACK, on homogeneous distributed memory systems. The experimental results show that a mixed-parallel realization of Strassen's algorithm can lead to a better performance than data-parallel matrix multiplication algorithms. The resulting performance mainly depends on the number of processors and the size of the input matrices.

Mixed-parallel applications can be modeled as directed acyclic graphs (DAGs), in which a node represents an M-task and the edges denote data dependencies between M-tasks. Such a representation is well-suited to execute mixed-parallel algorithms across different clusters. Since the distributed execution of M-tasks on the grid requires a specific software infrastructure, TGrid has been developed as part of the present work. TGrid is a *grid middleware* and a *runtime system* to develop and to execute mixed-parallel applications. The TGrid middleware targets multi-clusters, which are clusters of clusters. In contrast to other grid middleware systems, TGrid allows the *co-allocation* of resources of different clusters to M-tasks belonging to the same application. It also provides a redistribution component that automatically performs data redistributions between cooperating M-tasks and that does not require to write data redistribution code by hand.

The execution of mixed-parallel applications also raises the question of how to schedule executable M-tasks on the platform. Hence, the present work introduces two novel scheduling algorithms, RePA and DMHEFT. They can be used to schedule dynamically generated DAGs of M-tasks in a multi-cluster environment. The scheduling performance

in terms of makespan of both algorithms is compared to well-performing compile time algorithms such as MHEFT.

Since TGrid can also be used to execute static DAGs, this work also introduces a novel compile time scheduling algorithm (RATS) for homogeneous clusters. The RATS algorithm uses two steps to produce the compile time schedule. First, an allocation phase assigns a number of processors to each M-task. In a second phase, the algorithm maps the allocations onto the platform using a list-scheduling approach. The RATS algorithm attempts to reduce the redistribution costs of cooperating M-tasks by reconsidering the allocation size in the mapping phase and by changing this size where applicable. Simulations have shown that the RATS algorithm significantly improves the performance of two-step approaches such as HCPA on homogeneous clusters.

In summary, the present work shows how mixed-parallel applications can be developed and efficiently executed. These mixed-parallel algorithms can lead to faster execution times than data-parallel implementations on homogeneous parallel systems. The present work also demonstrates that, in contrast to other programming models, mixed-parallel applications can be efficiently executed in grid environments such as multi-clusters.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Programme mit gemischter Parallelität	2
1.1.1	Programmiermodelle	2
1.1.2	Von taskparallelen zu gemischt-parallelen Programmen	2
1.2	Ziele der Arbeit	4
<b>2</b>	<b>Gemischt-parallele Matrixmultiplikationen</b>	<b>5</b>
2.1	Zielsetzung	5
2.2	Überblick über verwandte Arbeiten	6
2.3	Taskparallele Matrixmultiplikation (tpMM)	8
2.3.1	Matrixmultiplikation	8
2.3.2	Idee des parallelen Algorithmus	8
2.3.3	Voraussetzungen	10
2.3.4	Beschreibung des parallelen Algorithmus	10
2.3.5	Vorteile von tpMM	11
2.3.6	Experimentelle Auswertung	12
2.3.7	Fazit tpMM	13
2.4	Mehrstufige Realisierung des Algorithmus von Strassen	14
2.4.1	Motivation	14
2.4.2	Sequenzieller Algorithmus	14
2.4.3	Definition der Multiprozessor-Tasks	15
2.4.4	Algorithmische Bausteine	17
2.4.5	Kombination: Strassen und Ring	21
2.4.6	Kombination: Strassen und PDGEMM	21
2.4.7	Kombination: Strassen und tpMM	23
2.4.8	Anwendbarkeit der mehrstufigen Kombinationen	24
2.4.9	Laufzeitmodellierung	24
2.4.10	Experimentelle Auswertung	30
2.5	Automatische Parameterabstimmung von PDGEMM	39
2.5.1	Motivation	39
2.5.2	PDGEMM – Überblick über verwendete Algorithmen	39
2.5.3	Leistungssensitive Parameter von PDGEMM	41
2.5.4	Parameteranalyse	43
2.5.5	Automatische Bestimmung der logischen Blockgröße	46
2.5.6	Experimentelle Auswertung	48
2.5.7	Fazit	49
2.6	Zusammenfassung und Fazit	50

<b>3</b>	<b>Laufzeitsystem für M-Task-Programme auf homogenen Clustern</b>	<b>51</b>
3.1	Motivation und Ziele	51
3.2	Verwandte Arbeiten	53
3.3	Architektur der Bibliothek	54
3.4	Die vShark-Programmierschnittstelle	55
3.5	Fallstudie mit MPI und POSIX-Threads	57
3.6	Experimentelle Auswertung	61
3.7	Zusammenfassung und Fazit	66
<b>4</b>	<b>TGrid: Eine Laufzeitumgebung für M-Tasks</b>	<b>67</b>
4.1	Einführung Grids	67
4.2	Motivation und Entwicklungsziel	68
4.3	Überblick über verwandte Arbeiten	69
4.4	Laufzeitumgebung für M-Task-Programme	73
4.5	Definition der TGrid-Architektur	75
4.6	Konfiguration von TGrid	81
4.7	Programmierung von TGrid-Applikationen	81
4.8	Datenumverteilung – Die MxN-Komponente	83
4.8.1	Verwandte Ansätze	83
4.8.2	Aufbau der Datenumverteilungskomponente	85
4.8.3	Protokoll zur Datenumverteilung	87
4.9	Experimentelle Analyse von TGrid	89
4.9.1	Matrixmultiplikation nach Strassen	89
4.9.2	Parallele Berechnung eines Mandelbrot-Fraktals	94
4.9.3	Experimentelle Evaluation der MxN-Komponente	95
4.10	Zusammenfassung und Fazit	100
<b>5</b>	<b>Scheduling von M-Taskgraphen</b>	<b>103</b>
5.1	Motivation und Zielsetzung	104
5.2	Definition einiger Scheduling-Begriffe	105
5.3	Verwandte Arbeiten	107
5.4	Scheduling von M-Taskgraphen in Multiclustern	111
5.4.1	Plattformmodell	112
5.4.2	Applikationsmodell für gemischt-parallele Applikationen	113
5.5	Der ReP-Algorithmus (RePA)	113
5.5.1	Motivation und Ziel	113
5.5.2	Beschreibung des Algorithmus	114
5.5.3	Komplexitätsanalyse	118
5.5.4	Experimentelle Auswertung	119
5.6	DMHEFT	125
5.6.1	Motivation und Ziel	125
5.6.2	Beschreibung des DMHEFT-Algorithmus	127
5.6.3	Komplexitätsanalyse	130
5.6.4	Experimentelle Auswertung	131

---

5.6.5	Fazit: Vergleich RePA und DMHEFT . . . . .	135
5.7	RATS: Zweisritt-Scheduling mit Datenumverteilung . . . . .	136
5.7.1	Motivation . . . . .	136
5.7.2	Mapping unter Berucksichtigung der Datenumverteilung . . . . .	138
5.7.3	Komplexitatsanalyse . . . . .	142
5.7.4	Experimentelle Auswertung . . . . .	143
5.7.5	Fazit RATS . . . . .	153
<b>6</b>	<b>Zusammenfassung</b>	<b>155</b>
<b>A</b>	<b>Werkzeuge zur Untersuchung von Scheduling-Algorithmen</b>	<b>161</b>
A.1	DAG-Generator . . . . .	161
A.2	Grid-Generator . . . . .	162
A.3	Jedule . . . . .	162
A.4	mpiJava . . . . .	163
<b>B</b>	<b>bersicht uber die verwendeten Plattformen</b>	<b>165</b>
<b>C</b>	<b>Zugehorige Publikationen</b>	<b>169</b>
<b>D</b>	<b>Danksagung</b>	<b>171</b>
	<b>Literaturverzeichnis</b>	<b>173</b>



# Abbildungsverzeichnis

1.1	Beispiel für eine taskparallele Abarbeitung von Instruktionen . . . . .	3
2.1	Berechnungsmuster von tpMM . . . . .	9
2.2	Taskgraph von tpMM . . . . .	9
2.3	Gruppenhierarchie für 4 Prozessoren . . . . .	10
2.4	Kommunikationsmuster von tpMM für 4 Prozessoren . . . . .	10
2.5	Berechnungsreihenfolge von tpMM . . . . .	12
2.6	Leistungsvergleich von PDGEMM mit tpMM . . . . .	13
2.7	Vergleich: Operationen Strassen vs. Standardmethode . . . . .	16
2.8	Kommunikationsmuster der Ring-Methode . . . . .	20
2.9	Kombinationsmöglichkeiten für Strassen-Algorithmus . . . . .	21
2.10	Verteilung Matrix $B$ für <i>Strassen+Ring</i> . . . . .	22
2.11	Blockzyklische Verteilung der Matrixblöcke . . . . .	22
2.12	Abbildung der Prozessoren für <i>Strassen+PDGEMM</i> . . . . .	23
2.13	Abbildung der Prozessoren für <i>Strassen+tpMM</i> . . . . .	23
2.14	Erwartete vs. gemessene Laufzeit von PDGEMM und <i>Strassen+Ring</i> . . . . .	35
2.15	Leistungsvergleich Opteron-Cluster . . . . .	36
2.16	Leistungsvergleich IBM Regatta p690 . . . . .	37
2.17	Leistungsvergleich SGI Altix 4700 . . . . .	38
2.18	DIMMA-Snapshot . . . . .	41
2.19	Fortran-Schnittstelle von PDGEMM . . . . .	41
2.20	Leistungsvergleich von PDGEMM mit verschiedenen Blockgrößen . . . . .	44
2.21	Leistung von PDGEMM für unterschiedliche logische Blockgrößen . . . . .	45
2.22	MFLOPS vs. Cache-Fehlzugriffe von DGEMM . . . . .	46
2.23	DGEMM-Profil . . . . .	47
2.24	Leistungsspektrum von DGEMM . . . . .	48
2.25	Leistung von PDGEMM auf unterschiedlichen Clustern . . . . .	49
3.1	Datenverteilung der Matrix $B$ bei tpMM . . . . .	52
3.2	vShark-Architektur . . . . .	55
3.3	vShark-Komponenten . . . . .	56
3.4	Send/Recv-Programm mit vShark . . . . .	58
3.5	Aufbau vShark mit MPI und POSIX-Threads . . . . .	59
3.6	vShark/MPI-Protokoll . . . . .	60
3.7	Konfigurationsdatei des vShark-MPI-Adapters . . . . .	61
3.8	Durchsatz bei COMMS1, XEON-SCI-Cluster . . . . .	63
3.9	Sättigungsbandbreite bei COMMS3, XEON-SCI-Cluster . . . . .	64

3.10	Laufzeit tpMM: vShark vs. MPI, XEON-SCI-Cluster	65
3.11	Laufzeit tpMM: MPICH-P4 vs. MPICH-VMI vs. vShark	65
4.1	Struktur eines TGrid-Programms	74
4.2	tgrid-Beispielkonfiguration	75
4.3	Kernmodule von TGrid	76
4.4	Abarbeitung eines TGrid-Programms	77
4.5	TGrid-Konfigurationsdatei	82
4.6	TGrid-Beispielprogramm	84
4.7	TGrid-Beispielkonfiguration	86
4.8	Umverteilungsprotokoll der MxN-Komponente	90
4.9	Struktur der Matrixmultiplikation nach Strassen für TGrid	91
4.10	Struktur der Strassen-Task	92
4.11	Laufzeiten der Matrixmultiplikation mit TGrid	93
4.12	Mandelbrot-Beispiel	94
4.13	Laufzeit Fraktalprogramm mit TGrid	95
4.14	Durchsatz Intra-Subnet-Kommunikation	96
4.15	Durchsatz 1×1-Umverteilung	98
4.16	Durchsatz Intra- vs. Inter-Subnet-Umverteilung	99
5.1	Illustration: Kritischer Pfad und Bottom-Level	106
5.2	Zweischritt-Algorithmus exemplarisch	108
5.3	Plattformmodell	112
5.4	Illustration der Motivation von RePA	115
5.5	Graphen mit verschiedenen <i>fat</i> -Werten	120
5.6	Datenumverteilung von blockverteilten Matrizen	122
5.7	Problem von RePA (grafisch)	126
5.8	Postponing-Strategie von DMHEFT	129
5.9	Empirische Bestimmung der Postponing-Parameter	132
5.10	Durchschnittlicher Makespan von DMHEFT	134
5.11	Arbeitsweise von RATS	137
5.12	Motivierendes Beispiel für RATS	138
5.13	FFT-DAG und Strassen-DAG	146
5.14	Relativer Makespan RATS vs. HCPA, Grillon	147
5.15	Relative Arbeit RATS vs. HCPA, Grillon	148
5.16	RATS: Optimierung von $\delta$ , relativer Makespan, Grillon	149
5.17	RATS: Optimierung von $\rho$ , relativer Makespan, Grillon	150
5.18	Relativer Makespan von RATS mit optimierten Parametern, Grillon	151
5.19	Relative Arbeit von RATS mit optimierten Parametern, Grillon	151
A.1	Beispielausgabe von <i>Jedule</i>	164
A.2	Bilderzeugung mit <i>Jedule</i> aus der Kommandozeile	164

# Tabellenverzeichnis

2.1	Zuordnung der $Q_i$ zu den M-Tasks . . . . .	17
2.2	Interner Aufbau Schema-Q7 . . . . .	18
2.3	Interner Aufbau Schema-Q8 . . . . .	19
2.4	Implementierungen von Strassens Algorithmus . . . . .	24
2.5	berblick der mehrstufigen Algorithmen und deren Bausteine . . . . .	31
2.6	Überblick über die Testsysteme . . . . .	32
2.7	Ermittelte logische Blockgrößen für PDGEMM . . . . .	32
2.8	Schnellster Algorithmus für jede Plattform . . . . .	39
2.9	Testsystem für PDGEMM-Optimierung . . . . .	43
3.1	Konfiguration für den COMMS1-Benchmark . . . . .	62
4.1	Konfiguration für Strassen-Test mit TGrid . . . . .	92
5.1	Vergleich der Makespans: MHEFT vs. RePA . . . . .	123
5.2	Paarweiser Algorithmenvergleich . . . . .	135
5.3	DMHEFT: Durchschnittliche Abweichung vom Besten . . . . .	135
5.4	RATS: Eckdaten der betrachteten Cluster. . . . .	144
5.5	RATS: Übersicht über randomisierte DAGs . . . . .	145
5.6	Ermittelte Parameter für RATS . . . . .	149
5.7	Paarweiser Vergleich von RATS und HCPA . . . . .	152
5.8	RATS: Durchschnittliche Abweichung vom besten Algorithmus . . . . .	153



# Algorithmenverzeichnis

2.1	tpMM	11
2.2	SUMMA	40
5.1	HEFT	107
5.2	CPA	108
5.3	RePA	116
5.4	DMHEFT	128
5.5	RATS	142



# Kapitel 1

## Einleitung

*The way to think of a supercomputer is as a special-purpose device. Only with these devices can we perform this cutting-edge research.*

---

JACK DONGARRA

Die effiziente und fehlerfreie gleichzeitige Ausführung von Programmen hat eine stetig steigende Bedeutung in unserer Gesellschaft. Die zunehmende Vernetzung unterschiedlicher Entitäten des Alltags (Laptops, Mobiltelefone, Terminals in Banken und Flughäfen) erfordert immer komplexere Software, welche die angebotenen Services miteinander verbindet. Die zur Verfügung stehenden hohen Bandbreiten zwischen Computern sind dabei zugleich eine Chance, Rechenleistung und Services einer großen Anzahl an Kunden zur Verfügung zu stellen. Mitte der 1990er Jahre – bedingt durch die drastische Reduzierung der Komponentenpreise und durch die Verfügbarkeit von freien Betriebssystemen wie Linux – wurden viele so genannte Beowulf-Cluster<sup>1</sup> installiert. Diese Cluster bestanden aus normalen, preiswerten PC-Komponenten und wurden mit einem dedizierten High-Speed-Netzwerk verbunden. Viele dieser Cluster setzten schon damals SMP-Systeme (2 oder 4 Wege) als Basisbausteine ein. Als Beispiel sei der CLIC (Chemnitzer Linux Cluster) genannt, der im Jahr 2000 mit 528 Pentium-III-Prozessoren eine damals erhebliche Rechenleistung von 143 GFLOPS erzielte. In den letzten Jahren folgte eine immer weitere Verbreitung solcher Cluster-Systeme, so dass sie heute an vielen wissenschaftlichen Einrichtungen zu finden sind. Damit einher ging eine Verbesserung der Infrastruktur des Internets, so dass Latenzzeiten verkürzt und vor allem Bandbreiten drastisch erweitert wurden. Damit waren die Voraussetzungen erfüllt, um die Rechenleistung einzelner geografisch verteilter Cluster-Systeme zu vereinen und Forschern diese Kapazitäten zur Verfügung zu stellen. Die von diesem Gedanken getragenen Ansätze werden heute unter dem Begriff Grid-Computing zusammengefasst. Das „Grid“ soll eine Infrastruktur bieten, um Dienste im Netz transparent benutzen zu können. Diese Dienste können mannigfaltig sein, z. B. können Tickets gebucht, Plätze in einem Konzert reserviert oder Rechenleistung zur Verfügung gestellt werden.

---

<sup>1</sup><http://www.beowulf.org>

Die vorliegende Arbeit versucht, die im Grid verfügbare Rechenleistung für eine einzelne Applikation nutzbar zu machen. Es ist nicht möglich, traditionelle parallele Programme, die für dedizierte homogene Supercomputer entwickelt wurden, auf das Grid zu übertragen, da hierfür die gesamte Plattform zu dynamisch und zu heterogen ist. Aus diesem Grund wird versucht, aus rein datenparallelen Verfahren gemischt-parallele Implementierungen zu erzeugen, die für die Ausführung im Grid besser angepasst sind. Diese Arbeit untersucht deshalb die Eignung von gemischt-parallelen Programmen für homogene und heterogene Plattformen.

## 1.1 Programme mit gemischter Parallelität

### 1.1.1 Programmiermodelle

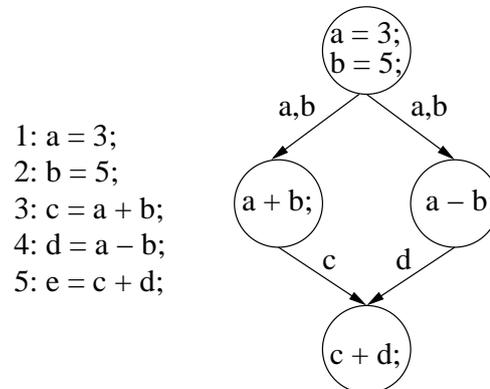
Es gibt zwei klassische Programmiermodelle für parallele Systeme. Beim SPMD-Modell (Single Program Multiple Data) wird das gleiche Programm auf unterschiedliche Daten angewendet [91]. Im MPMD-Programmiermodell (Multiple Program Multiple Data) hingegen arbeiten mehrere Programme gleichzeitig auf unterschiedlichen Daten, was einen größeren Verwaltungsaufwand bedeutet. Der zweite Ansatz bietet aber mehr Flexibilität und eine erhöhte Gesamtparallelität des Programms. Folgen die Programme diesen Modellen spricht man auch von *datenparallelen* (SPMD) oder *taskparallelen* (MPMD) Applikationen. Das Hauptaugenmerk dieser Arbeit liegt auf der effizienten Ausnutzung von Taskparallelität in homogenen und heterogenen Systemen.

### 1.1.2 Von taskparallelen zu gemischt-parallelen Programmen

**Taskparallelität** Taskparallele Programme sind aus mehreren *Blöcken* (Tasks) zusammengesetzt, welche durch eine *Dekomposition* des Problems entstanden sind. Tasks können einzelne Instruktionen oder ganze Programme umfassen. Die damit verbundene *Granularität* ist von zentraler Bedeutung für die Bewertung der potenziellen Parallelität eines Verfahrens. Taskparallele Programme können auf unterschiedlichen parallelen Rechnerarchitekturen und Laufzeitumgebungen ausgeführt werden. Es ist beispielsweise möglich, dass Tasks in einem Master-Worker-Programm von Workern abgearbeitet werden. Ein Worker kann dabei als Thread implementiert sein, welcher wiederum auf einen SMP-Knoten oder einen Multicore-Prozessor abgebildet wird.

Ein einfaches Beispiel für taskparallele Abarbeitung von Instruktionen ist in Abb. 1.1 dargestellt. Die Abbildung zeigt einen *DAG* (directed acyclic graph), in dem die Tasks durch Knoten repräsentiert werden und die Kanten die Abhängigkeiten (Kontroll- und Datenabhängigkeiten) angeben. Da die Statements 3 und 4 voneinander unabhängig sind, können sie parallel ausgeführt werden. Wie bei allen taskparallelen Ansätzen spielen auch bei diesem Beispiel die *Datenabhängigkeiten* eine zentrale Rolle, d. h. Task ( $a + b$ ) kann erst ausgeführt werden, wenn  $a$  und  $b$  verfügbar sind. In parallelen Systemen mit gemeinsamem Speicher sind solche Datenabhängigkeiten vor allem mit Synchronisationsaufwand verbunden, da Task ( $a + b$ ) erst starten darf, wenn  $a$  und  $b$  vorliegen. In Systemen mit ver-

teiltem Speicher spielt beim Datenaustausch die Kommunikationszeit über das Netzwerk eine entscheidende Rolle für die Gesamtperformance der taskparallelen Implementierung.



**Abb. 1.1:** Beispiel einer taskparallelen Abarbeitung von Instruktionen.

Eine Task kann auch hierarchisch aufgebaut sein, d. h. sie kombiniert verschiedene Tasks und bildet mit diesen eine neue Task. Des Weiteren müssen Taskgraphen (DAGs) keineswegs nur statischer Natur sein (siehe „Characteristics of Tasks“ [49]). Tasks können auch dynamisch bei der Ausführung des Programms erzeugt werden [59]. Dynamische Tasks und deren Granularität sind speziell für die Lastbalancierung zwischen Berechnungseinheiten von besonderer Bedeutung, siehe *pool-of-tasks* [109] oder *The Task Parallelism Pattern* [72].

**Gemischte Parallelität** Eine für diese Arbeit wichtige Taskklasse sind die *Multiprozessor-Tasks* (multiprocessor tasks, M-Tasks) [39]. Bei diesem Modell wird einer Task eine Anzahl von Prozessoren zugeordnet, welche diese Task dann ausführen. M-Tasks können datenparallel implementiert werden, z. B. kann MPI (Message Passing Interface) für die interne Datenkommunikation verwendet werden. Sie können aber auch hierarchisch aus anderen M-Tasks zusammengesetzt worden sein. TLib [89] ist eine Bibliothek, die das Entwickeln von hierarchisch strukturierten M-Task-Programmen erleichtert. Die Bibliotheks-API stellt Funktionen zur Erzeugung hierarchischer Prozessorgruppen und zur Koordination nebenläufiger M-Tasks bereit, wobei die M-Tasks beliebig verschachtelt werden können. Damit unterstützt die Bibliothek die Realisierung von taskparallelen Programmen für homogene verteilte Systeme, wobei jede Task datenparallel mit MPI implementiert sein kann. Diese Art von Programmen, die sowohl task- als auch datenparallele Anteile besitzen, werden als Programme mit *gemischter Parallelität* (mixed-parallel) bezeichnet.

In der vorliegenden Arbeit werden gemischt-parallele Algorithmen entwickelt und implementiert, z. B. verschiedene Varianten einer Matrixmultiplikation. Die mit diesen Algorithmen erzielte Leistung (Ausführungsgeschwindigkeit) wird der Leistung der datenparallelen Pendanten gegenübergestellt. Außerdem wird untersucht, welche Voraussetzungen gegeben sein müssen, um gemischt-parallele Algorithmen auch in Grid-Umgebungen effizient abarbeiten zu können.

## 1.2 Ziele der Arbeit

Aus dem Kontext der oben beschriebenen Themen ergeben sich für diese Arbeit folgende Zielstellungen:

1. Es soll untersucht werden, inwieweit sich Standardimplementierungen der linearen Algebra, wie eine *Matrixmultiplikation*, gemischt-parallel realisieren lassen. Wichtig sind dabei Analysen zur Verteilung der Arbeitslast, der erforderlichen Datenlayouts und der zu erwartenden theoretischen Performance.
2. Nach Formulierung von rekursiven, hierarchischen und iterativen Verfahren zur Matrixmultiplikation soll evaluiert werden, ob die *Kombination von unterschiedlichen Algorithmen* in verschiedenen Berechnungsstufen zu einer Verbesserung der Laufzeit führt.
3. Die entwickelten Verfahren sollen hinsichtlich ihrer Leistungsfähigkeit mit Standardverfahren aus gängigen Algebrabibliotheken in homogenen parallelen Systemen experimentell untersucht werden.
4. Ein zentrales Ziel dieser Arbeit ist die Ausführung von taskparallelen Programmen in heterogenen Umgebungen. Speziell richtet sich das Augenmerk auf die Ausführung von gemischt-parallelen Programmen (DAGs aus Multiprozessor-Tasks) in Multiclustern (Cluster bestehend aus homogenen Clustern). Dazu ist eine *Laufzeitumgebung* für diesen Programmtyp zu implementieren, wobei eine Anforderungsanalyse in Bereichen wie Sicherheit, Effizienz und Praktikabilität vorangehen soll.
5. Bei der Abarbeitung von DAGs aus M-Tasks in heterogenen Umgebungen spielt die Zuordnung (Mapping) einer Task auf eine Gruppe von Prozessoren eine entscheidende Rolle für die Ausführungsgeschwindigkeit. Es sollen deshalb verschiedene Verfahren zum *Scheduling* von dynamischen und statischen DAGs entwickelt und ihre Verwendbarkeit und Leistungsfähigkeit bewertet werden.

# Kapitel 2

## Gemischt-parallele Matrixmultiplikation: Design, Implementierung und Leistungsbewertung

*Simplicity is prerequisite for reliability.*

---

EDSGER DIJKSTRA

### 2.1 Zielsetzung

Methoden und Algorithmen der linearen Algebra spielen für viele Berechnungen im wissenschaftlich-technischen Bereich eine zentrale Rolle, z. B. bei Strömungssimulationen oder Grafikanimationen. Aus diesem Grund haben sich unterschiedliche Bibliotheken auf dem Markt etabliert, die Grundfunktionen wie Matrix-Vektor-Multiplikation, Matrix-Transposition oder Eigenwert-Berechnung zur Verfügung stellen. Es gibt Bibliotheken, die speziell die Ausführungsgeschwindigkeit für einen einzelnen Prozessor optimieren, z. B. BLAS [37], ESSL [73] oder ATLAS [112]. Darüber hinaus existiert auch eine Vielzahl von Bibliotheken für parallele und verteilte Systeme wie PLAPACK [6], ScaLAPACK [14] oder PETSc [7]. Mehrere dieser Bibliotheken, wie BLAS, wurden zum Teil schon Ende der 1970er Jahre entworfen. Das gilt auch für viele Bibliotheken für Parallelrechner. Dadurch haben sich verschiedene Implementierungsarten als De-facto-Standards etabliert. Eine solcher De-facto-Standard ist z. B. das Datenlayout von verteilten Matrizen, für das meist eine blockzyklische Datenverteilung verwendet wird. Die Datenverteilung forciert auch die parallele Programmierung im SPMD-Stil, d. h. es werden die gleichen Funktionen auf unterschiedliche Daten angewendet. In vielen Bereichen der mathematischen und naturwissenschaftlichen Programme hat sich gezeigt, dass eine Ausnutzung der im Lösungsverfahren enthaltenen taskparallelen Anteile leistungssteigernd sein kann. Aus diesem Grund untersucht diese Arbeit, wie Algorithmen der linearen Algebra – die Matrix-Matrix-Multiplikation im Speziellen – in gemischt-parallele Programme zu überführen sind. Die Matrixmultiplikation wurde ausgewählt, da sie als Basisbaustein für eine Vielzahl anderer Algorithmen Verwendung findet, z. B. bei der LU- oder QR-Faktorisierung [22, 36]. Die Leistungsfähigkeit der gemischt-parallelen Implementierungen soll dann im Vergleich zu korrespondierenden

datenparallelen Algorithmen bewertet werden. Da hier nicht alle Arten von Matrizen untersucht werden können, beschränkt sich diese Arbeit auf die Multiplikation von zwei dicht besetzten Matrizen. Außerdem werden für die meisten der hierin beschriebenen gemischt-parallelen Algorithmen quadratische Eingabematrizen vorausgesetzt.

## 2.2 Überblick über verwandte Arbeiten

### Matrixmultiplikationsalgorithmen für Einprozessorsysteme oder Systeme mit gemeinsamem Speicher

Es gibt viele effiziente Realisierungen der Matrixmultiplikation, sowohl für einzelne Prozessoren als auch für Parallelrechner. Die Geschwindigkeit der Implementierung hängt in erster Linie vom gewählten Algorithmus ab (z. B. Strassen oder Strassen-Winograd). Sehr entscheidend für eine hohe Performance ist jedoch auch eine optimale Anpassung an die gegebene Hardware. Für viele mathematische Berechnungen wird auf die BLAS (Basic Linear Algebra Subprograms) zurückgegriffen, die für viele Plattformen speziell angepasst wurden [37]. Die BLAS enthalten eine Routine GEMM, welche eine *generelle Matrixmultiplikation* realisiert. Effiziente Implementierungen von GEMM für Einprozessorsysteme sowie für Systeme mit gemeinsamem Speicher (Multicore, SMP) sind u. a. die Bibliotheken ATLAS [112] (Automatically Tuned Linear Algebra Software), Goto-BLAS [48] (BLAS-Implementierung von Kazushige Goto) und PHiPAC [13] (Portable High Performance Ansi C), welche die Berechnungsreihenfolge an die spezifischen Speichergegebenheiten anpassen (Ausnutzung der Caches) und damit zu einer verbesserten Auslastung der Funktionseinheiten (u. a. der Streaming SIMD Extensions 2, SSE2) führen. Diese Implementierungen bilden eine leistungsfähige Grundlage der Matrixmultiplikationen für Rechner mit verteiltem Speicher. Die meisten dieser Cache-optimierten Bibliotheken benutzen die Standardmethode zur Berechnung der Matrixmultiplikation. Huss-Lederman et al. haben jedoch auch gezeigt, wie man Strassens Algorithmus zur Implementierung von GEMM Gewinn bringend einsetzt [57]. Sie zeigen in der Arbeit auch, wie man Matrizen mit ungeraden Dimensionen durch *Padding* so anpasst, dass Strassens Algorithmus auf sie angewendet werden kann. Da es sich oftmals nicht genau vorhersagen lässt, welcher Algorithmus bessere Ergebnisse erzielt, wurden empirische Studien durchgeführt, die für eine bestimmte Hardware untersuchen, ob der Standardalgorithmus oder Strassens Algorithmus in GEMM verwendet werden sollte [31].

### Parallele Matrixmultiplikation

Sehr viele parallele Verfahren der Matrixmultiplikation sind optimierte Varianten der Algorithmen von Fox und Cannon [47]. In beiden Verfahren werden Matrixblöcke auf ein rechteckiges Prozessorgitter verteilt. Danach arbeiten die Algorithmen in alternierenden Berechnungs- und Kommunikationsschritten. Sobald eine lokale Matrixmultiplikation beendet ist, werden die Blöcke der Ausgangsmatrizen zu den Nachbarprozessoren weitergeschickt, z. B. als Zeilen- oder Spaltenbroadcast. Leistungsfähige Varianten des Algorithmus von Fox sind SUMMA [108] und PUMMA [27]. Eine ausführliche Übersicht über Strategien

zur Matrixverteilung sowie eine Diskussion über den Einfluss der Matrixdimensionen auf die Geschwindigkeit wird in [51] gegeben. Eine veränderte Variante des Algorithmus von Fox ist SRUMMA [62], welche RMA-Operationen (*remote memory access*) statt Message-Passing einsetzt, wodurch das Verfahren eine bessere Überlagerung von Kommunikation und Berechnung erreicht. SRUMMA kann demzufolge auch nur angewendet werden, wenn die Plattform RMA unterstützt. Für massiv-parallele Systeme existieren Algorithmen, welche die komplexen 3-dimensionalen Netzwerktopologien (z. B. Torus) ausnutzen, um die erforderliche Kommunikationszeit signifikant zu reduzieren [1].

Parallele Varianten der Matrixmultiplikation nach Strassen oder Strassen-Winograd weisen zwar theoretisch, durch die geringere Anzahl an auszuführenden Operationen, eine kürzere Berechnungszeit auf, allerdings gibt es viele Einschränkungen, wie die Anzahl der Prozessoren, die Dimension der Matrizen und den Speicherverbrauch in den Rekursionsstufen, welche es für eine effiziente Realisierung zu beachten gilt. Parallele Implementierungen der Matrixmultiplikation nach Strassen findet man in [35, 50, 70]. Eine Möglichkeit zur Parallelisierung von Strassens Algorithmus ist, die  $7$  Berechnungen der temporären Matrizen auf  $7^i, i \geq 0$  Prozessoren zu verteilen, z. B. in einer Torus- oder Ring-Konfiguration [28, 40].

Eine weitere hybride Implementierung eines auf Broadcasts basierenden Algorithmus von Cannon wird in [77] für heterogene Cluster vorgestellt. Dabei kommt Cannons Algorithmus zur Verteilung der Berechnungsaufgaben zum Einsatz. Die eigentlichen Berechnungen (lokale *matrix updates*) sind mit Strassens Algorithmus implementiert. Die Autoren benutzen eine spezielle Datenverteilung, welche von der Geschwindigkeit der einzelnen Prozessoren abhängt, d. h. je schneller ein Prozessor, umso mehr Elemente werden ihm zugeordnet. Dieses Vorgehen der Elementzuordnung anhand der Prozessorgeschwindigkeit ist ein oft verwendeter Ansatz, um trotz Heterogenität Arbeitslast günstig zu verteilen [9].

## Poly-Algorithmen

Eine Studie verschiedener Implementierungen der Algorithmen von Fox und Cannon sowie unterschiedlichster Modifikationen (z. B. Broadcast-Broadcast) kommt zu dem Ergebnis, dass es keinen Algorithmus für eine Matrixmultiplikation gibt, der allein immer die besten Resultate für beliebige Probleminstanzen erzielt [69]. Stattdessen stellen die Autoren einen Poly-Algorithmus (Sammlung von Algorithmen) vor, der einen effizienten Algorithmus für eine bestimmte Matrixgröße und ein Prozessorgitter aus der Menge der Algorithmen auswählt. Darüber hinaus gibt es auch hybride Implementierungen der Matrixmultiplikation, die expliziten Datenaustausch und Thread-Programmierung verbinden, um eine geringe Kommunikationszeit zu erzielen. Des Weiteren wurde von Nasri et al. ein weiterer Poly-Algorithmus [74] vorgeschlagen, welcher für eine gegebene Konfiguration aus Matrixdimensionen und Hardware den besten Algorithmus auswählt, wobei auch der Algorithmus von Strassen in die Betrachtungen einbezogen wird.

## Parallele Implementierungen von Strassens Algorithmus

Neben dem in dieser Arbeit vorgeschlagenen Verfahren gibt es bereits unterschiedliche Ansätze, um Strassens Matrixmultiplikation parallel oder gemischt-parallel zu implementieren.

tieren. Grayson et al. entwickelten einen verteilt arbeitenden Algorithmus von Strassen, welcher auf der untersten Stufe auf GEMM zurückgreift [50]. Es gibt außerdem Arbeiten, welche das Verfahren nach Fox (BMR = broadcast multiply roll) mit Strassens Algorithmus kombinieren [70]. Eine erste echte gemischt-parallele Implementierung von Strassens Algorithmus ist in [35] beschrieben. In dem darin beschriebenen Verfahren werden die beiden Ausgangsmatrizen auf zwei disjunkte Prozessorgitter aufgeteilt. Die benötigten Berechnungen der sieben temporären Matrizen werden jeweils von einer disjunkten Prozessormenge erledigt. Mit der vorgestellten Variante kann maximal eine Rekursionsstufe von Strassen angewendet werden. Danach übernimmt die PDGEMM-Routine aus ScaLAPACK die restliche Berechnung. Song et al. haben ebenso eine gemischt-parallele Implementierung von Strassens Algorithmus vorgeschlagen [99]. In dieser Arbeit wird der Algorithmus von Strassen mittels Workflow-Beschreibung von Netsolve [5] modelliert. Mit diesem Ansatz lassen sich mehrere Rekursionen von Strassen anwenden. Nach dem Ende der Rekursion (*cut-off level*) wird auch hier PDGEMM verwendet. Das Verfahren benutzt 15 Tasks, sieben Multiplikationen und acht Additionen, um die vier Ergebnismatrizen  $C_{ij}$ ,  $1 \leq i, j \leq 2$  zu berechnen.

## 2.3 Taskparallele Matrixmultiplikation (tpMM)

Ein erstes Ziel dieser Arbeit bestand in der Realisierung einer Multiplikation zweier dicht-besetzter Matrizen als gemischt-paralleles Programm. Dabei wurde der Algorithmus tpMM (*task parallel matrix multiplication*) entwickelt, dessen Arbeitsweise im folgenden Abschnitt genauer beschrieben wird.

### 2.3.1 Matrixmultiplikation

Das Produkt  $C = A \cdot B$  zweier Matrizen  $A \in R^{m \times k}$  und  $B \in R^{k \times n}$  ist definiert als:

$$c_{ij} = \sum_{l=1}^k a_{il}b_{lj} \quad \text{mit } 1 \leq i \leq m, 1 \leq j \leq n, \quad (2.1)$$

wobei  $c_{ij}$ ,  $a_{il}$ ,  $b_{lj}$  die Elemente der Matrizen  $C$ ,  $A$  und  $B$  bezeichnen.

### 2.3.2 Idee des parallelen Algorithmus

Die Realisierung eines gemischt-parallelen Algorithmus beginnt mit der Definition der Tasks. Dazu muss zuerst die grundlegende Struktur der Berechnung herausgearbeitet werden. Da es sich um Multiprozessor-Tasks handelt, müssen vor allem die zur Verfügung stehenden Prozessoren bei diesen Betrachtungen einbezogen werden. Die Abb. 2.1 veranschaulicht das Basismuster des tpMM-Algorithmus. In der oberen Zeile ist dargestellt, dass die gesamte Matrix  $C$  berechnet werden kann, wenn allen  $p$  Prozessoren (Menge  $\mathcal{P} = \{P_0, \dots, P_{p-1}\}$ ) die kompletten Eingabematrizen  $A$  und  $B$  zur Verfügung stehen. Da die Matrizen verteilt gespeichert sind, muss die Matrix  $C$  schrittweise berechnet werden. Dazu wird, wie in Zeile 2 verdeutlicht, die Matrix  $A$  in zwei Zeilenblöcke und die

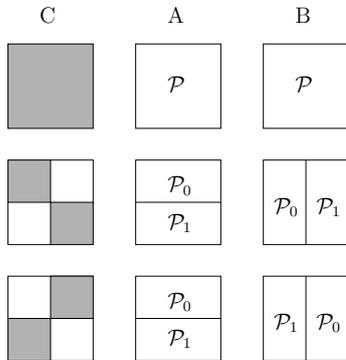


Abb. 2.1: Berechnungsmuster von tpMM.

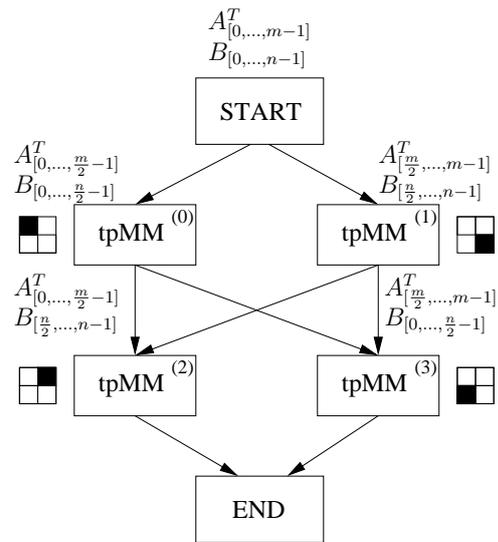
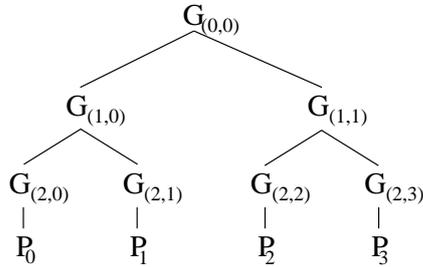


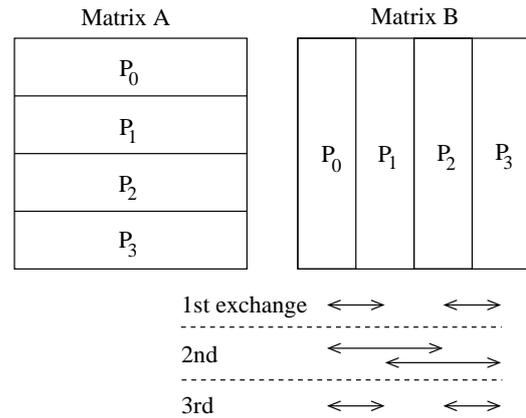
Abb. 2.2: Taskgraph für die Berechnung der Matrix  $C$  mit tpMM.

Matrix  $B$  in zwei Spaltenblöcke zerlegt. Dabei wird der obere Block der Matrix  $A$  auf der Prozessorgruppe  $\mathcal{P}_0 = \{P_0, \dots, P_{\frac{p}{2}-1}\}$  verteilt gespeichert und der untere Block von  $\mathcal{P}_1 = \{P_{\frac{p}{2}}, \dots, P_{p-1}\}$ . Die Zuordnung der Spaltenblöcke von  $B$  ist analog. Mit dieser Aufteilung könnte die Prozessorgruppe  $\mathcal{P}_0$  die linke obere Teilmatrix von  $C$  berechnen und die Gruppe  $\mathcal{P}_1$  die rechte untere. Würden danach die Prozessorgruppen  $\mathcal{P}_0$  und  $\mathcal{P}_1$  ihre Spaltenblöcke von  $B$  vertauschen, ließen sich auch die Blöcke auf der Gegendiagonalen von  $C$  berechnen. Wenn die Prozessorgruppen mehr als einen Prozessor enthalten, kann das Schema weiter rekursiv angewendet werden.

Daraus ergibt sich der Taskgraph aus Abb. 2.2, der von den Prozessoren abstrahiert. Die Eingabe von tpMM sind zwei verkettete Matrizen  $A$  und  $B$ , wobei  $A$  aus  $m$  Zeilen besteht und  $B$  aus  $n$  Spalten. Verkettet sind beide über die Dimension  $k$ , d. h. die Matrix  $A$  besitzt  $k$  Spalten und die Matrix  $B$  hat  $k$  Zeilen. Im ersten Schritt werden die Matrizen  $A$  und  $B$  halbiert und auf die zwei Multiplikationstasks  $\text{tpMM}^{(0)}$  und  $\text{tpMM}^{(1)}$  aufgeteilt. Dafür wird die Matrix  $A$  in zwei Blöcke mit jeweils  $m/2$  Zeilen (dargestellt als  $A^T$ ) und die Matrix  $B$  in zwei Spaltenblöcke der Größe  $n/2$  geteilt. Diese Teilblöcke werden jeweils einer Task zugewiesen, womit die Diagonalblöcke von  $C$  berechnet werden können. Anschließend vertauschen die beiden Matrixmultiplikationstasks der oberen Stufe die Spaltenblöcke von  $B$  und rufen tpMM rekursiv auf. Nach Beendigung der unteren Stufe sind demnach alle Elemente der Matrix  $C$  berechnet worden. Werden die Taskgruppen  $\{\text{tpMM}^{(0)}, \text{tpMM}^{(2)}\}$  sowie  $\{\text{tpMM}^{(1)}, \text{tpMM}^{(3)}\}$  jeweils auf unterschiedliche SMP-Knoten abgebildet, muss nur ein einziger Austausch zwischen diesen Knoten stattfinden. Alle anderen Kommunikationsoperationen könnten innerhalb des Knotens ausgeführt werden können.



**Abb. 2.3:** Aufbau der Gruppenhierarchie für 4 Prozessoren.



**Abb. 2.4:** Kommunikationsmuster von tpMM für 4 Prozessoren.

### 2.3.3 Voraussetzungen

Der Algorithmus tpMM implementiert eine Matrixmultiplikation  $C = A \times B$ , wobei  $A \in R^{m \times k}$ ,  $B \in R^{k \times n}$ ,  $C \in R^{m \times n}$ . Das angegebene Verfahren arbeitet dabei mit  $p = 2^i$ ,  $i \geq 0$  Prozessoren. Außerdem wird gefordert, dass die Dimensionen  $m$  und  $n$  ein Vielfaches von  $p$  sind, also  $m = p \cdot j \wedge n = p \cdot l$ ,  $j \geq 1$ ,  $l \geq 1$ .

Die Prozessoren werden hierarchisch in Gruppen  $G_{(u,v)}$ ,  $0 \leq u \leq \log p$ ,  $0 \leq v < 2^u$  zusammengefasst: Die größte (oberste) Gruppe  $G_{(0,0)}$  enthält alle Prozessoren. Diese Gruppe setzt sich aus den Untergruppen  $G_{(1,0)}$  und  $G_{(1,1)}$  zusammen, welche die Prozessoren  $P_0, \dots, P_{p/2-1}$  und  $P_{p/2}, \dots, P_{p-1}$  enthalten. Diese Aufteilung wird rekursiv fortgeführt bis jede Gruppe nur noch einen Prozessor enthält. Einer Gruppe  $G_{(u,v)}$  sind demnach die Prozessoren mit den Nummern  $v \cdot p/2^u$  bis  $(v+1) \cdot p/2^u - 1$  zugeordnet. Ein Beispiel der Gruppenbildung mit 4 Prozessoren ist in Abb. 2.3 dargestellt.

Die Matrizen  $A$  und  $B$  werden blockweise auf die Prozessoren verteilt (*1D block layout* oder *1D block mapping* [49]), wobei Matrix  $A$  in Zeilenblöcke ( $m/p \times k$ ) und Matrix  $B$  in Spaltenblöcke ( $k \times n/p$ ) aufgeteilt ist. Eine schematische Darstellung der Zerlegung mit vier Prozessoren ist in Abb. 2.4 veranschaulicht.

### 2.3.4 Beschreibung des parallelen Algorithmus

Der Pseudocode von tpMM zur verteilten Berechnung von  $C = A \times B$  wird in Alg. 2.1 veranschaulicht. Die Funktion `tpmm` besitzt drei Argumente: die aktuelle Prozessorgruppe  $G_{(u,v)}$  (charakterisiert die Hierarchiestufe), den Spaltenindex  $j$  der Matrix  $C$  und die Dimension  $n$  der betrachteten Teilmatrix. Zu Beginn der Berechnung wird `tpmm` mit den Werten  $G_{(0,0)}$ ,  $j = 0$  und  $n = k$  aufgerufen.

Die Funktion überprüft zuerst, ob die aktuelle Gruppe nur einen Prozessor enthält. Ist dies der Fall wird eine lokale Matrixmultiplikation z. B. durch `xGEMM` durchgeführt<sup>1</sup>. Diese lokale Multiplikation ist ein so genanntes  $k$ -Panel-Update, da die Teilblöcke der Matrizen

<sup>1</sup>x in `xGEMM` bezeichnet den Datentyp (d)ouble oder (f)loat.

**Algorithmus 2.1** tpMM

---

```

function tpmm( Group  $G_{(u,v)}$ , Column  $j$ , Dimension  $n$  )
1: if  $|G_{(u,v)}| == 1$  then
2:   perform local matrix update (xGEMM)
3:   return
4: let  $G_1 \leftarrow G_{(u+1,2 \cdot v)}$  /* create sub-groups */
5: let  $G_2 \leftarrow G_{(u+1,2 \cdot v+1)}$ 
6: if  $pid \in G_1$  then /* task-parallel */
7:   tpmm(  $G_1, j, \frac{n}{2}$  )
8: else
9:   tpmm(  $G_2, j + \frac{n}{2}, \frac{n}{2}$  ) /* task-parallel */
10: exchangeB(  $G_1, G_2$  )
11: if  $pid \in G_1$  then /* task-parallel */
12:   tpmm(  $G_1, j + \frac{n}{2}, \frac{n}{2}$  )
13: else
14:   tpmm(  $G_2, j, \frac{n}{2}$  ) /* task-parallel */

```

---

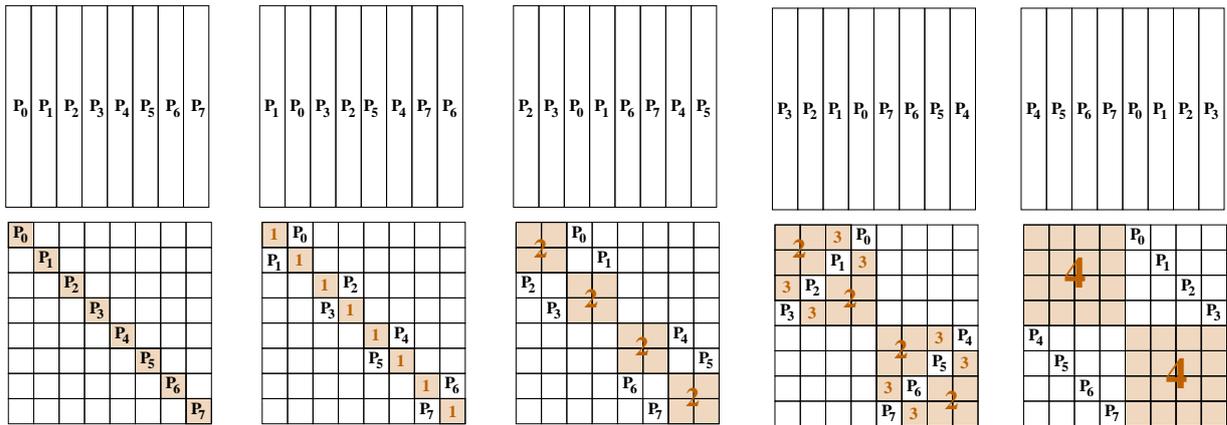
$A$  und  $B$  die Größe  $m/p \times k$  respektive  $k \times n/p$  haben. Die verkettete Dimension ist demnach  $k$ . Mit diesen beiden  $k$ -Panels können dann  $m/p \times n/p$  Elemente der Matrix  $C$  berechnet werden.

Sind der Task mehrere Prozessoren zugeordnet, wird die zugehörige Gruppe in zwei gleich große Teilgruppen zerlegt. Die erste Teilgruppe  $G_1$  berechnet danach die  $n/2$  Spalten der Matrix  $C$  beginnend bei Spalte  $j$ . Die andere Gruppe  $G_2$  berechnet die andere Hälfte von  $C$  beginnend bei Spalte  $j + n/2$ . Nach dieser Berechnung sind die linke obere und die rechte untere Teilmatrix von  $C$  fertiggestellt. Im Anschluss werden die Spalten von  $B$  zwischen den Gruppen  $G_1$  und  $G_2$  ausgetauscht. Nachdem jede Prozessorgruppe einen neuen Block von  $B$  erhalten hat, können die beiden restlichen Teilblöcke von  $C$  vervollständigt werden.

Die Abb. 2.5 verdeutlicht die Arbeitsweise auf Prozessorebene bei 8 Prozessoren. In der oberen Zeile ist die jeweilige Aufteilung der Elemente von Matrix  $B$  auf die Prozessoren dargestellt. Darunter sind von links nach rechts vier Berechnungsschritte der Matrix  $C$  angegeben. Nach dem ersten lokalen Matrixupdate sind die Diagonalblöcke von  $C$  fertig gestellt. Die Prozessoren einer Zweiergruppe teilen danach untereinander die Spalten von  $B$  auf. Damit lassen sich im zweiten Schritt größere Blöcke auf der Diagonalen berechnen. Nach  $p$  lokalen Matrixmultiplikationen und  $(p - 1)$  Vertauschoperationen ist die Berechnung der Matrix  $C$  abgeschlossen.

### 2.3.5 Vorteile von tpMM

Die Anzahl an Kommunikations- und Berechnungsoperationen von tpMM ist identisch mit anderen Panel-Panel-Ansätzen, z. B. der Ring-Methode, bei der die Teilblöcke von  $B$  ringförmig zwischen den Prozessoren vertauscht werden, siehe Abschnitt 2.4.4. Allerdings unterscheidet sich tpMM im angewandten Kommunikationsmuster. Ein Beispiel des Musters für 4 Prozessoren ist in Abb. 2.4 dargestellt. Diese Grafik zeigt, dass tpMM bei



**Abb. 2.5:** Beispiel der Berechnungsreihenfolge der Matrix  $C$  von tpMM für 8 Prozessoren. Die farbigen Zahlen geben den Berechnungsschritt an, in dem der markierte Block berechnet wurde.

zwei von drei Kommunikationsschritten ohne Netzwerkverkehr auskommen kann, falls die Prozessoren  $P_0$  und  $P_1$  sowie  $P_2$  und  $P_3$  auf jeweils einen Dual-SMP-Knoten abgebildet werden.

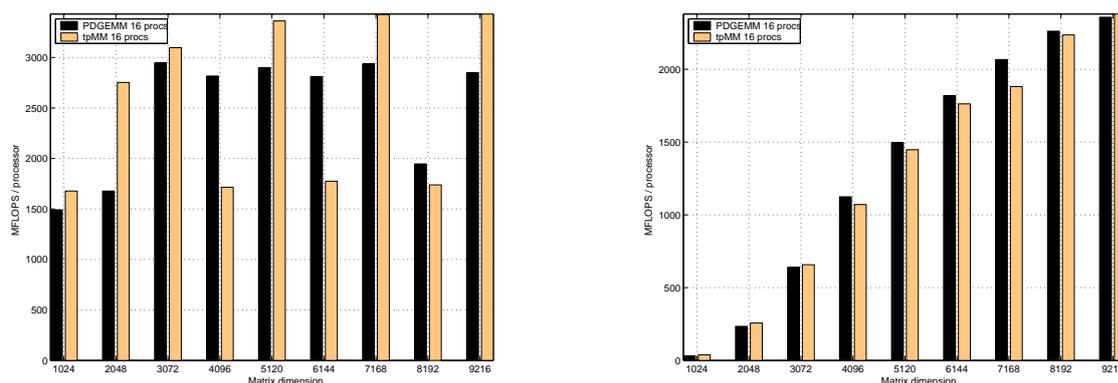
Allgemein gilt für tpMM: Die Anzahl der Kommunikationsschritte  $o_{comm}^{SMP}(p, m)$ , die ein Prozessor ohne Netzwerkkommunikation durchführen kann, lässt sich wie folgt abschätzen: Ein Cluster sei aus baugleichen SMP- oder Multicore-Systemen mit jeweils  $m \geq 2$  Prozessoren pro Mainboard aufgebaut. Da tpMM eine Kommunikationsstruktur eines binären Baums aufweist, können maximal  $\log m$  Ebenen des Baumes auf einen SMP-Knoten abgebildet werden. Diese Ebenen werden aber mehrfach durchlaufen, da sie sich am Ende der Rekursionstiefe befinden. Nach den Berechnungen auf Ebene  $l \geq 1$  werden Tauschoperationen auf der Ebene  $l + 1$  durchgeführt, wobei  $l = \log p$  die Wurzel des Baumes bezeichnet. Für jede höher liegende Ebene von  $l$  wird ein Tausch ausgeführt und die Berechnung rekursiv gestartet. Insgesamt gibt es  $\log p - l$  Ebenen oberhalb von  $l$ . Daraus folgt:

$$o_{comm}^{SMP}(p, m) \geq \sum_{i=1}^{\log m} 2^{\log p - i}, m \leq p. \quad (2.2)$$

Angenommen, ein Cluster besteht aus 8 Knoten mit jeweils 4 Prozessoren. Aus  $p = 32$  und  $m = 4$  folgt  $o_{comm}^{SMP}(32, 4) = 24$ . Da es insgesamt  $p - 1 = 31$  Kommunikationsschritte gibt, können bei optimalem Mapping ca. 77% innerhalb der Knoten ausgeführt werden. Würde man die Blöcke von  $B$  z. B. ringförmig zwischen den Prozessoren austauschen, würden alle 31 Kommunikationsschritte eine Netzwerkkommunikation benötigen.

### 2.3.6 Experimentelle Auswertung

Die praktische Leistungsfähigkeit des tpMM-Algorithmus wurde durch eine Reihe von Experimenten auf unterschiedlichen parallelen und verteilten Systemen evaluiert. Als Vergleichsroutine wurde PDGEMM aus der ScaLAPACK-Bibliothek herangezogen. Die Funk-



**Abb. 2.6:** Leistungsvergleich in MFLOPS/Prozessor von PDGEMM und tpMM mit 16 Prozessoren auf IBM Regatta p690 (links) und XEON-SCI-Cluster.

tionsweise und die interne Realisierung von PDGEMM werden detailliert in Abschnitt 2.5 beschrieben.

Als Testsysteme wurden die IBM Regatta p690 und der XEON-SCI-Cluster verwendet (siehe Anhang B). Da es sich bei beiden Rechnern um SMP-Cluster handelt, sollte dort die günstige Kommunikationsstruktur von tpMM zum Tragen kommen. Die Laufzeitergebnisse für beide Systeme mit jeweils 16 Prozessoren sind in Abb. 2.6 dargestellt. Als Maß für die Leistung der Algorithmen wird in den Grafiken MFLOPS pro Prozessor verwendet. Mit diesem Maß können Aussagen über die Skalierbarkeit und die Effizienz von verteilt arbeitenden Algorithmen getroffen werden. Andere gängige Maß wie der Speedup lassen sich wegen der Größe der Eingabedaten nicht berechnen, da sie oft nicht in den Speicher eines einzelnen Prozessors passen. Für den XEON-SCI-Cluster erreichen beide Algorithmen, tpMM und PDGEMM, für alle Matrixgrößen ähnliche Resultate. Für größere Matrizen ist ein kleiner Vorsprung für tpMM erkennbar. Dieser resultiert vor allem aus dem besseren Verhältnis von Kommunikations- und Berechnungszeit. Je größer die Matrix wird, umso mehr profitiert tpMM durch die geringe Anzahl von Kommunikationsoperationen über das Netzwerk. Auch für die IBM Regatta p690 konnten sehr gute Resultate erzielt werden. Hier sind allerdings bei ein paar Matrixgrößen (4096, 6144) Leistungseinbrüche festzustellen. Diese liegen in einer geringen Leistung der DGEMM-Routine aus ESSL (von IBM) begründet. Für manche Matrixgrößen lieferten lokale Berechnungen mit DGEMM sehr schlechte MFLOPS-Raten bei der Verwendung von ESSL. Sehr wahrscheinlich wird bei den verwendeten Parametern ein ungünstiger lokaler Kernel gewählt, der zu vielen Zugriffsfehlern im Daten-Cache führt. Da die Bibliothek nicht quelloffen ist, kann das Problem auch nicht näher untersucht werden.

### 2.3.7 Fazit tpMM

Der Algorithmus tpMM war ein erster Versuch, um ein gängiges Verfahren durch die Definition von Multiprozessor-Tasks gemischt-parallel zu realisieren. Dazu wurde ein Taskgraph erzeugt, der die algorithmische Struktur abbildet, wobei die Datenabhängigkeiten durch Kanten repräsentiert sind. Nach Definition der Tasks wurde eine Verteilungsstrategie der

Matrizen gewählt und eine Abbildung der Prozessorengruppen auf die Tasks festgelegt. Der tpMM-Algorithmus versucht durch ein günstiges Berechnungsmuster, möglichst viele Kommunikationsoperationen auf einem SMP-Knoten durchzuführen. Die experimentellen Resultate haben gezeigt, dass tpMM auch mit anderen algorithmischen Implementierungen der Matrixmultiplikation wie PDGEMM mithalten kann. tpMM eignet sich am besten, wenn große SMP-Systeme über ein langsames Netzwerk verbunden sind, idealerweise mit einer Baumstruktur, da dies der Kommunikationsstruktur entspricht.

## 2.4 Mehrstufige Realisierung des Algorithmus von Strassen

### 2.4.1 Motivation

Neben der hierarchischen Formulierung eines  $k$ -Panel-Updates (tpMM) wurde auch eine task- und gemischt-parallele Implementierung des Algorithmus von Strassen angestrebt. Es soll dabei untersucht werden, wie sich Strassens Algorithmus effizient in Tasks zerlegen lässt und ob die Leistung eines solchen Verfahrens mit gängigen Implementierungen von verteilten Matrixmultiplikationsalgorithmen (z. B. PDGEMM) mithalten kann. Wie oben beschrieben existieren verschiedene Ansätze einer taskparallelen Implementierung des Strassen-Algorithmus. Der hier vorgestellte Ansatz unterscheidet sich von diesen verwandten Arbeiten wie folgt:

- Die Formulierung der Tasks ist ergebnis- und nicht systemorientiert, d. h. die Tasks sollen möglichst unabhängig von der Anzahl der zu verwendeten Prozessoren sein.
- Da der Strassen-Algorithmus rekursiv definiert ist, soll auch die taskparallele Formulierung rekursiv anwendbar sein.
- Beim Erreichen der Cut-off-Rekursionstiefe sollen unterschiedliche Verfahren zur verteilten Matrixmultiplikation zum Einsatz kommen. Je nach verwendetem Algorithmus ist in der Cut-off-Tiefe ein unterschiedliches Datenlayout der Matrizen erforderlich, welches die Geschwindigkeit des Gesamtverfahrens maßgeblich beeinflussen kann.

### 2.4.2 Sequenzieller Algorithmus

Im Jahr 1969 veröffentlichte Volker Strassen einen Artikel, in dem er das folgende Verfahren zur Multiplikation zweier quadratischer Matrizen angab [47, 100]:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad (2.3)$$

$C_{11}$ ,  $C_{12}$ ,  $C_{21}$ , und  $C_{22}$  erhält man wie folgt:

$$\begin{aligned}
 Q_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
 Q_2 &= (A_{21} + A_{22})B_{11} \\
 Q_3 &= A_{11}(B_{12} - B_{22}) \\
 Q_4 &= A_{22}(B_{21} - B_{11}) \\
 Q_5 &= (A_{11} + A_{12})B_{22} \\
 Q_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
 Q_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
 \end{aligned} \tag{2.4}$$

und

$$\begin{aligned}
 C_{11} &= Q_1 + Q_4 - Q_5 + Q_7 \\
 C_{12} &= Q_3 + Q_5 \\
 C_{21} &= Q_2 + Q_4 \\
 C_{22} &= Q_1 + Q_3 - Q_2 + Q_6
 \end{aligned} \tag{2.5}$$

Der Vorteil dieser Methode ist die geringere Anzahl auszuführender Operationen ab einer bestimmten Matrixgröße. Angenommen die Eingabematrizen haben Größe  $n \times n$  und es wird nur eine Rekursion von Strassens Algorithmus durchgeführt. Dann werden 18 Matrixadditionen und 7 Matrixmultiplikationen mit Matrizen der Größe  $m = n/2$  durchgeführt. Die Standardmatrixmultiplikation zweier Matrizen aus  $R^{n \times n}$ -Matrizen kann in

$$2n^3 - n^2 \tag{2.6}$$

Berechnungsschritten ausgeführt werden (jeweils eine Addition und eine Multiplikation<sup>2</sup>). Die Matrixmultiplikation nach Strassen mit einem Rekursionsschritt benötigt dann

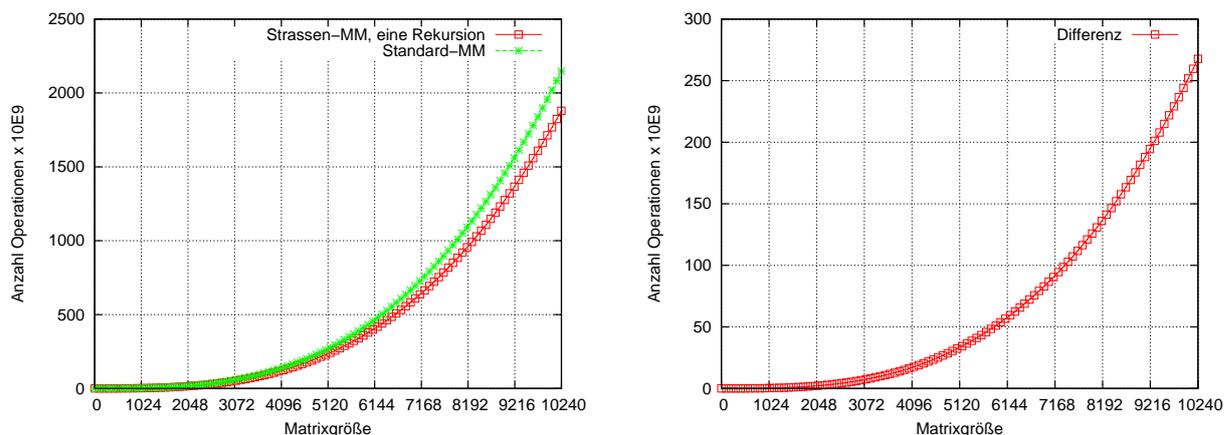
$$7(2m^3 - m^2) + 18m^2 = 7 \cdot 2 \left(\frac{n}{2}\right)^3 + 11 \left(\frac{n}{2}\right)^2 = \frac{7}{8}(2n^3) + \frac{11}{4}n^2 \tag{2.7}$$

Operationen. Die Abb. 2.7 verdeutlicht den Unterschied der Berechnungskomplexität beider Verfahren. In der linken Grafik wird die Gesamtanzahl der auszuführenden Operationen für beide Algorithmen verglichen, während die rechte Grafik die Differenz der Operationen veranschaulicht. Es ist sehr gut zu erkennen, dass der Algorithmus von Strassen speziell bei großen Matrizen einen Vorteil bringt. Man darf bei den Betrachtungen jedoch nicht vergessen, dass die Multiplikation nach Strassen auch einen erheblich höheren Speicherverbrauch, bedingt durch die Rekursionsstufen, nach sich zieht.

### 2.4.3 Definition der Multiprozessor-Tasks

Die Task-Spezifikation von M-Task-Programmen spielt für die zu erwartende Leistung eine wichtige Rolle, denn sie entscheidet über den Kommunikations- und Datenoverhead,

<sup>2</sup>Beim ersten Update von  $C_{ij}$  ist keine Addition notwendig. Darum werden  $n^2$  abgezogen.



**Abb. 2.7:** Links: Vergleich der Anzahl der auszuführenden Operationen mit Strassens Algorithmus und mit der Standardmethode; Rechts: Differenz für jede Matrixdimension.

über die Granularität und damit über die Skalierbarkeit. Für eine gemischt-parallele Implementierung des Strassen-Algorithmus standen verschiedene Möglichkeiten zur Auswahl. Eine Möglichkeit wäre, allen Operationen wie Multiplikationen und Additionen eine eigene Task zuzuordnen. Dadurch gewinnt das Programm einerseits an potenzieller Parallelität, denn die Granularität wird erhöht. Andererseits erhöht sich die Komplexität, die sich vor allem beim Scheduling (Mapping der Tasks auf Prozessorengruppen) bemerkbar macht. Außerdem müssen beim Scheduling die Kommunikationskosten berücksichtigt werden.

Deshalb wurde in dieser Arbeit ein anderer Weg zur Aufteilung der Matrixmultiplikation nach Strassen in Tasks gewählt. Ausgehend von der Berechnung der vier Teilmatrizen  $C_{ij}, 1 \leq i, j \leq 2$  werden genau vier M-Tasks  $T_{C_{ij}}$  erzeugt, denen jeweils eine Berechnung einer Teilmatrix zugeordnet wird. Somit werden in den hier vorgestellten Varianten in jeder Rekursionstufe der Matrixmultiplikation nach Strassen vier M-Tasks erzeugt, welche die Berechnung der Gesamtmatrix  $C$  übernehmen. Entscheidend für die Lastbalancierung und damit für die Leistung ist die Frage, wie die sieben Teilberechnungen  $Q_i$  den vier M-Tasks zugeordnet werden (siehe Gleichung (2.4)). Um diese Frage zu beantworten, sind einige Annahmen erforderlich. Es wird vorausgesetzt, dass die Prozessoren, die den Tasks  $T_{C_{ij}}$  zugeordnet werden, auch die dem Quadranten entsprechenden Teilmatrizen speichern. Beispielsweise werden die Matrizen  $C_{11}, A_{11}, B_{11}$  von der Prozessorgruppe gespeichert, die auch der Task  $T_{C_{11}}$  zugeordnet werden soll.

**Schema-Q7** Eine mögliche Zuordnung der Teilberechnungen  $Q_i$  zu den Tasks  $T_{C_{ij}}$  ist das Schema-Q7, welches in Tab. 2.1 dargestellt ist. Die Zuteilung der  $Q_i$  zu den vier M-Tasks ist dabei so gewählt, dass der Kommunikationsoverhead gegenüber allen anderen Varianten minimiert und gleichzeitig eine gute Lastverteilung erreicht wird. Wie in diesem Schema zu erkennen, werden den ersten drei Tasks zwei und der letzten Task genau eine Teilberechnung ( $Q_6$ ) zugeordnet. Daraus ergibt sich ein weiteres Problem: wie viele Prozessoren werden jeder Task zugewiesen. Wenn man jeder Task ein Viertel der verfügbaren Prozessoren zuweisen würde, entstünde eine Ungleichverteilung der Arbeitslast, da

**Tab. 2.1:** Zuordnung der  $Q_i$  zu den M-Tasks  $T_{C_{ij}}$  für beide Schemata.

Schema-Q7				Schema-Q8			
$T_{C_{11}}$	$T_{C_{12}}$	$T_{C_{21}}$	$T_{C_{22}}$	$T_{C_{11}}$	$T_{C_{12}}$	$T_{C_{21}}$	$T_{C_{22}}$
$Q_1$	$Q_3$	$Q_4$	$Q_6$	$Q_1$	$Q_3$	$Q_4$	$Q_1$
$Q_7$	$Q_5$	$Q_2$		$Q_7$	$Q_5$	$Q_2$	$Q_6$

die Gruppe von Task  $T_{C_{22}}$  nur eine Teilmatrix berechnet. Deshalb werden jeder Task in Schema-Q7 nur so viele Prozessoren zugeordnet, wie sie im Verhältnis zu den anderen arbeiten muss. Demnach bekommen die Tasks  $T_{C_{11}}$ ,  $T_{C_{12}}$  und  $T_{C_{21}}$  jeweils 2/7 der Prozessoren zugeteilt und der Task  $T_{C_{22}}$  wird 1/7 der Prozessoren zur Verfügung gestellt. Aus der Aufteilung der Matrizen auf die Prozessorgruppen ergibt sich die interne Realisierung der M-Tasks für Schema-Q7, welche in Tab. 2.2 veranschaulicht wird.

**Schema-Q8** Das oben beschriebene Schema-Q7 hat auch Nachteile. Die Zuordnung der Prozessoren zu den Tasks ist bei diesem Schema dann am besten, wenn die Anzahl der Prozessoren ein Vielfaches von sieben ist. Ist dies nicht der Fall, müssen die restlichen Prozessoren ( $p \bmod 7$ ) auf die Tasks  $T_{C_{ij}}$  aufgeteilt werden, wodurch eine ungleichmäßige Arbeitslastverteilung entsteht. Ein anderes Problem sind die durch die Prozessorzuteilung entstehenden Kommunikationskosten. Einerseits müssen Teilmatrizen zwischen Prozessorgruppen unterschiedlicher Größe ausgetauscht werden (z. B. zwischen  $T_{C_{11}}$  und  $T_{C_{22}}$ ). Die unterschiedliche Größe führt dabei zu höheren Kommunikationskosten, da ein Prozessor der kleineren Gruppe Daten nur von jeweils einem Prozessor gleichzeitig empfangen kann, d. h. es muss nacheinander empfangen werden. Andererseits müssen Prozessoren aus der kleineren Gruppe beim Senden mehrfach Daten verschicken, wodurch zusätzliche Latenzzeiten entstehen. Um die Kommunikationskosten zu reduzieren, wird auch Schema-Q8 betrachtet, das auf der rechten Seite in Tab. 2.1 dargestellt ist. Bei dieser Variante wird die Berechnung der Teilmatrix  $Q_1$  redundant ausgeführt. Dadurch erhöht sich zwar der Berechnungsaufwand pro Prozessor im Vergleich zu Schema-Q7, die Kommunikationskosten werden aber deutlich verringert.

Die interne Realisierung der M-Tasks für Schema-Q8 ist in Tab. 2.3 dargestellt. Der geringere Kommunikationsaufwand im Fall von Schema-Q8 lässt sich gut daran erkennen, dass eine Task maximal 8 Sende- oder Empfangsoperationen benötigt. In Schema-Q7 werden dagegen 9 solcher Kommunikationsoperationen innerhalb einer Task aufgerufen. Zudem brauchen in Schema-Q7 einige Kommunikationsoperationen länger als andere, was bei Schema-Q8 nicht der Fall ist.

#### 2.4.4 Algorithmische Bausteine

Die Definition und die interne Realisierung der M-Tasks aus Abschnitt 2.4.3 bilden die Basis der Implementierung der hierarchischen Matrixmultiplikation nach Strassen. Ein wichtiges Ziel dieser Arbeit ist die Kombination verschiedener Algorithmen auf unterschiedlichen Berechnungsstufen. Der Algorithmus von Strassen ist eine gute Wahl zur

**Tab. 2.2:** Interner Aufbau der vier M-Tasks für Schema-Q7.

Task $T_{C_{11}}$	Task $T_{C_{12}}$	Task $T_{C_{21}}$	Task $T_{C_{22}}$
Send $A_{11}$	Recv $A_{11}$	Recv $A_{22}$	Send $A_{22}$
Recv $A_{22}$	Send $A_{11}$	Send $A_{22}$	Recv $A_{11}$
Send $B_{11}$	Recv $B_{22}$	Recv $B_{11}$	Send $B_{22}$
Recv $A_{12}$	Send $A_{12}$	Send $B_{11}$	Recv $B_{11}$
Recv $B_{21}$	Send $B_{12}$	Send $B_{21}$	Recv $B_{12}$
Recv $B_{22}$	Send $B_{22}$	Send $A_{21}$	Recv $A_{21}$
$T_1 = A_{11} + A_{22}$	$T_1 = B_{12} - B_{22}$	$T_1 = B_{21} - B_{11}$	$T_1 = A_{21} - A_{11}$
$T_2 = B_{11} + B_{22}$	Strassen( $Q_3, A_{11}, T_1$ )	Strassen( $Q_4, A_{22}, T_1$ )	$T_2 = B_{11} + B_{12}$
Strassen( $Q_1, T_1, T_2$ )	$T_1 = A_{11} + A_{12}$	$T_1 = A_{21} + A_{22}$	Strassen( $Q_6, T_1, T_2$ )
$T_1 = A_{12} - A_{22}$	Strassen( $Q_5, T_1, B_{22}$ )	Strassen( $Q_2, T_1, B_{11}$ )	
$T_2 = B_{21} + B_{22}$			
Strassen( $Q_7, T_1, T_2$ )			
Send $Q_1$			Recv $Q_1$
Recv $Q_4$	Send $Q_3$	Send $Q_4$	Recv $Q_3$
Recv $Q_5$	Send $Q_5$	Send $Q_2$	Recv $Q_2$
$T_1 = Q_1 + Q_7$	$C_{12} = Q_3 + Q_5$	$C_{21} = Q_2 + Q_4$	$T_1 = Q_1 + Q_6$
$T_1 = T_1 + Q_4$			$T_1 = T_1 + Q_3$
$C_{11} = T_1 - Q_5$			$C_{22} = T_1 - Q_2$

Realisierung der obersten Stufe, da eine Reduzierung der auszuführenden Operationen erreicht wird (siehe Abb. 2.7). Da der Vorteil von Strassens Algorithmus mit zunehmender Matrixgröße wächst, bietet sich die Anwendung von Strassen am Anfang an, also bei den Eingabematrizen. Bei der rekursiven Berechnung mit Strassens Algorithmus nimmt die Größe der Matrizen logarithmisch ab. Dadurch kommen auf unteren Rekursionsstufen auch andere Algorithmen als Kandidaten in Betracht, da der Algorithmus von Strassen auch Nachteile mit sich bringt (erhöhte Kommunikationskosten, erhöhter Speicherbedarf).

Im folgenden Abschnitt werden verschiedene verteilt arbeitende Algorithmen vorgestellt, welche als Bausteine für die hierarchische Implementierung der Matrixmultiplikation benutzt werden.

### tpMM

Der schon in Abschnitt 2.3 beschriebene Algorithmus tpMM ist ein Kandidat für die Berechnung der Teilresultate nach der initialen Anwendung des Algorithmus von Strassen. Der Algorithmus tpMM arbeitet rekursiv mit einer hierarchischen Gruppierung von Prozessoren und implementiert ein  $k$ -Panel-Update. Er wurde speziell für SMP-Cluster entwickelt und ist besonders dann interessant, wenn eine tpMM-M-Task auf SMP-Knoten abgebildet werden kann.

### Ring-Methode

Die Ring-Methode ist wie tpMM zu den Verfahren zu zählen, die eine Sequenz von  $k$ -Panel-Updates durchführen, um das Gesamtergebnis zu berechnen. Demzufolge verwendet die Ring-Methode eine ähnliche Verteilungsstrategie wie tpMM, aber nicht die hierarchische

**Tab. 2.3:** Interner Aufbau der vier M-Tasks für Schema-Q8.

Task $T_{C_{11}}$	Task $T_{C_{12}}$	Task $T_{C_{21}}$	Task $T_{C_{22}}$
Send $B_{11}$	Recv $B_{22}$	Recv $B_{11}$	Send $B_{22}$
Recv $B_{22}$	Send $B_{22}$	Send $B_{11}$	Recv $B_{11}$
Send $A_{11}$	Recv $A_{11}$	Recv $A_{22}$	Send $A_{22}$
Recv $A_{22}$	Send $A_{11}$	Send $A_{22}$	Recv $A_{11}$
Recv $A_{12}$	Send $A_{12}$	Send $A_{21}$	Recv $A_{21}$
Recv $B_{21}$	Send $B_{12}$	Send $B_{21}$	Recv $B_{12}$
$T_1 = A_{11} + A_{22}$	$T_1 = B_{12} - B_{22}$	$T_1 = B_{21} - B_{11}$	$T_1 = A_{11} + A_{22}$
$T_2 = B_{11} + B_{22}$	Strassen( $Q_3, A_{11}, T_1$ )	Strassen( $Q_4, A_{22}, T_1$ )	$T_2 = B_{11} + B_{22}$
Strassen( $Q_1, T_1, T_2$ )	$T_1 = A_{11} + A_{12}$	$T_1 = A_{21} + A_{22}$	Strassen( $Q_1, T_1, T_2$ )
$T_1 = A_{12} - A_{22}$	Strassen( $Q_5, T_1, B_{22}$ )	Strassen( $Q_2, T_1, B_{11}$ )	$T_1 = A_{21} - A_{11}$
$T_2 = B_{21} + B_{22}$			$T_2 = B_{11} + B_{12}$
Strassen( $Q_7, T_1, T_2$ )			Strassen( $Q_6, T_1, T_2$ )
Recv $Q_4$	Send $Q_3$	Send $Q_4$	Recv $Q_3$
Recv $Q_5$	Send $Q_5$	Send $Q_2$	Recv $Q_2$
$T_1 = Q_1 + Q_7$	$C_{12} = Q_3 + Q_5$	$C_{21} = Q_3 + Q_5$	$T_1 = Q_1 + Q_6$
$T_1 = T_1 + Q_4$			$T_1 = T_1 + Q_3$
$C_{11} = T_1 - Q_5$			$C_{22} = T_1 - Q_2$

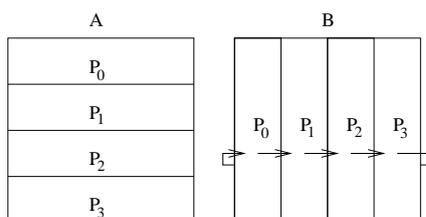
Organisation der Prozessoren. Die Ring-Methode arbeitet mit einer beliebigen Prozessorzahl  $p > 0$ . Sie kann auch verwendet werden, wenn die Anzahl der Prozessoren keine Zweierpotenz ist, also  $p \neq 2^i, i \geq 0$ , die man zur Anwendung von tpMM bräuchte. Die Matrizen  $A$  und  $B$  werden in Zeilenblöcken und in Spaltenblöcken auf die Prozessorgitter von  $A$  und  $B$  verteilt. Die restlichen Zeilen ( $m \bmod p$ ) und Spalten ( $n \bmod p$ ) werden zyklisch auf die Prozessoren abgebildet. Dabei wird aber lediglich die Anzahl der gespeicherten Zeilen oder Spalten pro Prozessor erhöht. Jeder Prozessor speichert weiterhin einen zusammenhängenden Block von  $A$  und  $B$ . Bei der Ring-Methode wechseln sich Berechnungsschritte (Updates der Panels) und Kommunikationsoperationen ab. In jedem Kommunikationsschritt schickt ein Prozessor  $P_i$  sein Panel der Matrix  $B$  der Größe  $k \times m/p$  zu seinem Nachbarn  $P_{(i+1) \bmod p}$ . Das dadurch entstehende Kommunikationsmuster gleicht deshalb einem Ring, vergleiche Abb. 2.8. Nach jedem Kommunikationsschritt kann die lokale Teilmatrix auf  $P_i$  aktualisiert werden. Ein Prozessor  $P_i, 0 \leq i < p$  berechnet mit dieser Methode die folgenden Elemente der Matrix  $C$ :

$$\begin{pmatrix} c_{(i\frac{m}{p}+1)1} & \cdots & c_{(i\frac{m}{p}+1)n} \\ \vdots & \ddots & \vdots \\ c_{(i\frac{m}{p}+1)1} & \cdots & c_{(i\frac{m}{p}+1)n} \end{pmatrix}.$$

Die Berechnung der Matrix  $C$  ist nach  $p$  Berechnungsschritten und  $(p - 1)$  Kommunikationsschritten abgeschlossen.

## PDGEMM

PDGEMM ist eine Funktion der PBLAS (Parallel Basic Linear Algebra Subprograms), welche als Teil des ScaLAPACK-Projekts entwickelt wurde. Die Routinen aus PBLAS sind zu



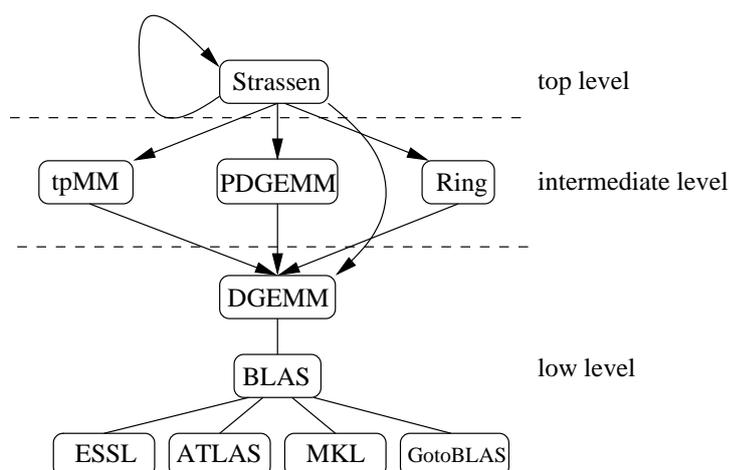
**Abb. 2.8:** Kommunikationsmuster der Ring-Methode.

einem De-facto-Standard für effiziente parallele Algorithmen geworden und sind somit auf den meisten parallelen Systemen verfügbar. Es existieren verschiedene Implementierungen der Schnittstelle, sowohl herstellerspezifische als auch freie Varianten. Demzufolge werden auch unterschiedliche Algorithmen innerhalb einer Implementierung eingesetzt. Diese Arbeit konzentriert sich hauptsächlich auf den Vergleich mit der freien Implementierung von PDGEMM in ScaLAPACK. Falls bei den Experimenten andere Implementierungen zum Einsatz kommen, wird ausdrücklich darauf hingewiesen. Für eine detailliertere Beschreibung der PDGEMM-Routine sei auf Abschnitt 2.5 verwiesen, worin neben den verwendeten Algorithmen auch ein Verfahren zur Optimierung der Geschwindigkeit der Routine angegeben wird.

### Kombination der algorithmischen Bausteine

Die im Folgenden beschriebenen Algorithmen entstehen durch die Kombination der algorithmischen Bausteine gemäß der Berechnungshierarchie aus Abb. 2.9. Auf der obersten Stufe kann der Algorithmus von Strassen mehrere Male rekursiv angewendet werden. Die maximale Anzahl von Rekursionsstufen ist vor allem durch die Anzahl der zur Verfügung stehenden Prozessoren beschränkt. Beim Erreichen des Cut-off-Levels können für weitere verteilte Berechnungen entweder tpMM, Ring oder PDGEMM verwendet werden. Es ist aber auch möglich, dass nach der rekursiven Anwendung von Strassens Algorithmus die entstandene M-Task nur einem Prozessor zugewiesen wurde. In diesem Fall lässt sich direkt die DGEMM-Berechnungsroutine zur Lösung einer Matrixmultiplikation auf einem einzelnen Rechner aufrufen. Die Routine DGEMM wird außerdem von den Algorithmen der mittleren Stufe (*intermediate level*) als Berechnungskern für Einprozessormaschinen eingesetzt. Die Routine DGEMM ist Teil der BLAS (Basic Linear Algebra Subprograms) und wie für PDGEMM gibt es auch für DGEMM Implementierungen unterschiedlicher Hersteller. Im Rahmen dieser Arbeit wurden abhängig von der parallelen Maschine mehrere Implementierungen von BLAS benutzt. Dazu zählen ATLAS für Linux-Cluster, die von Intel bereitgestellte Bibliothek MKL (auf Itanium-Systemen wie der SGI Altix) und die ESSL-Bibliothek, welche von IBM für deren Systeme mitgeliefert wird (IBM Regatta).

In den folgenden Abschnitten werden verschiedene Poly-Algorithmen betrachtet, die durch Anwendung der gegebenen Berechnungshierarchie möglich sind. Für jeden beschriebenen Algorithmus werden die Voraussetzungen an die Datenverteilung und die Prozessanzahl angegeben und erläutert. Da die Poly-Algorithmen den Algorithmus von Strassen in der obersten Berechnungsstufe verwenden, werden im Folgenden quadratische Matrizen aus  $R^{n \times n}$  als Eingabe vorausgesetzt.



**Abb. 2.9:** Kombinationsmöglichkeiten der mehrstufigen Realisierung des Strassen-Algorithmus.

### 2.4.5 Kombination: Strassen und Ring

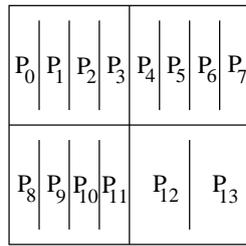
In dieser algorithmischen Kombination soll nach dem Algorithmus von Strassen die Ring-Methode angewendet werden. Diese Kombination wird speziell für Schema-Q7 betrachtet, da mit diesem Schema in den Rekursionstufen Prozessorgruppen mit einer ungeraden Anzahl von Prozessoren entstehen können. Für die Anwendung dieser Kombination ist ein spezielles Datenlayout notwendig, um die Matrizen  $A$  und  $B$  der Größe  $n \times n$  auf die  $p$  Prozessoren zu verteilen. Zuerst werden in jedem Rekursionsschritt von Strassens Algorithmus die Matrizen in vier gleich große, quadratische Untermatrizen aufgeteilt. Jede dieser Untermatrizen wird einer Gruppe von Prozessoren zugeordnet, welche entweder  $2p/7$  oder  $p/7$  der ursprünglichen  $p$  Prozessoren enthält, vgl. Schema-Q7 in Tab. 2.1. Damit die Ring-Methode angewendet werden kann, müssen die Teilmatrizen blockweise – für Matrix  $A$  ( $B$ ) in Blöcken aus Zeilen (Spalten) – auf die Prozessoren verteilt abgelegt sein. Aus diesem Grund ist die Datenverteilung der Eingangsmatrizen mit der Aufrufhierarchie der algorithmischen Kombination verbunden, d. h. unterschiedliche Rekursionstiefen von Strassens Algorithmus verlangen unterschiedliche Datenverteilungen. Abb. 2.10 zeigt ein Beispiel einer Datenverteilung für die Kombination *Strassen+Ring* für 14 Prozessoren.

Um eine möglichst ausgeglichene Datenverteilung und damit bessere Leistung zu ermöglichen, sollte die Anzahl der Prozessoren für *Strassen+Ring* ein Vielfaches von 7 sein. Obwohl dies wegen des algorithmischen Musters eine günstigere Wahl ist, kann die Kombination *Strassen+Ring* mit jeder beliebigen Anzahl von Prozessoren  $p \geq 4$  (mindestens eine Rekursion) verwendet werden.

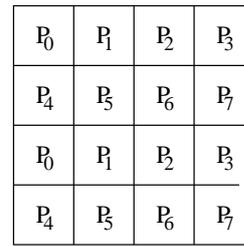
### 2.4.6 Kombination: Strassen und PDGEMM

#### Blockzyklische Datenverteilung, Schema-Q7

Diese Kombination von Strassens Algorithmus und PDGEMM basiert auf einer blockzyklischen Ausgangsverteilung der Matrizen  $A$  und  $B$  auf alle Prozessoren. Ein Beispiel für eine solche blockzyklische Verteilung ist in Abb. 2.11 dargestellt. Um die Daten blockzyklisch



**Abb. 2.10:** Verteilung von 14 Prozessoren auf Matrix  $B$  für die Kombination *Strassen+Ring* (Schema-Q7).



**Abb. 2.11:** Blockzyklische Verteilung der Matrixblöcke auf die Prozessoren (Schema-Q7).

aufzuteilen, ist ein rechteckiges Prozessorgitter notwendig, wenn möglich wird ein quadratisches Prozessorgitter der Größe  $\sqrt{p} \times \sqrt{p}$  bevorzugt. Bedingt durch die Datenverteilung sind im Unterschied zu den anderen hier betrachteten mehrstufigen Kombinationen die Matrizen und Teilmatrizen wie z. B.  $A_{11}$  auf alle teilnehmenden Prozessoren aufgeteilt<sup>3</sup>.

In jeder Rekursionsstufe des Algorithmus von Strassen wird die Gruppe der Prozessoren in vier disjunkte Untergruppen der Größe  $2p/7$  oder  $p/7$  aufgeteilt. Wenn die  $\hat{p}$  Prozessoren einer solchen Untergruppe nicht in einem rechteckigen oder quadratischen Gitter angeordnet werden können, werden die Prozessoren in einem linearen Gitter ( $1 \times \hat{p}$ ) für die Erzeugung des neuen BLACS-Kontextes gruppiert<sup>4</sup>.

Da die Teilmatrizen ( $A_{ij}$  und  $B_{ij}$ ,  $1 \leq i, j \leq 2$ ) über alle Prozessoren verteilt sind, müssen auch alle Prozessoren am initialen Datenaustausch teilnehmen. Damit ist es nicht möglich, dass – wie bei *Strassen+Ring* – der Austausch von Teilmatrizen zwischen den Tasks  $T_{C_{ij}}$ ,  $1 \leq i, j \leq 2$  gleichzeitig stattfinden kann. Das führt zu einem größeren Kommunikationsoverhead im Vergleich zu anderen Kombinationen. Andererseits bietet diese Variante die Möglichkeit, direkt in einer Kette von Algorithmen mit blockzyklischem Layout Verwendung zu finden.

### Blockweise Datenverteilung, Schema-Q8

Ein weiteres zu untersuchendes Verfahren ist die Kombination von Strassens Algorithmus und PDGEMM unter Anwendung von Schema-Q8. Der entscheidende Vorteil von Schema-Q8 ist, dass die Kommunikationskosten geringer sind als bei Schema-Q7. Deshalb wird auch hier eine speziell konfigurierte Datenverteilung vorausgesetzt. Das Hauptziel ist dabei, dass die Intra-Task-Kommunikation (zwischen den  $T_{C_{ij}}$ ) parallel stattfinden kann. Es wird angenommen, dass  $p = p_{cut} \cdot 4^l$ ,  $l \geq 1$  Prozessoren zur Verfügung stehen, wobei  $p_{cut}$  die Anzahl der Prozessoren pro Gruppe nach dem Ende der Strassen-Rekursionen (*Cut-off-Level*) bezeichnet. Die  $p$  Prozessoren werden in einem rechteckigen Prozessorgitter der Größe  $p_r \times p_c = p$  angeordnet und die Matrizen  $A$ ,  $B$  und  $C$  werden in Blöcken der Größe  $n/p_r \times n/p_c$  diesen Prozessoren zugewiesen. Die Abb. 2.12 zeigt exemplarisch eine

<sup>3</sup>Dies gilt nur bei einer entsprechend kleinen Blockgröße.

<sup>4</sup>BLACS ist die von ScaLAPACK verwendete Schnittstelle zur Datenkommunikation. Sie abstrahiert von Low-level-Bibliotheken wie MPI und PVM.

$P_0$   $P_1$	$P_2$   $P_3$
$P_4$   $P_5$	$P_6$   $P_7$
$P_8$   $P_9$	$P_{10}$   $P_{11}$
$P_{12}$   $P_{13}$	$P_{14}$   $P_{15}$

**Abb. 2.12:** Abbildung der Prozessoren auf die Matrizen  $A$ ,  $B$  und  $C$ , um einen Rekursionsschritt von Strassens Algorithmus und anschließend PDGEMM auszuführen (Schema-Q8).

$P_0$	$P_4$	$P_0$	$P_1$	$P_2$	$P_3$	$P_4$	$P_5$	$P_6$	$P_7$
$P_1$	$P_5$	$P_8$	$P_9$	$P_{10}$	$P_{11}$	$P_{12}$	$P_{13}$	$P_{14}$	$P_{15}$
$P_2$	$P_6$								
$P_3$	$P_7$								
$P_8$	$P_{12}$								
$P_9$	$P_{13}$								
$P_{10}$	$P_{14}$								
$P_{11}$	$P_{15}$								

Matrix A                      Matrix B

**Abb. 2.13:** Abbildung der Prozessoren auf die Matrizen  $A$  ( $p'_r = 8$ ,  $p'_c = 2$ ) und  $B$  ( $p_r = 2$ ,  $p_c = 8$ ) zur Anwendung einer Rekursion von Strassens Algorithmus und anschließend tpMM (Schema-Q8).

solche Verteilung für 16 Prozessoren. Die Rekursionstiefe von Strassens Algorithmus ist abhängig von der Anzahl der Prozessoren und kann maximal  $\log_4 p$  betragen, da in jedem Rekursionsschritt die Prozessoren in je vier gleich große Gruppen aufgeteilt werden. Am Cut-off-Level wird die Routine PDGEMM aufgerufen. Sollte die tiefsten Rekursionsstufe nur mit einem Prozessor besetzt sein, kann direkt die Routine DGEMM aufgerufen werden.

### 2.4.7 Kombination: Strassen und tpMM

Um Strassens Algorithmus mit tpMM bei Anwendung von Schema-Q8 zu kombinieren, wird vorausgesetzt, dass für  $i$  auszuführende Rekursionen  $p = 4^i 2^j$ ,  $i \geq 1$ ,  $j \geq 0$  Prozessoren verfügbar sind. Der Faktor  $4^i$  repräsentiert die Erzeugung der vier Prozessorgruppen pro Rekursionsstufe. Zur Anwendung von tpMM sind  $2^j$ ,  $j \geq 0$  Prozessoren notwendig. Das Prozessorgitter wird so gewählt, dass nach der Anwendung von Strassens Algorithmus die Matrix  $A$  ( $B$ ) so in Blöcken aus Zeilen (Spalten) auf die Untergruppe der Prozessoren verteilt ist, wie es von tpMM benötigt wird. Ein Beispiel für die notwendige Datenverteilung zur Anwendung eines Rekursionsschritts von Strassens Algorithmus und anschließender Berechnung durch tpMM mit 16 Prozessoren ist in Abb. 2.13 dargestellt. Das so definierte Prozessorgitter von Matrix  $B$  für  $i$  Rekursionen hat die Dimensionen  $p_r \times p_c$ , wobei  $p_r = 2^i$  und  $p_c = \frac{p}{2^i}$ . Das Muster des Prozessorgitters von Matrix  $A$  ( $p'_r \times p'_c$ ) entspricht einer Drehung des Gitters von  $B$  um  $90^\circ$ , also  $p'_r = p_c$  und  $p'_c = p_r$ .

**Tab. 2.4:** Implementierte mehrstufige Verfahren von Strassens Algorithmus.

Schema-Q7	Schema-Q8
	<i>Strassen+PDGEMM</i> (blockverteilt)
<i>Strassen+PDGEMM</i> (ScaLAPACK)	
<i>Strassen+Ring</i>	
	<i>Strassen+tpMM</i>

### 2.4.8 Anwendbarkeit der mehrstufigen Kombinationen

Wie in den vorherigen Abschnitten dargestellt, hängt die Anwendung eines algorithmischen Bausteins in der Cut-off-Rekursionstiefe von der Anzahl der in dieser Stufe verfügbaren Prozessoren ab. Durch die Aufteilungsstrategie der Prozessoren in jedem Rekursionsschritt von Schema-Q7 ist die Anzahl der Prozessoren häufig ungerade. Deshalb kommt in diesem Schema die Ring-Methode zum Einsatz. Außerdem wird für beide Schemata die Kombination aus Strassens Algorithmus und PDGEMM betrachtet, bei der die Umverteilungsroutinen aus ScaLAPACK zum Einsatz kommen. Für ungerade Prozessorenzahlen  $p_{odd}$  am Cut-off-Level wird für *Strassen+PDGEMM* in Schema-Q7 ein  $1 \times p_{odd}$ -Prozessorgitter gebildet oder wenn möglich ein Gitter der Größe  $\sqrt{p_{odd}} \times \sqrt{p_{odd}}$  für eine bessere Leistung von PDGEMM. Bei Schema-Q8 wird in jedem Rekursionsschritt die Prozessoranzahl geviertelt. Am Cut-off-Level gibt es dann zwei mögliche Konfigurationen: Ist die Anzahl ein Vielfaches von 2, kann PDGEMM angewendet werden. Ist die Anzahl eine Zweierpotenz, kann außer PDGEMM auch tpMM zur Lösung benutzt werden. Die Tab. 2.4 fasst die realisierten algorithmischen Kombinationen für jedes Schema zusammen.

### 2.4.9 Laufzeitmodellierung der mehrstufigen Kombinationen

Im Folgenden wird ein Überblick über die Kostenfunktionen der betrachteten algorithmischen Kombinationen gegeben. Die Kostenfunktionen dienen zur Beschreibung der Laufzeit und können verwendet werden, um eine Vorhersage der Leistungsfähigkeit in Abhängigkeit von der Plattform und den Eingabematrizen zu ermöglichen. Diese Kostenfunktionen verwenden die folgenden Parameter, um die Eigenschaften der Zielplattform zu beschreiben:

- $\lambda$  bezeichnet die Latenz des Netzwerks in Sekunden.
- $\tau$  ist die Anzahl der Floating-Point-Werte in doppelter Genauigkeit (*double precision*), die in einer Sekunde über das Netzwerk übertragen werden können. Diese ist proportional zur inversen Bandbreite des Netzwerks.
- $t_{op}$  bezeichnet die Zeit, um eine arithmetische Operation auf der Zielplattform auszuführen und ist gegeben durch das Inverse der FLOPS (*Floating Point Operations Per Second*). Des Weiteren wird angenommen, dass alle Fließkommaoperationen die gleiche Zeit benötigen, egal ob Addition oder Multiplikation.

Die Kommunikationzeit eines Datentransfers wird als lineare Funktion der Nachrichten-  
größe  $ms$  modelliert:

$$T_{comm}(ms) = \lambda + ms \cdot \frac{1}{\tau} \quad (2.8)$$

Darüber hinaus wird angenommen, dass Datenaustausch zwischen je zwei Paaren von  
Prozessoren gleichzeitig ablaufen kann.

Die Variable  $r$  bezeichnet die Anzahl der Rekursionen von Strassens Algorithmus in der  
obersten Stufe der Berechnungshierarchie. Wie zuvor dargestellt, werden die Prozessoren  
je nach Schema unterschiedlich den Tasks zugewiesen. In Schema-Q7 werden mindestens  
sieben Prozessoren benötigt, um eine balancierte Aufteilung in Gruppen mit  $2p/7$  oder  
 $p/7$  Prozessoren vorzunehmen. Hingegen werden in Schema-Q8 vier gleich große Prozessoren-  
gruppen gebildet. Demzufolge lässt sich die maximale Rekursionstiefe für die gemischt-  
parallele Anwendung von Strassens Algorithmus wie folgt angeben:

$$r \leq \begin{cases} \lfloor \log_7 p \rfloor, & \text{für Schema-Q7} \\ \lfloor \log_4 p \rfloor, & \text{für Schema-Q8.} \end{cases} \quad (2.9)$$

## Basisbausteine

**DGEMM und Matrixaddition** Dem Modell der Kostenfunktion von DGEMM liegt das  
Standardverfahren einer Matrixmultiplikation  $C = A \times B$  mit  $C \in R^{m \times n}$ ,  $A \in R^{m \times k}$ ,  $B \in R^{k \times n}$   
zu Grunde. Für jedes zu berechnende Wertepaar muss eine Multiplikation und eine  
Addition zur Ergebnismatrix ausgeführt werden. Es ist allerdings zu beachten, dass  
DGEMM folgende Funktion implementiert:

$$C := \alpha \cdot A \times B + \beta \cdot C.$$

Das bedeutet, dass die Berechnung jedes Elements  $c_{ij}$  der Matrix  $C$  auch  $n$  Additionen  
erfordert. Daraus ergibt sich für die Laufzeit von DGEMM folgende Abschätzung<sup>5</sup>:

$$T_{dgemm}(m, n, k) = 2 \cdot m \cdot n \cdot k \cdot t_{op}. \quad (2.10)$$

Die Berechnungskosten für die parallele Addition zweier quadratischer Matrizen mit  
Dimension  $n$  auf  $p$  Prozessoren sind:

$$T_{mat\_add}(n, p) = \frac{n^2}{p} \cdot t_{op}. \quad (2.11)$$

**Ring-Methode** Wie in Abschnitt 2.4.4 beschrieben, implementiert die Ring-Methode  
eine Serie von  $k$ -Panel-Updates. Die betrachteten quadratischen Matrizen der Dimension  
 $n \times n$  werden dazu in Blöcke (Panels) der Größe  $n \times n/p$  aufgeteilt. Bei  $p$  Prozessoren sind  
demnach  $p$  Berechnungsschritte notwendig. Bei jedem Berechnungsschritt handelt es sich  
um eine sequenzielle Matrixmultiplikation, welche mit DGEMM berechnet wird:

$$T_{Comp\_ring}(n, p) = p \cdot T_{dgemm} \left( \left\lceil \frac{n}{p} \right\rceil, \left\lceil \frac{n}{p} \right\rceil, n \right). \quad (2.12)$$

<sup>5</sup>Für die Betrachtungen sei  $\alpha = 1$  und  $\beta = 1$  vorausgesetzt.

Nach jedem Berechnungsschritt müssen die Blöcke von  $B$  an den benachbarten Prozessor weitergegeben werden. Dazu sendet jeder Prozessor einen Block an einen Nachbarn und empfängt einen Block von einem anderen Nachbarn. Insgesamt sind  $(p - 1)$  Kommunikationsschritte notwendig, um die Ergebnismatrix  $C$  zu berechnen:

$$T_{\text{Comm\_ring}}(n, p) = 2(p - 1) \cdot \left( \lambda + n \cdot \left\lceil \frac{n}{p} \right\rceil \cdot \frac{1}{\tau} \right). \quad (2.13)$$

**Algorithmus tpMM** Beim tpMM-Algorithmus sind die Matrizen  $A$ ,  $B$  und  $C$  (alle mit Dimension  $n \times n$ ) in Blöcken der Größe  $\frac{n^2}{p}$  auf  $p$  Prozessoren verteilt. Die Ergebnismatrix  $C$  wird berechnet, indem jeder Prozessor eine lokale Matrixmultiplikation durchführt und danach seinen Block der Matrix  $B$  mit dem der Berechnungsstufe zugehörigen Partnerprozessor austauscht. Dieses Verfahren wiederholt sich bis jeder Prozessor alle Elemente von  $B$  gesehen hat. Zusammenfassend lassen sich die Berechnungskosten für tpMM angeben:

$$T_{\text{Comp\_tpMM}}(n, p) = p \cdot T_{\text{dgemm}} \left( \frac{n}{p}, \frac{n}{p}, n \right). \quad (2.14)$$

Wie bei der Ring-Methode werden  $(p - 1)$  Schritte benötigt, damit jeder Prozessor alle Elemente der Matrix  $B$  in die Berechnung einfließen lassen hat. Nach jeder lokalen Matrixmultiplikation wird der aktuelle Block zu einem anderen Prozessor gesendet und von diesem ein neuer empfangen. Die Kommunikationskosten belaufen sich auf:

$$T_{\text{Comm\_tpMM}}(n, p) = 2(p - 1) \cdot \left( \lambda + \frac{n^2}{p} \cdot \frac{1}{\tau} \right). \quad (2.15)$$

## PDGEMM

Das Kostenmodell der Routine PDGEMM bezieht sich auf die von ScaLAPACK verwendete freie Implementierung und basiert zum Teil auf den Funktionen, welche Caron et al. bei ihrer Erweiterung der FAST-Bibliothek entwickelt haben [21]. Die Kosten von PDGEMM für die Matrixmultiplikation zweier quadratischer Matrizen der Dimension  $n \times n$  hängen von der gewählten Blockgröße  $nb$  und dem Prozessorgitter  $p_r \cdot p_c = p$  ab ( $p_r$  bezeichnet die Anzahl der Zeilen und  $p_c$  die Spalten des Gitters). Eine detailliertere Beschreibung der einzelnen Parameter ist in Kapitel 2.5 zu finden. Bei der blockzyklischen Verteilung der Matrizen werden die Matrizen in Blöcken der Größe  $nb \times nb$  auf die Prozessoren aufgeteilt. Die Prozessoren speichern jeweils  $n/p_r \times n/p_c$  Elemente von  $A$  und  $B$ , welche auf  $n/p_r/nb \cdot n/p_c/nb$  Blöcke aufgeteilt sind. In der Berechnungsphase wird dann eine Serie von Panel-Updates ausgeführt. Die Panels der Matrix  $A$  enthalten  $n/p_r \times nb$  Elemente und die der Matrix  $B$  enthalten  $nb \times n/p_c$  Elemente. Insgesamt müssen also  $n/nb$  solcher Panel-Updates von einem Prozessor ausgeführt werden, bis dessen Block der Matrix  $C$  vollständig berechnet wurde. Die Berechnungskosten können damit wie folgt abgeschätzt werden:

$$T_{\text{pdgemm\_comp}}(n, nb, p_r, p_c) = \frac{n}{nb} \cdot T_{\text{dgemm}} \left( \frac{n}{p_r}, \frac{n}{p_c}, nb \right) = \frac{2n^3}{p} \cdot t_{op}. \quad (2.16)$$

Die Kommunikationkosten setzen sich aus den einzelnen Startup-Zeiten der Verbindungen und den Transferzeiten der Matrixblöcke zusammen. Pro Kommunikationsschritt nimmt ein Prozessor in einem Zeilen-Broadcast der Matrix  $A$  und einem Spalten-Broadcast der Matrix  $B$  teil. Bei  $n$  Elementen pro Dimension werden  $n/nb$  Broadcasts ausgeführt. Verwendet die Broadcastimplementierung einen Baumalgorithmus, werden pro Broadcast  $\log p_r$  ( $\log p_c$ ) Nachrichten verschickt. Insgesamt werden pro Prozessor  $n^2/p_r$  Elemente in Zeilenbroadcasts und  $n^2/p_c$  in Spaltenbroadcasts übertragen. Die Kommunikationskosten von PDGEMM betragen:

$$T_{\text{pdgemm\_comm}}(n, p_r, p_c, p) = \left( \log_2 p_r \cdot \frac{n^2}{p_c} + \log_2 p_c \cdot \frac{n^2}{p_r} \right) \cdot \frac{1}{\tau} + \lceil \frac{n}{nb} \rceil \cdot (\log_2 p_r + \log_2 p_c) \cdot \lambda. \quad (2.17)$$

### Kombination: Strassen und Ring

Die Modellierung der Kommunikationskosten dieser algorithmischen Kombination setzt die Betrachtung der M-Task-Struktur von Schema-Q7 voraus. Zuerst werden die notwendigen Transferoperationen der Teilmatrizen  $A_{11}, A_{12}, \dots$  und der Zwischenresultate  $Q_l, 1 \leq l \leq 7$  zwischen den einzelnen M-Tasks betrachtet. Wie in Tab. 2.2 zu sehen, haben dabei die Tasks  $T_{C_{11}}$  und  $T_{C_{22}}$  mit 9 Matrixtransfers die meisten Kommunikationsoperationen zu leisten. Als einen weiteren Faktor für die Abschätzung der Kommunikationskosten ist die Datenverteilung der Matrizen  $A, B$  und  $C$  sowie der Zwischenmatrizen  $Q_l$  einzubeziehen. Die Verteilung der Matrizen innerhalb einer M-Task ist nicht gleichmäßig, da jeder Task unterschiedlich viele Prozessoren zugewiesen sind, z. B. kann sich die Datenverteilung der Tasks  $T_{C_{11}}$  und  $T_{C_{22}}$  unterscheiden. Demzufolge müssen einzelne Prozessoren mehrere Nachrichten schicken, damit der Austausch einer Teilmatrix vollzogen werden kann.

In jedem Rekursionsschritt werden  $2 \cdot \frac{p}{7}$  Prozessoren den Tasks  $T_{C_{11}}, T_{C_{12}}$  und  $T_{C_{21}}$  zugewiesen. Da  $p$  nicht zwangsläufig ein Vielfaches von 7 sein muss, variiert die Anzahl der Prozessoren  $p'$  in diesen Tasks zwischen

$$\lfloor \frac{2}{7} \cdot p \rfloor \leq p' \leq \lceil \frac{2}{7} \cdot p \rceil + 1. \quad (2.18)$$

Der vierten Task werden gemäß Schema-Q7 nur  $\lceil \frac{p}{7} \rceil$  Prozessoren pro Rekursionsschritt zugeteilt. Um die maximale Anzahl an zu sendenden Nachrichten zwischen zwei Gruppen abschätzen zu können, genügt es, die größte und die kleinste Task pro Rekursionsschritt (gemäß Anzahl der Prozessoren) zu vergleichen. Die Anzahl der zu sendenden oder empfangenden Nachrichten bei Verwendung einer Blockverteilung kann mit Hilfe des Quotienten der größten und kleinsten Prozessorengruppe ermittelt werden:

$$\frac{\lfloor \frac{2}{7} \cdot p \rfloor + 1}{\lceil \frac{p}{7} \rceil} \leq \frac{\frac{2}{7}p + 1}{\frac{p}{7}} = 2 + \frac{7}{p} \leq 3, \text{ für } p \geq 7. \quad (2.19)$$

Die Gleichung (2.19) zeigt, dass ein Prozessor maximal 3 Nachrichten zu (von) einer anderen M-Task schicken (empfangen) muss. Damit erhöhen sich die Startup-Kosten für jeden

Matrixtransfer um  $3 \cdot \lambda$ . Mit den oben aufgeführten Teilkosten können die Kommunikationskosten für *Strassen+Ring* mit Schema-Q7 wie folgt rekursiv formuliert werden:

$$T_{\text{Comm\_strassen\_ring}}(n, p, r) = \begin{cases} T_{\text{Comm\_ring}}(n, p) & : \text{ falls } r = 0 \\ \begin{cases} 9 \left( \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{1}{\lceil \frac{p}{7} \rceil} \cdot \frac{1}{\tau} + 3\lambda \right) \\ + \max \left( T_{\text{Comm\_strassen\_ring}} \left( \frac{n}{2}, \lfloor \frac{2p}{7} \rfloor, r - 1 \right), \right. \\ \left. T_{\text{Comm\_strassen\_ring}} \left( \frac{n}{2}, \lceil \frac{p}{7} \rceil, r - 1 \right) \right) \end{cases} & : \text{ falls } r > 0 \end{cases} \quad (2.20)$$

In Schema-Q7 hängen die Berechnungskosten wie die Kommunikationskosten von der Zuteilungsstrategie der Prozessoren zu den M-Tasks  $T_{C_{ij}}$  ab. Demzufolge wird die Berechnungszeit in einem Rekursionsschritt entweder von Task  $T_{C_{22}}$  dominiert, welcher  $1/7$  der Prozessoren zugewiesen werden, oder von den drei anderen Tasks, welche von jeweils  $2/7$  der Prozessoren ausgeführt werden. Betrachtet man die Struktur der einzelnen Tasks in Tab. 2.2 ist zu erkennen, dass Task  $T_{C_{11}}$  mit sieben Additionen und zwei Multiplikationen die meisten Operationen der drei Tasks mit jeweils  $2/7$  Prozessoren zu leisten hat. Demgegenüber steht die Task  $T_{C_{22}}$ , welche pro Rekursionsschritt fünf Additionen und eine Multiplikation ausführen muss. Die Berechnungskosten für *Strassen+Ring* können nur rekursiv formuliert werden, da die Anzahl der Prozessoren pro Rekursionsstufe vom Ausgangswert abhängt.

$$T_{\text{Comp\_strassen\_ring}}(n, p, r) = \begin{cases} T_{\text{comp\_ring}}(n, p) & : \text{ falls } r = 0 \\ \max \left( \begin{array}{l} 5 \cdot T_{\text{mat\_add}} \left( \frac{n}{2}, \lceil \frac{p}{7} \rceil \right) + T_{\text{Comp\_strassen\_ring}} \left( \frac{n}{2}, \lceil \frac{p}{7} \rceil, r - 1 \right), \\ 7 \cdot T_{\text{mat\_add}} \left( \frac{n}{2}, \lfloor \frac{2p}{7} \rfloor \right) + 2 \cdot T_{\text{Comp\_strassen\_ring}} \left( \frac{n}{2}, \lfloor \frac{2p}{7} \rfloor, r - 1 \right) \end{array} \right) & : \text{ falls } r > 0 \end{cases} \quad (2.21)$$

### Kombination: Strassen + PDGEMM und Strassen + tpMM

Die Berechnungskosten für *Strassen+tpMM* und die blockorientierte Variante von *Strassen+PDGEMM* können bei Schema-Q8 gemeinsam betrachtet werden. In beiden algorithmischen Kombinationen werden die Matrizen in Blöcken mit  $n^2/p$  Elementen auf die Prozessoren verteilt. Für die Kostenfunktionen ist es egal, ob sie als Zeilen- oder Spaltenpanels wie bei tpMM oder als rechteckige Blöcke wie bei PDGEMM angeordnet sind.

Die Kommunikationszeit kann direkt aus der Struktur der M-Task  $T_{C_{11}}$  abgelesen werden, welche in Tab. 2.3 dargestellt ist. In dieser Implementierung werden zu Beginn sechs Teilmatrizen zwischen den Tasks verteilt und am Ende zwei Zwischenmatrizen ( $Q_4, Q_5$ ) empfangen. Da in Schema-Q8 jeder Task die gleiche Anzahl an Prozessoren zugewiesen wird, verwenden alle Tasks intern dieselbe Datenverteilung. In diesem Fall können alle Prozessoren ihren Anteil der Teilmatrizen mit einer einzigen Sendeoperation zu den Empfängerprozessor übermitteln. Die Abb. 2.13 kann als Beispiel betrachtet werden, in dem  $P_0$  Elemente an  $P_{12}$  und gleichzeitig  $P_1$  Daten an  $P_{13}$  schickt. Damit können die zusätzlichen Startup-Zeiten aus *Strassen+Ring* (siehe Abschnitt 2.4.9) eliminiert werden. Jeder

Prozessor sendet oder empfängt  $n^2/p$  Elemente. Man beachte hierbei, dass diese Anzahl unabhängig von der Rekursionstiefe ist. Im ersten Rekursionsschritt schickt ein Prozessor der Task  $T_{C_{11}}$   $n^2/p$  Elemente an einen Prozessor einer anderen Task. In diesem Schritt haben die Ausgangsmatrizen die Größe  $n \times n$  und sind auf  $p$  Prozessoren verteilt. Im zweiten Rekursionsschritt haben die Teilmatrizen  $A'$  und  $B'$  nur noch die Größe  $n/2 \times n/2$  und werden auf nur  $p/4$  der Prozessoren verteilt. Damit bleibt die Anzahl der zu übertragenden Elemente gleich, denn  $\frac{n^2/4}{p/4} = n^2/p$ .

Für  $r$  auszuführende Rekursionen lassen sich die Kommunikationskosten der parallelen Matrixmultiplikation nach Strassen (oberste Berechnungsstufe) in Schema-Q8 wie folgt formulieren:

$$T_{Comm}(n, p, r) = r \cdot 8 \cdot \left( \lambda + \frac{n^2}{p} \cdot \frac{1}{\tau} \right). \quad (2.22)$$

Für die Berechnungskosten von *Strassen+PDGEMM* können zwei Fälle unterschieden werden. Im ersten Fall sind jeder Task nach Ende der rekursiven Anwendung des Strassen-Algorithmus mindestens zwei Prozessoren zugewiesen. In diesem Fall wird PDGEMM aufgerufen. In einem zweiten Fall ist es möglich, dass genau  $\log_4 p$  Rekursionen ausgeführt werden, da die Anzahl der Prozessoren zu Beginn  $p = 4^i$ ,  $i \geq 1$  betrug. In diesem Fall ist einer Task in der untersten Rekursionsstufe nur ein Prozessor zugeordnet und es kann direkt DGEMM aufgerufen werden. Im Weiteren wird der zweite Fall als *full recursion* bezeichnet. Bei der Ausführung von  $r$  Rekursionen wird DGEMM genau  $2^r$  mal aufgerufen, da jede Task  $T_{C_{ij}}$  zweimal Strassen rekursiv aufruft, um die Teilresultate  $Q_l$  zu berechnen (siehe Tab. 2.3). Die Tasks  $T_{C_{11}}$  und  $T_{C_{22}}$  führen jeweils sieben Matrixadditionen pro Rekursionsstufe aus und dominieren somit die Berechnungszeit. Die Gesamtkosten für den Fall *full recursion* (fr) mit Schema-Q8 sind:

$$T_{Strassen\_fr\_Schema-Q8}(n, p, r) = T_{Comm}(n, p, r) + 7 \cdot r \cdot T_{mat\_add}(n, p) + \underbrace{2^r}_{2 \text{ pro Rekursion}} \cdot T_{dgemm} \left( \frac{n}{2^r}, \frac{n}{2^r}, \frac{n}{2^r} \right). \quad (2.23)$$

Wenn am Cut-off-Level einer M-Task mehrere Prozessoren zugewiesen sind, wird ein anderer Basisbaustein mit paralleler Implementierung aufgerufen, um die Berechnung zu vervollständigen. Demzufolge setzt sich die Zeit für die jeweilige Kombination aus der Zeit für Strassens Algorithmus und dem gewählten Baustein zusammen:

$$T_{Strassen\_Schema-Q8}(n, p, r) = T_{Comm}(n, p, r) + 7 \cdot r \cdot T_{mat\_add}(n, p) + 2^r \cdot \left\{ T_{tpMM}(n', p'), T_{PDGEMM}(n', p') \right\} \quad (2.24)$$

mit  $n' = \frac{n}{2^r}$  und  $p' = \frac{p}{4^r}$ .

## Zusammenfassender Überblick

Die Tab. 2.5 fasst die algorithmischen Bausteine und die mehrstufigen Kombinationen zusammen. Diese Übersicht vergleicht die Kostenfunktionen der verschiedenen Algorithmen, aufgeschlüsselt nach Kommunikation und Berechnung. Außerdem wird das verwendete Datenlayout exemplarisch veranschaulicht. Die algorithmische Variante *Strassen+PDGEMM*, welche Schema-Q8 oder Schema-Q7 mit einer blockzyklischen Verteilung der Matrizen implementiert, wird in dieser Arbeit nicht betrachtet, da sie auf Umverteilungsroutinen von ScaLAPACK basiert. Für Details der verwendeten Algorithmen zur Umverteilung von blockzyklischen Matrizen in ScaLAPACK sei auf [80] verwiesen.

### 2.4.10 Experimentelle Auswertung

Die mehrstufigen Algorithmen wurden auf verschiedenen Plattformen getestet, welche in Tab. 2.6 aufgeführt sind. Für die Experimente wurde meist auf ScaLAPACK 1.7 zurückgegriffen, damit eine Adaption der logischen Blockgröße vorgenommen werden konnte. Auf jeder Plattform wurde die bestmögliche BLAS-Implementierung eingesetzt: Auf dem Linux-Cluster (Opteron-Cluster) wurde ATLAS benutzt, auf der SGI Altix 4700 kam die MKL-Bibliothek von Intel zum Einsatz, und auf der IBM Regatta p690 wurde die von IBM gestellte ESSL-Bibliothek verwendet.

### Parameterabstimmung von PDGEMM

Bevor die Leistung der mehrstufigen Algorithmen mit PDGEMM verglichen werden kann, muss sichergestellt werden, dass PDGEMM die bestmögliche Leistung auf einer Zielplattform erreicht. Die Leistung von PDGEMM hängt stark von der Leistung der darunter liegenden Routine DGEMM ab. Um eine gute Leistung von DGEMM zu erzielen, muss die logische Blockgröße  $lb$  angepasst werden, die von den ScaLAPACK-Routinen verwendet wird. Das Kapitel 2.5 umfasst deshalb eine weitergehende Analyse der in PDGEMM verwendeten Algorithmen und zeigt, warum und wie die Geschwindigkeit durch Anpassung der logischen Blockgröße verbessert werden kann.

An dieser Stelle soll deshalb nur auf die für jede parallele Maschine angepassten Werte des Parameters  $lb$  eingegangen werden. Für jede DGEMM-Implementierung (ESSL, MKL) wird ein Profil erstellt, welches die Leistung verschiedener lokaler Matrixmultiplikationen aufzeichnet. Jede dieser Matrixmultiplikationen verwendet einen unterschiedlichen Wert der Dimension  $k$  (Dimension, über die beide Matrizen *verkettet* sind). Diese Dimension entspricht der logischen Blockgröße. Die anderen beiden Dimensionen (Zeilen von  $A$  und Spalten von  $B$ ) bleiben bei der Erstellung des Profils unverändert.

Die Tab. 2.7 fasst die experimentell ermittelten Werte für die verschiedenen Plattformen zusammen. Die Zeile „Max MFLOPS“ enthält die maximal ermittelten MFLOPS pro Prozessor für einen Aufruf von DGEMM und gibt die dafür verantwortliche Dimension  $k$  an. Wird  $lb$  zu groß gewählt, sind die in PDGEMM übertragenen Matrixblöcke u. U. zu groß, um ein Überlappen von Kommunikations- und Berechnungszeit zu ermöglichen. Deshalb wird der kleinste Wert der Dimension  $k$  gewählt, der mindestens 98 % der maximalen

Tab. 2.5: Überblick der mehrstufigen Algorithmen und deren Bausteine.

	KOMMUNIKATION	BERECHNUNG	LAYOUT
mat_add	-	$\frac{n^2}{p} \cdot t_{op}$	
DGEMM	-	$2 \cdot m \cdot n \cdot k \cdot t_{op}$	
tpMM	$2(p-1) \cdot (\lambda + n \cdot \lceil \frac{n}{p} \rceil \cdot \frac{1}{\tau})$	$p \cdot T_{dgemm}(\lceil \frac{n}{p} \rceil, \lceil \frac{n}{p} \rceil, n)$	
Ring	$2(p-1) \cdot (\lambda + n \cdot \lceil \frac{n}{p} \rceil \cdot \frac{1}{\tau})$	$p \cdot T_{dgemm}(\lceil \frac{n}{p} \rceil, \lceil \frac{n}{p} \rceil, n)$	
PDGEMM	$(\log_2 p_r \cdot \frac{n^2}{p_c} + \log_2 p_c \cdot \frac{n^2}{p_r}) \cdot \frac{1}{\tau} + \lceil \frac{n}{nb} \rceil \cdot (\log_2 p_r + \log_2 p_c) \cdot \lambda$	$\frac{2n^3}{p} \cdot t_{op}$	
Strassen+Ring Schema-Q7	$\begin{cases} T_{Comm\_ring}(n, p) & \text{falls } r = 0 \\ 9 \left( \frac{n}{2} \cdot \frac{n}{2} \cdot \frac{1}{\lceil \frac{p}{\tau} \rceil} \cdot \frac{1}{\tau} + 3\lambda \right) \\ + \max \left( T_{Comm\_strassen\_ring} \left( \frac{n}{2}, \lceil \frac{2p}{\tau} \rceil, r-1 \right), \right. \\ \left. T_{Comm\_strassen\_ring} \left( \frac{n}{2}, \lceil \frac{p}{\tau} \rceil, r-1 \right) \right) \\ \text{falls } r > 0 \end{cases}$	$\begin{cases} T_{comp\_ring}(n, p) & \text{falls } r = 0 \\ \max \left( \begin{aligned} &5 \cdot T_{mat\_add} \left( \frac{n}{2}, \lceil \frac{p}{\tau} \rceil \right) \\ &+ T_{Comp\_strassen\_ring} \left( \frac{n}{2}, \lceil \frac{p}{\tau} \rceil, r-1 \right), \\ &7 \cdot T_{mat\_add} \left( \frac{n}{2}, \lceil \frac{2p}{\tau} \rceil \right) \\ &+ 2 \cdot T_{Comp\_strassen\_ring} \left( \frac{n}{2}, \lceil \frac{2p}{\tau} \rceil, r-1 \right) \end{aligned} \right) \\ \text{falls } r > 0 \end{cases}$	
Strassen+tpMM Schema-Q8	$r \cdot 8 \cdot (\lambda + \frac{n^2}{p} \cdot \frac{1}{\tau}) + 2^r \cdot T_{tpMM}^{Komm.}(\frac{n}{2^r}, \frac{p}{4^r})$	$7 \cdot r \cdot T_{mat\_add}(n, p) + 2^r \cdot T_{tpMM}^{Ber.}(\frac{n}{2^r}, \frac{p}{4^r})$	
Strassen+ PDGEMM Schema-Q8	$r \cdot 8 \cdot (\lambda + \frac{n^2}{p} \cdot \frac{1}{\tau}) + 2^r \cdot T_{PDGEMM}^{Komm.}(\frac{n}{2^r}, \frac{p}{4^r})$	$7 \cdot r \cdot T_{mat\_add}(n, p) + 2^r \cdot T_{PDGEMM}^{Ber.}(\frac{n}{2^r}, \frac{p}{4^r})$	

**Tab. 2.6:** Überblick über die Testsysteme.

	IBM Regatta p690	Opteron-Cluster	SGI Altix 4700
Anbieter	NIC Jülich	Universität Bayreuth	LRZ München
Prozessor	1312 Power4+, 1.7 GHz	64 Opteron 246, 2 GHz	4096 Itanium2 1.6 GHz
Cachegrößen (L1/L2/L3)	32 kB, / 1.5 MB / 32 MB	64 kB / 1 MB / n/a	16 kB / 256 kB / 6 MB
Prozessoren pro Knoten	32	2	256
Cluster-Netzwerk	High Performance Switch	Infiniband	NUMalink 4
MPI	IBM-MPI	MPICH 1.2.5 + VMI 2.0	SGI MPT 1.14
Compiler	IBM XL C/C++ V8.0	GCC 3.2.2	GCC 4.1.0
Compiler Flags	-O3 -q64	-O3	-O3

**Tab. 2.7:** Ermittelte logische Blockgrößen für PDGEMM.

	IBM Regatta p690	SGI Altix 4700	Opteron-Cluster
DGEMM-Implementierung	ESSL	MKL	ATLAS
Max MFLOPS (Dimension $k$ )	4736 (680)	6125 (752)	3509 (732)
98 % MFLOPS	4641	6002	3437
Resultierende logische Blockgröße	196	208	668

MFLOPS-Rate erreicht. Dieser prozentuale Wert wurde empirisch festgelegt, da er sich in der Praxis als gut erwiesen hat. Der so ermittelte Wert  $k_{opt} \leq k$  definiert den in den Experimenten verwendeten Wert  $lb$ , welcher als „Resultierende logische Blockgröße“ in der letzten Zeile der Tab. 2.7 für die jeweilige Plattform angegeben ist.

## Übersicht der verglichenen Implementierungen

Die Diagramme dieser Auswertung verwenden die Einheit „MFLOPS / processor“, um die Leistungsfähigkeit der Algorithmen zu vergleichen. Die MFLOPS pro Prozessor bezeichnen dabei die durchschnittliche Anzahl an Fließkommaoperationen pro Prozessor ( $\times 10^6$ ), die in einer Sekunde ausgeführt werden. Zur Normalisierung wird angenommen, dass für die gesamte Matrixmultiplikation genau  $2n^3$  Operationen ausgeführt werden. Dies entspricht der Anzahl der Operationen einer Standard-Matrixmultiplikation. Der so ermittelte Wert der MFLOPS pro Prozessor ist ein guter Indikator für die parallele Leistungsfähigkeit und Skalierbarkeit der mehrstufigen Algorithmen für eine parallele Plattform.

Die folgenden Bezeichner markieren die verschiedenen Algorithmen in den Abbildungen:

- *Strassen (Schema-Q7,  $r$  recs) + Ring* bezeichnet die Implementierung, die den Algorithmus von Strassen mit Schema-Q7 in der oberen Ebene anwendet. Nach  $r$  Rekursionen wird die Ring-Methode angewendet.
- *Strassen (Schema-Q7,  $r$  recs, ScaLAPACK) + PDGEMM* bezeichnet die Strassen-Variante mit  $r$  Rekursionen, die in allen Hierarchieebenen eine blockzyklische Datenverteilung der Matrizen voraussetzt. Außerdem setzt diese Implementierung ausschließlich Umverteilungsfunktionen aus ScaLAPACK ein, d. h. es gibt im Anwendercode keine direkten Aufrufe von MPI.

- *PDGEMM (default)* bezeichnet die Standard-PBLAS-Funktion. Wird noch der Bezeichner „(default)“ angegeben, ist die Variante gemeint, welche die ursprüngliche logische Blockgröße (ohne Tuning) einsetzt.
- *Strassen (Schema-Q8,  $r$  recs) + PDGEMM* bezeichnet die Kombination aus Strassens Algorithmus ( $r$  Rekursionen) mit Schema-Q8 und PDGEMM auf der mittleren Ebene.
- *Strassen (Schema-Q8,  $r$  recs) + tpMM* bezeichnet die Kombination aus dem Algorithmus von Strassen und tpMM, welche das Schema-Q8 implementiert.
- *Mixed Strassen* bezeichnet die gemischt-parallele Implementierung des Algorithmus von Strassen aus [35]. Am Code dieses Verfahrens wurden keine Änderungen vorgenommen, so dass verschiedene Beschränkungen existieren. Das Verfahren setzt blockzyklisch verteilte quadratische Matrizen  $A$  und  $B$  voraus, die jeweils einem eigenen Prozesskontext zugeordnet sind. Die Prozessorgruppen beider Kontexte müssen disjunkt sein. Es wird *eine* Rekursionsstufe von Strassens Algorithmus ausgeführt und dann PDGEMM zur Lösung der Teilprobleme verwendet.

### Überprüfung der vorhergesagten Laufzeiten für den Opteron-Linux-Cluster

An dieser Stelle wird eine detaillierte Analyse der erwarteten und gemessenen Leistung der Kombination aus Strassens Algorithmus und Ring-Methode bei Anwendung von Schema-Q7 präsentiert. Das Schema-Q7 bietet sich deshalb an, da es keine redundanten Berechnungen ausführt und deshalb von der Reduzierung der auszuführenden Operationen am meisten profitiert. Der detaillierte Vergleich der erwarteten und gemessenen Leistung wurde für 56 Prozessoren auf dem **Opteron-Cluster** durchgeführt. Es wurden 56 Prozessoren gewählt, da diese Zahl durch 7 und durch 2 teilbar ist. Dadurch können die Prozessoren in einem rechteckigen Prozessorgitter angeordnet werden und man erhält eine recht gute Lastbalancierung bei Schema-Q8.

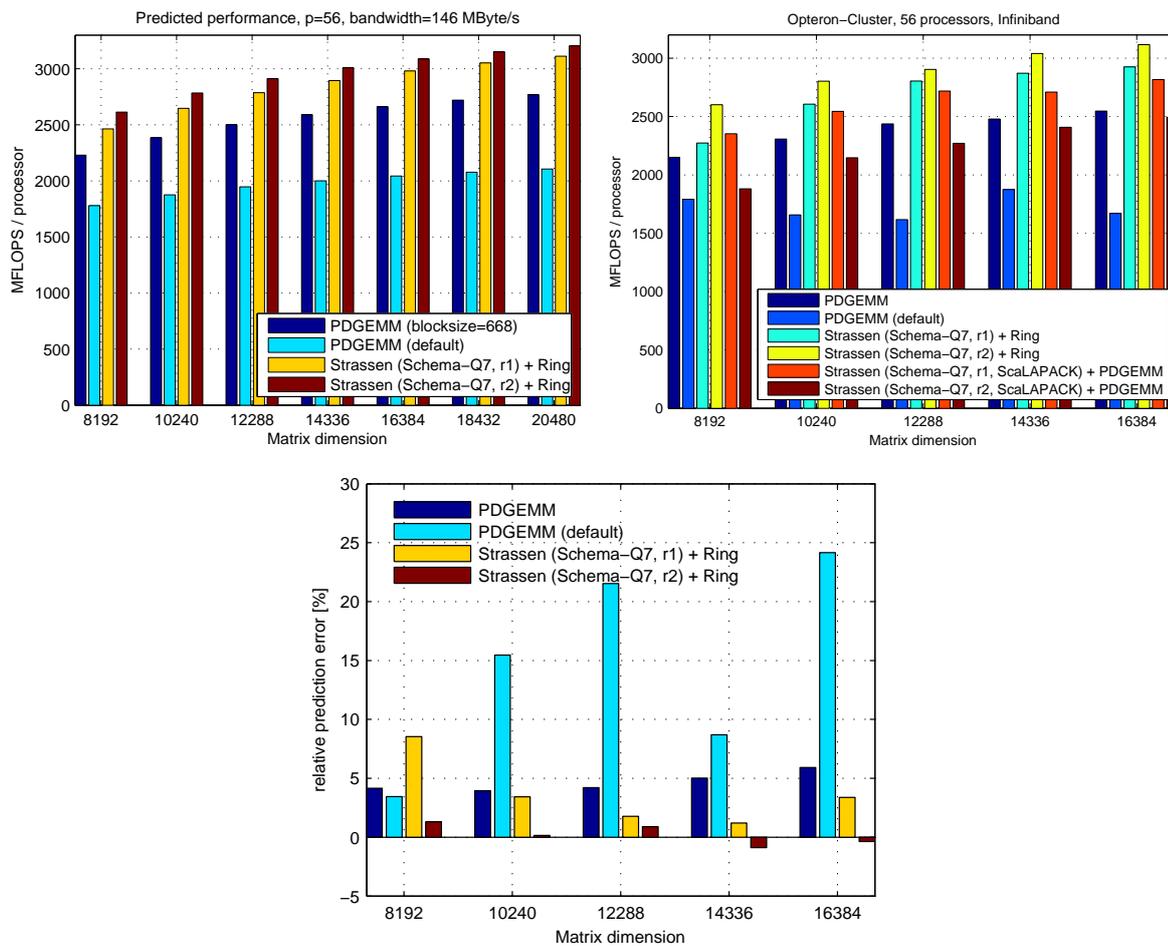
Für die Vorhersage der Kommunikations- und Berechnungszeit der mehrstufigen Algorithmen werden die Kostenfunktionen aus Abschnitt 2.4.9 benutzt. Die aus der Vorhersage gewonnene Information kann z. B. verwendet werden, um ein günstiges Verfahren für eine gewählte Zielplattform zu bestimmen. Damit ist es möglich eine Sammlung von Algorithmen bereitzustellen, um zur Laufzeit ein für ein Parameterset optimales Verfahren adaptiv zu selektieren. Für die Berechnung der Kosten mit den Formeln aus Abschnitt 2.4.9 werden spezifische Parametern der Zielplattform wie Latenzzeit, Bandbreite und Prozessortakt benötigt. Für den **Opteron-Cluster** wurden folgende Werte experimentell bestimmt: Die Prozessorgeschwindigkeit wurde auf 3300 MFLOPS festgesetzt. Die Optimierung der logischen Blockgröße ergab einen maximalen Wert von 3437 MFLOPS, siehe Tab. 2.7. Dieser Wert entspricht der Geschwindigkeit der Fließkommaoperationen, die nach Optimierung der logischen Blockgröße auf dem **Opteron-Cluster** mit DGEMM erzielt werden konnte. Allerdings zeigten weitere Tests, dass ein Wert von 3300 MFLOPS im Mittel besser den realen Bedingungen entspricht, da die logische Blockgröße für eine feste Matrixgröße und für eine bestimmte Anzahl von Prozessoren optimiert wurde. Damit ergibt

sich  $t_{op} = \frac{1}{3300 \cdot 10^6} s \approx 0.3 ns$ . Die Prozessorgeschwindigkeit für die Standardvariante von PDGEMM wurde auf 2500 MFLOPS festgesetzt, da dies der Leistung der DGEMM-Routine entsprach, die vor Optimierung der logischen Blockgröße (bei  $lb = 32$ ) mit PDGEMM erzielt werden konnte.

Die effektiv nutzbare Netzwerkbandbreite pro Prozessor ( $\lambda$ ) wurde mit dem COMMS3-Benchmarks der ParkBench-Sammlung [79] mit allen verfügbaren 64 Prozessoren ermittelt. COMMS3 benutzt einen All-To-All-Test, um die verfügbare Bandbreite festzustellen, die mit einer Message-Passing-Bibliothek auf einem verteilten System bei hoher Netzlast möglich ist. Da die Kommunikationsschritte der gemischt-parallelen Varianten von Strassens Matrixmultiplikation viel Netzlast erzeugen, kann mit den Resultaten von COMMS3 eine einigermaßen realistische Bandbreite zwischen zwei Prozessoren angegeben werden. Die Bandbreite des Infiniband-Netzwerks für eine Point-To-Point-Kommunikation von ca. 600 MB/s fiel bei dem All-To-All-Test auf einen Wert von 146 MB/s. Das bedeutet, dass deutlich weniger Bandbreite zur Verfügung steht, wenn alle Prozessoren gleichzeitig Daten über den Switch senden, als wenn nur zwei Prozessoren Daten austauschen. Ein Ping-Pong-Benchmark diente zur Bestimmung der Latenzzeit, welche für den Opteron-Cluster mit  $\lambda = 17 ms$  angegeben werden kann.

Die Abb. 2.14 vergleicht die vorhergesagten Leistungen (linken Seite) in MFLOPS pro Prozessor mit den gemessenen Werten (rechte Seite) von PDGEMM und *Strassen+Ring* mit 56 Prozessoren. Die Grafik mit den vorhergesagten Leistungswerten enthält nicht *Strassen+PDGEMM* für Schema-Q7, da sich die komplexen Umverteilungsroutinen von ScaLAPACK schwierig modellieren lassen. Die Leistung der Algorithmen hängt in erster Linie von der Matrixgröße und der Anzahl der Prozessoren ab. Die mehrstufige Kombination aus *Strassen+Ring* bringt einen größeren Kommunikationsoverhead im Vergleich zu PDGEMM mit. Andererseits werden im Vergleich auch weniger Operationen ausgeführt, je mehr Rekursionen angewendet werden können und desto größer die Matrizen sind. Für viele Konfigurationen gibt es eine Matrixgröße, an dem sich die Leistungskurven von *Strassen+Ring* und PDGEMM kreuzen (Crossover-Punkt). Ab dieser Matrixgröße beginnt sich die Anwendung von *Strassen+Ring* auszuzahlen. Ziel ist natürlich, diesen Punkt so früh wie möglich zu erreichen (im Sinne der Matrixgröße).

Im Modell ist jedoch *Strassen+Ring* für alle betrachteten Matrixgrößen schneller als PDGEMM. Dies deckt sich mit den experimentellen Resultaten für 56 Prozessoren. Das bedeutet, dass man den Crossover-Punkt schon erreicht hat. In der unteren Grafik ist der relative Fehler der berechneten Laufzeit in Prozent dargestellt. Der Fehler für die beiden Varianten von *Strassen+Ring* ist erstaunlich gering. Ebenso wurde die Laufzeit der optimierten Version von PDGEMM sehr genau vorausgesagt. Dies ist aber auch ein Indikator für die geringere Leistung von PDGEMM, denn das Modell betrachtet keine Überlappung von Kommunikations- und Rechenzeit. Diese Überlappung ist aber das erklärte Ziel der PDGEMM-Implementierung, die demzufolge auf dem vorliegenden Testsystem nicht optimal ausgenutzt werden kann. Die Schwankungen der Standardvariante von PDGEMM ist durch Laufzeitunterschiede der Basisroutine DGEMM zu erklären. Mit dem Standardwert der logischen Blockgröße  $lb = 32$  treten je nach Eingabegröße der Matrizen schnell Schwankungen bei der erzielten Leistung von DGEMM auf. Aus diesem Grund ist auch die Leistung der parallelen Routine schwerer vorherzusagen.

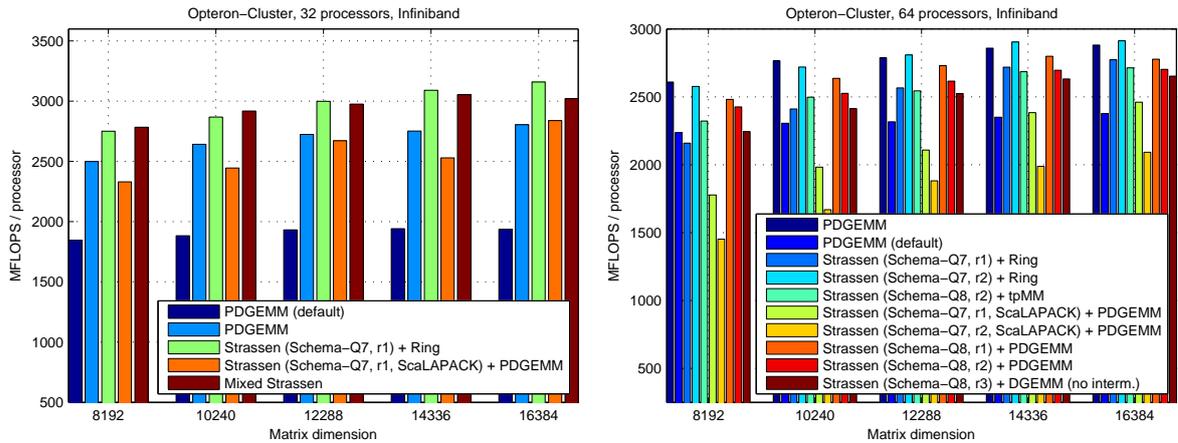


**Abb. 2.14:** Erwartete (links oben) und gemessene Leistung von PDGEMM und *Strassen+Ring* in MFLOPS pro Prozessor für  $p = 56$  auf Opteron-Cluster. Darunter: Relativer Fehler der Laufzeitabschätzung in Prozent.

Um noch bessere Vorhersagen zu treffen, könnte man ein noch besseres Benchmarking verwenden. Anstatt nur die maximale Leistung der Prozessoren und der Netzwerkkarten für eine bestimmte Konfiguration zu betrachten, ist es möglich, mehrere Testfälle zu analysieren und entsprechend zu intra- oder extrapolieren. Trotzdem ist das Modell genau genug, um eine adaptive Auswahl eines Algorithmus für eine vorliegende Konfiguration vorzunehmen.

### Experimentelle Ergebnisse für den Opteron-Linux-Cluster

Die auf Opteron-Cluster experimentell mit 32 und 64 Prozessoren gemessenen Laufzeiten (in MFLOPS pro Prozessor) für die unterschiedlichen Matrixmultiplikationsalgorithmen werden in Abb. 2.15 veranschaulicht. Für 32 Prozessoren wird die Laufzeit von PDGEMM von den Algorithmen *Strassen+Ring* und *Mixed Strassen* unterboten. Die Kombination aus *Strassen+Ring* zeigt dabei einen kleinen Vorteil für größere Matrixgrößen. Im Fall von

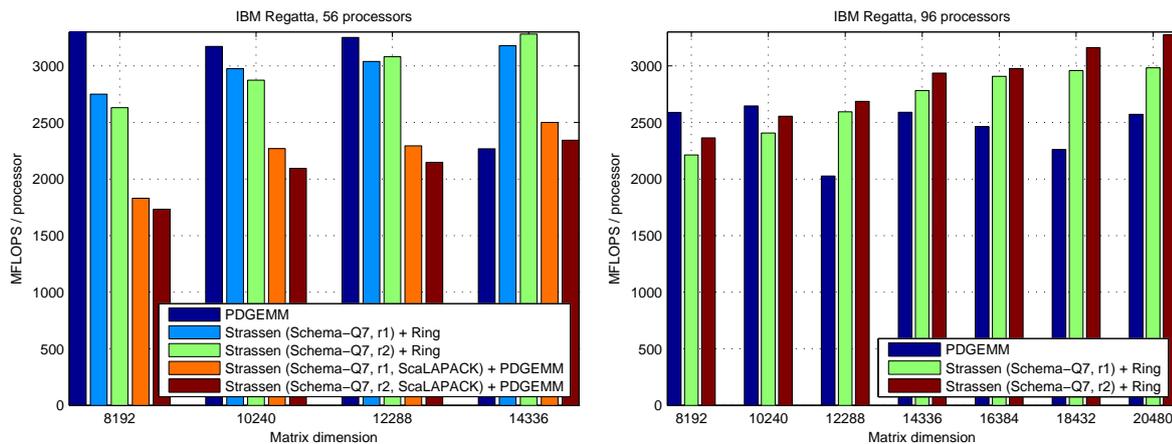


**Abb. 2.15:** Leistungsvergleich für Opteron-Cluster,  $p = 32$  (links) und  $p = 64$ .

64 Prozessoren werden Algorithmen beider Schemata verglichen, da für diese Anzahl alle Varianten angewendet werden können. Mit diesem experimentellen Vergleich wird insbesondere evaluiert, ob die redundante Ausführung einer Task in Schema-Q8 von den geringen Kommunikationskosten im Gesamtergebnis profitieren kann. Es ist gut zu erkennen, dass im Schema-Q8 eine mehrfache rekursive Anwendung des Algorithmus von Strassen nicht zu einer verbesserten Leistung führt. Für 64 Prozessoren zeigt die optimierte Variante von PDGEMM eine sehr gute Leistung für alle Matrixdimensionen. Die anderen mehrstufigen Kombinationen wie *Strassen+PDGEMM* und *Strassen+tpMM* bringen nur einen Vorteil gegenüber der nicht optimierten Variante von PDGEMM. Eine ähnlich gute Performance wie die von PDGEMM wird nur von *Strassen+Ring* erreicht. Die Kombination *Strassen+Ring* erzielt sogar bessere Ergebnisse für größere Matrizen, obwohl PDGEMM in den Vorhersagen für diese Prozessoranzahl besser war. Für diese Ergebnisse gibt es verschiedene Gründe:

1. Die mehrstufige Kombination *Strassen+Ring* ist für diese parallele Plattform eine sehr passende Variante.
2. Die Routine DGEMM erzielt eine leicht bessere Leistung für größere Matrixblöcke. Bei *Strassen+Ring* sind die Matrizen, die beim Aufruf von DGEMM als Parameter übergeben werden oft größer als bei PDGEMM.
3. Die Kostenfunktion von *Strassen+Ring* ist nur eine obere Schranke für die realen Kosten, denn es wird oft gerundet und davon ausgegangen, dass ein Verbindungsaufbau bei der Matrixumverteilung zwischen zwei Prozessorengruppen mehrfach stattfinden muss.

Der Grund für die geringere Leistung von *Strassen (ScaLAPACK) + PDGEMM* ist der höhere Kommunikationsoverhead, welcher mit der Anwendung der Umverteilungsroutinen für blockzyklische Datenverteilungen von ScaLAPACK verbunden ist. Im Normalfall sind die blockzyklisch verteilten Matrizen über alle Prozessoren verteilt. Deshalb muss jeder



**Abb. 2.16:** Leistungsvergleich für die IBM Regatta p690,  $p = 56$  (links) und  $p = 96$ .

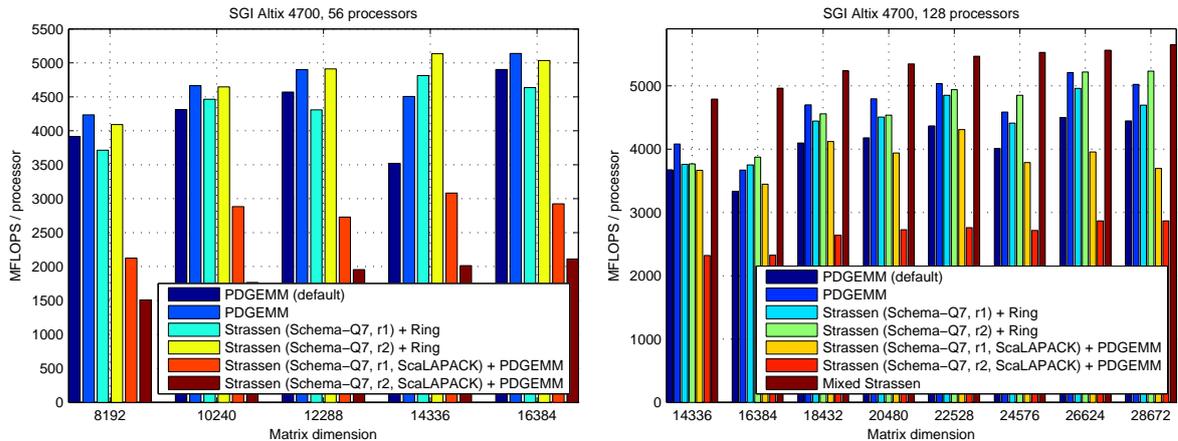
Prozessor eines Kontextes Nachrichten schicken, um einen Matrixtransfer zwischen Kontexten zu ermöglichen. Beispielsweise müssen alle Prozessoren Nachrichten verschicken, um die Teilmatrix  $A_{11}$  auf eine Teilmenge der Prozessoren zu übertragen.

Die Ergebnisse für diese Plattform werden wie folgt zusammengefasst: Die optimierte Variante von PDGEMM erzielt durchweg gute Ergebnisse. Für quadratische Prozessorgitter ist die Leistung von PDGEMM im Vergleich zu den anderen Algorithmen am besten. Eine ähnlich gute oder sogar bessere Performance als PDGEMM ist nur möglich wenn Schema-Q7 angewendet wird (*Strassen+Ring* oder *Strassen+PDGEMM*). Aus diesem Grund konzentrieren sich die weiteren experimentellen Betrachtungen auf den Vergleich zwischen PDGEMM und den Kombinationen, die Schema-Q7 anwenden können.

### Ergebnisse auf anderen Parallelrechnern

Die Abb. 2.16 zeigt die experimentellen Ergebnisse für 56 (links) und 96 Prozessoren (rechts), die auf der IBM Regatta p690 erzielt wurden. Die Grafiken enthalten keine ausgezeichneten Balken für die unoptimierte Variante von PDGEMM (default), da die besten Resultate von PDGEMM mit der PESSL-Bibliothek von IBM erreicht wurden. Die Ergebnisse für 56 Prozessoren zeigen, dass PDGEMM außer bei der Matrixdimension 14 336 der schnellste Algorithmus ist. Der Leistungseinbruch bei 14 336 liegt wahrscheinlich wieder an Cache-Effekten. Man kann außerdem feststellen, dass der Leistungsunterschied zwischen *Strassen+Ring* und PDGEMM mit zunehmender Matrixgröße abnimmt. Speziell in der Grafik für 96 Prozessoren wird die stetige Verbesserung der Leistung von *Strassen+Ring* gegenüber PDGEMM deutlich. Die Kombination von *Strassen+Ring* zieht den Vorteil aus einer geringeren Anzahl an auszuführenden arithmetischen Operation im Vergleich zu PDGEMM. Kommen mehr Prozessoren zum Einsatz, wirkt sich die leicht unbalancierte Arbeitslast von *Strassen+Ring*<sup>6</sup> nicht so stark aus, so dass sich die Gesamtleistung trotzdem verbessert.

<sup>6</sup>96 ist kein Vielfaches von 7.



**Abb. 2.17:** Leistungsvergleich für die SGI Altix 4700,  $p = 56$  (links) und  $p = 128$ .

Ähnliche Aussagen können nach Auswertung der Resultate für 56 und 128 Prozessoren für die SGI Altix 4700 getroffen werden, welche in Abb. 2.17 dargestellt sind. Die Implementierung von *Strassen+PDGEMM* auf Basis von blockzyklischen Matrizen ist aufgrund des Kommunikationsoverheads am langsamsten. Bei 56 Prozessoren zeigen PDGEMM und *Strassen+Ring* vergleichbare Resultate für alle getesteten Matrixgrößen. Die Abb. 2.17 (rechts), welche die Ergebnisse für 128 Prozessoren veranschaulicht, enthält außerdem noch die Laufzeiten der Implementierung von *Mixed Strassen*. Wie bereits auf der IBM Regatta p690 gesehen, hat PDGEMM einen Vorteil gegenüber *Strassen+Ring* für kleinere Matrizen. Ist die Matrixdimension groß genug, so dass der Kommunikationsoverhead von *Strassen+Ring* kleiner ist als die Zeit, die durch eine geringe Anzahl an Operationen zur Berechnung benötigt wird, erzielt *Strassen+Ring* bessere Resultate als PDGEMM. Dies lässt sich sehr gut in der Grafik für 128 Prozessoren erkennen, da PDGEMM bis zu einer Dimension von 22 528 Elementen schneller ist als *Strassen+Ring*. Bei dieser Dimension ist der Crossover-Punkt erreicht und *Strassen+Ring* erzielt danach geringere Laufzeiten für größere Matrizen. Es ist außerdem bemerkenswert, dass der Algorithmus *Mixed Strassen* im Fall von 128 Prozessoren immer am besten abschneidet. Es ist aber auch zu erkennen, dass sich der Leistungsunterschied zwischen *Strassen+Ring* und *Mixed Strassen* mit größerer Dimension verkleinert. Im Gegensatz zu *Strassen+Ring* arbeitet *Mixed Strassen* nur mit zwei disjunkten quadratischen Prozessorgittern, so dass es nicht möglich war, die Leistungsbewertung für andere Prozessoranzahlen als 32 ( $2 \times 4^2$ ) und 128 ( $2 \times 8^2$ ) vorzunehmen. Da 128 kein Vielfaches von 7 ist, entstehen daraus für *Strassen+Ring* Nachteile in der Lastbalancierung, die zu der geringeren Leistung führen.

Die Experimente haben gezeigt, dass die mehrstufigen Kombinationen aus Strassens Algorithmus und tpMM, Ring und PDGEMM konkurrenzfähig zur Routine PDGEMM sind. Die Kombination *Strassen+Ring* erzielt sogar bessere Ergebnisse als PDGEMM, selbst wenn die logische Blockgröße von PDGEMM vorher an das System angepasst wurde. Es wurde auch deutlich, dass PDGEMM sehr gut bei quadratischen Prozessorgittern funktioniert. Trotzdem konnte *Strassen+Ring* für  $p = 49 = 7^2$  mit PDGEMM mithalten und für

**Tab. 2.8:** Übersicht der schnellsten Algorithmen für jede Plattform.

#Prozessoren	IBM Regatta p690	SGI Altix 4700	Opteron-Cluster
56	<i>Strassen+Ring</i>	PDGEMM	<i>Strassen+Ring</i>
64	PDGEMM	PDGEMM	<i>Strassen+Ring</i>
96	<i>Strassen+Ring</i>	PDGEMM	– ( <i>max. 64 Proz.</i> )
98	<i>Strassen+Ring</i>	<i>Strassen+Ring</i>	– ( <i>max. 64 Proz.</i> )

größere Matrizen bessere Laufzeiten erzielen. Für rechteckige (aber nicht quadratischen) Prozessorgitter kann *Strassen+Ring* die Leistung von PDGEMM überbieten, wenn die Anzahl der Prozessoren ein Vielfaches von 7 ist (z. B. 56, 98). Die Tab. 2.8 enthält eine Zusammenfassung der besten Algorithmen für jede Zielplattform.

## 2.5 Leistungsverbesserung von PDGEMM durch automatische Parameterabstimmung

Dieser Abschnitt beschreibt die Funktionsweise der Routine PDGEMM aus ScaLAPACK und zeigt, wie ihre Ausführungsgeschwindigkeit durch eine geeignete Parameterwahl verbessert werden kann.

### 2.5.1 Motivation

Bei der Analyse und beim Vergleich der Laufzeiten der mehrstufigen Verfahren zur Matrixmultiplikation (Strassen, tpMM) wurden bei PDGEMM Unregelmäßigkeiten festgestellt. Auf verschiedenen Plattformen konnte PDGEMM einfach nicht mit anderen Verfahren mithalten, obwohl der Algorithmus und die Implementierung sehr viel mehr Potenzial haben. Je nach PDGEMM-Implementierung gibt es unterschiedliche Parameter, die die Gesamtperformance beeinflussen können. Dieser Abschnitt gliedert sich in zwei Teile. Zuerst wird die PDGEMM-Implementierung analysiert, um herauszufinden, welche Parameter auf die Leistung einen essenziellen Einfluss haben. Aus den so gewonnen Erkenntnissen lässt sich dann ein Verfahren angeben, mit dessen Hilfe diese Parameter für eine gegebene Hardware-Plattform optimiert werden können.

### 2.5.2 PDGEMM – Überblick über verwendete Algorithmen

Die Routine PDGEMM ist eine Funktion aus den PBLAS (Parallel Basic Linear Algebra Subprograms), die wiederum Teil von ScaLAPACK sind [14]. Der Name PDGEMM leitet sich von der Einprozessorroutine DGEMM (*GE*neral *M*atrix *M*ultiply) ab, wobei *P* für „parallel“ und *D* für „double“ stehen. Mittlerweile gibt es eine Vielzahl von Implementierungen der PBLAS-Routinen von verschiedenen Hardware-Herstellern, z. B. PESSL von IBM oder MKL von Intel. Diese Bibliotheken sind von den Herstellern speziell auf ihre Hardware zugeschnitten. Die Implementierung von ScaLAPACK hingegen ist universell

**Algorithmus 2.2** SUMMA

---

```

while  $C$  noch nicht komplett berechnet do
  Broadcast einer Spalte von  $A$  entlang einer Zeile von Prozessoren
  Broadcast einer Zeile von  $B$  entlang einer Spalte von Prozessoren
  Lokale Matrixmultiplikation (xGEMM)

```

---

verwendbar und quelloffen<sup>7</sup>. Die in ScaLAPACK verwendeten Algorithmen wurden in verschiedenen Veröffentlichungen beschrieben, was eine Untersuchung erst möglich macht.

Die Basis der Implementierung von PDGEMM bildet der Algorithmus SUMMA (Scalable Universal Matrix Multiplication Algorithm) [108]. Bei SUMMA werden die Matrizen  $A$  und  $B$  auf ein rechteckiges Prozessorgitter blockzyklisch mit einer Blockgröße  $nb$  verteilt. Bei dieser Verteilung wird der erste Matrixblock der Größe  $nb \times nb$  auf Prozessor 0 abgebildet, der nächste Block auf Prozessor 1, usw. Die Verteilung der Daten entspricht der Darstellung in Abb. 2.18<sup>8</sup>. Die generelle Arbeitsweise von SUMMA wird in Alg. 2.2 veranschaulicht. Um die Elemente der Ergebnismatrix  $C$  zu berechnen, broadcastet zuerst die erste Prozessorspalte eine Spalte der Matrix  $A$  und die erste Prozessorzeile eine Zeile der Matrix  $B$ . Nachdem jeder Prozessor einen Teil von  $A$  und  $B$  erhalten hat, kann eine lokale Matrixmultiplikation, z. B. mit DGEMM, durchgeführt werden. Danach wird die nächste Spalte von  $A$  und die nächste Zeile von  $B$  als Pivotzeile benutzt und mit Broadcasts verteilt. Als nächste zu sendende Spalte von  $A$  oder als nächste Zeile von  $B$  wird die Nummerierung der Ursprungsmatrizen beibehalten, d. h. im Beispiel wird für  $A$  nach Zeile 0 die Zeile 1 gebroadcastet. Demnach senden in jedem Kommunikationsschritt unterschiedliche Prozessorspalten. Diese Prozedur wird wiederholt bis alle Spalten (Zeilen) von  $A$  ( $B$ ) verwendet wurden.

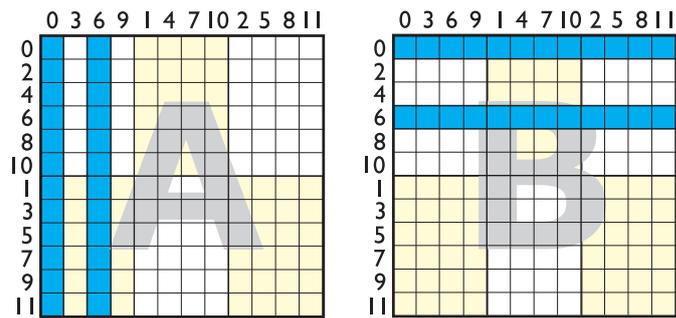
Der von PDGEMM in ScaLAPACK eingesetzte Algorithmus ist DIMMA [26], welcher die Konzepte von SUMMA ein wenig verfeinert. DIMMA benutzt ein Kommunikationsschema nach LCM-Prinzip (*least common multiple*, kleinstes gemeinsames Vielfaches), um ein besseres Pipelining – die Überlappung von Kommunikation und Berechnung – zu erzielen. Die Abb. 2.18 zeigt exemplarisch die blockzyklische Verteilung der Matrix  $A$  auf sechs Prozessoren ( $p_r \times p_c = 2 \times 3$ ). Die Spalten von  $A$ , die zuerst via Broadcast ausgetauscht werden, sind schattiert eingezeichnet (zuerst Spalte 0, dann Spalte 6, weil  $\text{LCM}(2, 3) = 6$ ). Bei DIMMA broadcastet jede Prozessorspalte des Prozessorgitters alle Spalten von  $A$  nacheinander, d. h. nach Spalte 6 wird Spalte 3 von  $A$  verschickt. Es senden immer die Prozessorzeilen, die die korrespondierenden Zeilen von  $B$  besitzen.

Die Gesamtleistung von PDGEMM hängt direkt von der Leistung der Routine DGEMM ab, die als Berechnungskern für lokale Matrixmultiplikationen verwendet wird. DGEMM ist Teil der BLAS [37] und wird wie PDGEMM von verschiedenen Herstellern angeboten. Es existieren freie DGEMM-Implementierungen, z. B. in ATLAS [112] und GotoBlas [48]. Beide Bibliotheken versuchen die Ressourcen eines Prozessors möglichst optimal auszunutzen (z. B. SSE2). Dazu werden Speichertransferzeiten verdeckt und die Pipelines des Prozessors möglichst gut ausgenutzt. Trotz aller Optimierungen ist die Leistung von DGEMM

---

<sup>7</sup>BSD-style license

<sup>8</sup>SUMMA und DIMMA verwenden dieselbe Aufteilung.



**Abb. 2.18:** DIMMA-Snapshot für ein  $2 \times 3$ -Prozessorgitter. Die Nummern an den Zeilen und über den Spalten bezeichnen die Zeilen- oder Spaltennummern der Ursprungsmatrix vor der blockzyklischen Verteilung.

(in FLOPS) durch die Anzahl der Elemente der Matrizen limitiert. Für kleine Matrizen lassen sich die Cache-Hierarchie und die Berechnungseinheiten nicht optimal ausnutzen, was zu einer geringeren Performance der Routine führt. Deshalb ist es für die Geschwindigkeit der parallelen Routine von großer Bedeutung, dass die lokalen Matrixupdates fast die Peakperformance des Prozessors erreichen.

### 2.5.3 Leistungssensitive Parameter von PDGEMM

Im Folgenden wird ein Überblick über die einzelnen System- und Funktionsparameter von PDGEMM gegeben, um deren Einfluss auf die Leistung abschätzen zu können. Die Abb. 2.19 zeigt die Fortran-Schnittstelle von PDGEMM, wie sie in ScaLAPACK definiert ist. Die Parameter TRANSA und TRANSB geben an, ob die Matrizen in transponierter

```
CALL PDGEMM( TRANSA, TRANSB, M, N, K, ALPHA,
             A, IA, JA, DESC_A,
             B, IB, JB, DESC_B, BETA,
             C, IC, JC, DESC_C )
```

**Abb. 2.19:** Fortran-Schnittstelle von PDGEMM.

Form vorliegen. Die Werte ALPHA und BETA sind konstante Faktoren für A und B. Die IX- und JX-Werte definieren den linken oberen Rand des Prozessorgitters, falls die Matrizen nicht auf allen Prozessoren gespeichert sind.

Die für die Leistung entscheidenden Parameter sind die Dimensionen M, N und K sowie die Matrixdeskriptoren DESC\_A, DESC\_B und DESC\_C. In den Deskriptoren wird gespeichert, wie viele Zeilen und Spalten der Matrizen A, B und C jeder Prozessor speichert. Die Anzahl der pro Prozessor zu speichernden Elemente hängt neben den Matrixdimensionen auch von der Anzahl der Prozessoren, dem Prozessorgitter und der gewählten Blockgröße für die blockzyklische Datenverteilung der Matrizen ab. Mit dieser Information kann jeder Prozessor anhand seiner internen Nummerierung die Position im Gitter und die damit resultierende Anzahl an Elementen berechnen.

Implementiert man einen parallelen Algorithmus, wie z. B. die SUMMA-Matrixmultiplikation, stellt sich die Frage nach der Größe der zwischen den Prozessoren zu transferierenden Matrixblöcke. Bindet man diese Transferblockgröße an den Blockfaktor  $nb$  der blockzyklischen Datenverteilung, gerät man schnell in Leistungsfallen. Ist diese Blockgröße zu klein gewählt, entsteht ein drastischer Anstieg des Kommunikationsoverheads im Programm. Hingegen führt eine zu große Blockgröße zu weniger Potenzial, um Kommunikations- und Berechnungszeiten zu überlappen. Aus den genannten Gründen verwenden die PBLAS-Routinen aus ScaLAPACK eine logische Blockgröße ( $lb$ ), die von der eigentlichen Blockgröße der Datenverteilung abstrahiert. Die implementierten Algorithmen benutzen also die logische Blockgröße als kleinste Einheit.

Die Hauptgründe für das unerwartete Einbrechen der Laufzeit von PDGEMM im oben angesprochenen Laufzeittest kann damit auf eine Untersuchung der Blockgröße der Datenverteilung  $nb$  und der logischen Blockgröße  $lb$  des Algorithmus reduziert werden. Da sich die Analyse hauptsächlich mit der Veränderung von Blockgrößen beschäftigt, werden beide nochmals kurz zusammengefasst.

**Blockfaktor** Der Blockfaktor  $nb$  wird bei einer blockzyklischen Datenverteilung benutzt, um die Zeilen und Spalten der Matrizen auf das Prozessorgitter abzubilden. Ein Blockfaktor  $nb$  heißt, dass die Matrix  $M \in R^{n \times n}$  in  $n/nb \times n/nb$  Blöcken der Größe  $nb \times nb$  zyklisch auf das Gitter verteilt wird. Der Blockfaktor  $nb$  ist also ein Parameter der Datenverteilung. Es ist auch möglich, zwei unterschiedliche Blockgrößen zu verwenden, eine für Zeilen und eine für Spalten. Damit lassen sich Blöcke optimal an die Cache-Größe einzelner Prozessoren anpassen.

**Logische Blockgröße** Im Fall von PDGEMM bezeichnet die logische Blockgröße  $lb$  die Größe der Teilmatrix von Matrix  $C$ , die pro Prozessor in jedem Schritt parallel berechnet wird. In jeder Iteration werden  $lb$  Zeilen von  $A$  und  $lb$  Spalten von  $B$  zwischen den Prozessoren transferiert. Damit kann jeder Prozessor einen Teil der Matrix  $C$  mit  $lb \times lb$  Elementen berechnen. Die logische Blockgröße  $lb$  ist damit ein Parameter der algorithmischen Implementierung von ScaLAPACK.

Der Blockfaktor  $nb$  und die logische Blockgröße  $lb$  stehen nicht direkt miteinander in Verbindung. Der Blockfaktor  $nb$  entscheidet lediglich, wie viele und welche Elemente einer Matrix ein Prozessor speichern soll. Die logische Blockgröße  $lb$  bestimmt die Puffergröße innerhalb eines verteilt arbeitenden Algorithmus, d. h. es werden in den Kommunikationsschritten  $lb$  Spalten oder Zeilen versendet. Wie viele Blöcke der Größe  $nb$  dafür gebraucht werden ist variabel, d. h.  $nb$  kann größer als auch kleiner als  $lb$  sein. In der Praxis wird man aber gewöhnlich  $nb < lb$  antreffen, da eine kleine Blockgröße eine gut balancierte Berechnung ermöglicht und eine größere logische Blockgröße den Kommunikationsoverhead senkt.

**Tab. 2.9:** Konfiguration des Testsystems.

System	Beowulf-Cluster (32 Knoten mit je 2 AMD Opteron) Linux 2.4.21
Prozessor	Opteron 244, 2 GHz, 128 KB L1-Cache, 1024 KB L2-Cache
C/F77 Compiler	GCC 3.4.0
MPI Version	MPICH 1.2.5 + VMI 2.0 (Infiniband)
Infiniband driver	Mellanox HPC Gold Collection (IBHPC) v0.5.0 for Linux Mellanox THCA for Linux 3.2-rc17
ScaLAPACK	1.7
ATLAS	3.6.0

## 2.5.4 Parameteranalyse

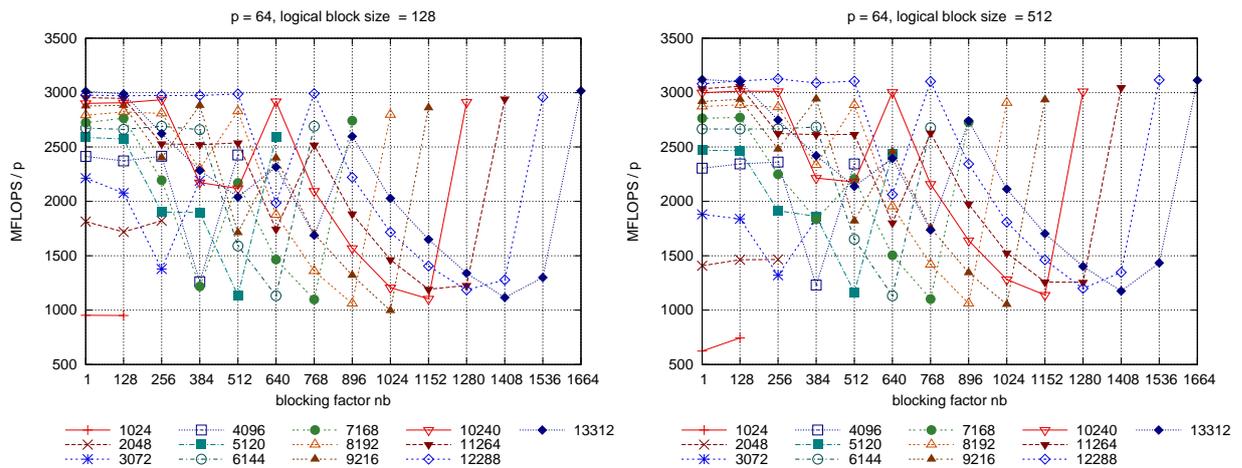
In diesem Abschnitt wird der Einfluss der beiden Parameter, Blockfaktor und logische Blockgröße, auf die Leistung von PDGEMM experimentell untersucht. Die Tab. 2.9 gibt einen Überblick über die verwendete Testplattform.

### Einfluss des Blockfaktors auf die Leistung von PDGEMM

Um die Abhängigkeit der Leistung von PDGEMM vom Blockfaktor zu analysieren, wurde eine Serie von Laufzeitmessungen mit unterschiedlichen Blockfaktoren durchgeführt. Damit die Zahl der Experimente, bedingt durch die große Anzahl an Freiheitsgraden, nicht explodiert, beschränkt sich die Studie auf quadratische Matrizen und einen gemeinsamen Blockfaktor für die Matrixdimensionen  $M$ ,  $N$  und  $K$ , also  $mb = nb = kb$ . In den Tests wurde der Blockfaktor für unterschiedliche Matrixdimensionen zwischen 1 und einem Maximum variiert. Dieses Maximum ergibt sich aus der Matrixdimension  $n$  und dem Prozessorgitter  $p_r \times p_c$ . In dieser Studie wurde die maximale Blockgröße als

$$lb_{max} = \frac{n}{p_r} \quad (2.25)$$

definiert. Die Abb. 2.20 zeigt die von PDGEMM erreichten MFLOPS pro Prozessor für unterschiedliche Matrixdimensionen in Abhängigkeit des Blockfaktors  $nb$  (x-Achse). In beiden Experimenten wurde ein quadratisches Prozessorgitter ( $p = 64 = 8 \times 8$ ) eingesetzt. Deshalb ist z. B. die Blockgröße für die Matrixdimension  $n = 1024$  auf  $1024/8 = 128$  begrenzt. Im linken Diagramm der Abb. 2.20 ist bei einer logischen Blockgröße von  $lb = 128$  auf Anhieb kein Muster zu erkennen. Es kann aber festgestellt werden, dass die Leistung von PDGEMM für kleine Blockgrößen  $nb$  (1, 128) immer sehr gute Ergebnisse erzielt. Für größere Werte von  $nb$  tauchen solch stetig gute Werte nicht auf. Dieselben Tendenzen zeigt auch die rechte Grafik für  $lb = 512$ . Das legt zumindest die Annahme nahe, dass die Leistungsunterschiede für verschiedene Blockgrößen  $nb$  nicht von  $lb$  abhängen. Untersucht man die Peakwerte für eine Matrixdimension genauer, kristallisieren sich Muster heraus. Als Beispiel wird die Matrixmultiplikation mit  $n = 13\,312$  betrachtet, welche für die Blockfaktoren  $nb = 1, 128$  und  $1664$  zu Peaks führt. Diese Werte sind Teiler von  $13\,312$ , womit eine



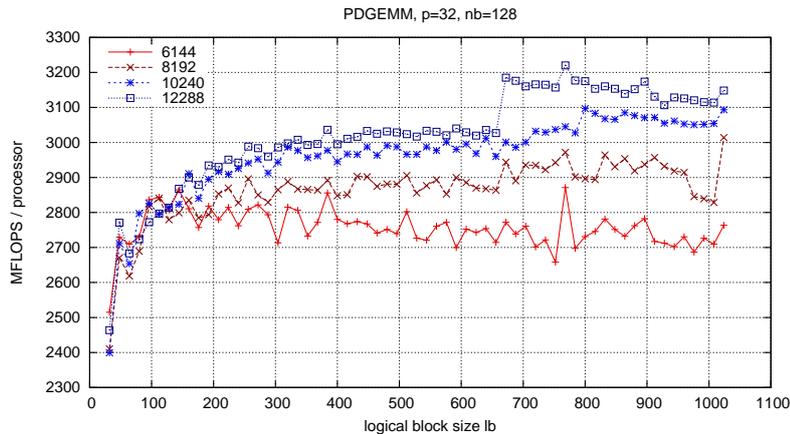
**Abb. 2.20:** Leistung von PDGEMM für mehrere Matrixgrößen (Linien) bei Anwendung verschiedener Blockfaktoren  $nb$  (x-Achse). Logische Blockgröße  $lb = 128$  (links) und  $lb = 512$  bei 64 Prozessoren.

gleichmäßige Datenverteilung auf alle Prozessoren gewährleistet ist (alle Prozessoren speichern dieselbe Anzahl an Elementen). Dadurch wird auch eine ausgewogene Arbeitslast während der Ausführung von PDGEMM erzeugt. Als Fazit kann aus dieser Betrachtung die Erkenntnis gewonnen werden, darauf zu achten, die Blockgröße als Teiler der Matrixdimension zu wählen. Im Zweifelsfall könnte man deshalb einfach einen Blockfaktor von 1 nehmen, was aber einen erheblichen Aufwand bei der initialen Verteilung bedeutet.

Es ist nicht überraschend, dass eine hohe Leistung erzielt wird, wenn die Matrixdimension ein Vielfaches der Blockgröße ist. Mit den durchgeführten Experimenten konnte gezeigt werden, dass eine Anpassung der Blockgröße zu weniger Leistungsschwankungen bei verschiedenen Matrixgrößen führt. Trotzdem konnte die Veränderung der Blockgröße nicht das grundsätzliche Performancedefizit von PDGEMM erklären, das pro Prozessor deutlich unter den zu erwarteten Werten lag (welche z. B. auch von `tpMM` erreicht werden konnten). Deshalb muss die Suche nach Gründen auf die Analyse der Performanceabhängigkeit von der logischen Blockgröße ausgedehnt werden.

### Einfluss der logischen Blockgröße auf die Leistung von PDGEMM

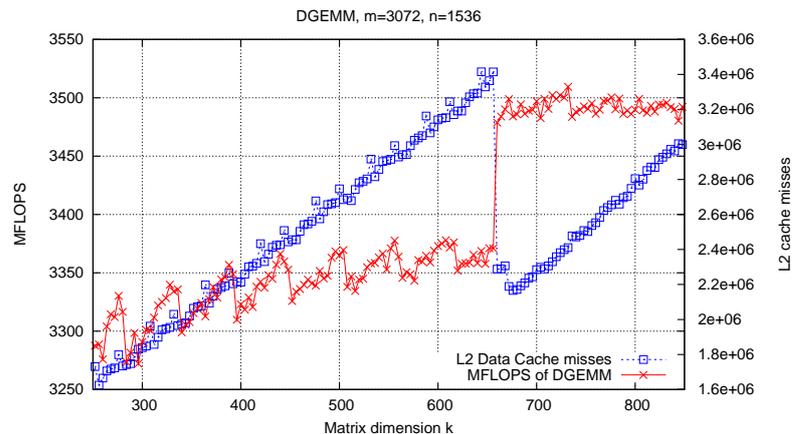
Um die Abhängigkeit von PDGEMM von der logischen Blockgröße  $lb$  aufzuschlüsseln, wurde eine Serie von Matrixmultiplikationstests mit variierendem  $lb$ -Wert durchgeführt. Da der Wert von  $lb$  fest im `ScaLAPACK`-Code, in der Datei `pilaenv.f`, codiert ist und die API nicht geändert werden sollte, wurde für jeden Test der Wert geändert und die Bibliothek neu übersetzt. Die Ergebnisse dieser Studie für 32 Prozessoren sind in Abb. 2.21 dargestellt. Es ist leicht zu erkennen, dass sich die Leistung von PDGEMM im Mittel bei größerer logischer Blockgröße insbesondere für große Matrizen erhöht. Bei genauerer Betrachtung fällt der starke Anstieg der Leistung bei Blockgröße 672 für die Matrixdimension 12288 auf. In diesem Fall speichert jeder der  $32 = 4 \times 8$  Prozessoren des Gitters  $12288/4 = 3072$  Zeilen und  $12288/8 = 1536$  Spalten der Matrix (wenn die Blockgröße



**Abb. 2.21:** Leistung von PDGEMM in MFLOPS für unterschiedliche logische Blockgrößen  $lb$  (Matrixblockgröße  $nb = 128$ , Prozessoren  $p = 32$ , Infiniband).

$nb$  die Zahlen 3072 und 1536 teilt). Mit einer logischen Blockgröße von  $lb = 672$  führt PDGEMM eine Reihe von lokalen Matrixmultiplikationen mittels Aufruf von DGEMM durch, in denen die Matrix  $A$   $3072 \times 672$  und die Matrix  $B$   $672 \times 1536$  Elemente enthalten. Eine solche lokale Matrixmultiplikation mit Dimension 672 erreichte ca. 3500 MFLOPS auf einem Opteron Prozessor des Testsystems. Im Vergleich dazu erzielte die DGEMM-Routine bei Änderung der Größe der verketteten Dimension auf 656 nur noch 3380 MFLOPS, was einem Leistungseinbruch von ca. 4% entspricht. Um diese Unregelmäßigkeit weiter zu beleuchten, wurden die von DGEMM in beiden Fällen erzeugten Cache-Fehlzugriffe (*cache misses*) mit Hilfe der PAPI-Schnittstelle [18] protokolliert. Die Abb. 2.22 zeigt die Entwicklung der zwei in diesem Experiment untersuchten Parameter. Zum einen wird die erreichte MFLOPS-Rate angezeigt, zum anderen die dabei auftretenden Fehlzugriffe auf den Level-2-Cache. Die Grafik untermauert eindeutig die Vermutung, dass der Leistungsgewinn bei  $lb = 672$  in einer verbesserten Ausnutzung des Caches begründet liegt. Wenn man einen Call-Trace des DGEMM-Aufrufs aufnimmt, kann man feststellen, dass ATLAS für die Werte von  $lb = 656$  und  $lb = 672$  verschiedene Aufrufgraphen erstellt (also intern verschiedene Funktionen anspricht). Das führt dazu, dass DGEMM unterschiedliche interne Routinen zur Lösung benutzt. Demnach sind die Routinen für den Fall  $lb = 656$  nicht optimal an die Cache-Eigenschaften des verwendeten Opteron Prozessors angepasst, was wiederum Performanceeinbußen bedeutet.

Abschließend zu diesen Untersuchungen kann festgehalten werden, dass die logische Blockgröße  $lb$  einen direkten Einfluss auf die Gesamtleistung von PDGEMM und anderen PBLAS-Routinen hat. Eine zu kleine logische Blockgröße führt zu einem erhöhten Kommunikationsoverhead und außerdem zu vielen lokalen Matrixmultiplikationen mit geringerer Leistung, da die Peak-Performance des Prozessors nicht ausgereizt werden kann (weil der Daten-Cache nicht komplett gefüllt wird). Andererseits darf die Blockgröße auch nicht zu groß gewählt werden, damit das Pipelining-Schema der PDGEMM-Routine funktioniert.



**Abb. 2.22:** MFLOPS und Anzahl der Level2-Cache-Fehlzugriffe von DGEMM (ATLAS) für unterschiedliche Werte der Dimension  $k$  ( $m = 3072$ ,  $n = 1536$ ).

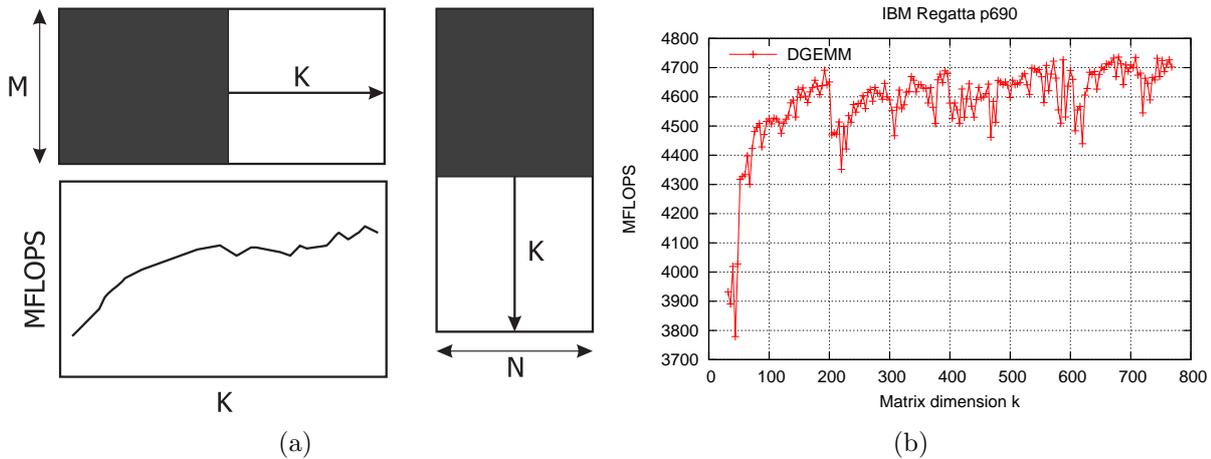
Aus den oben genannten Gründen ist es von entscheidender Bedeutung die logische Blockgröße  $lb$  an die jeweilige Maschine anzupassen. Im Folgenden wird ein Verfahren beschrieben, welches die logische Blockgröße für eine Zielplattform empirisch optimiert.

### 2.5.5 Automatische Bestimmung der logischen Blockgröße

Wie bereits gesehen, hängt eine gute Wahl der logischen Blockgröße für die PBLAS-Funktionen sehr stark von den spezifischen Eigenschaften der Zielplattform und von der BLAS-Implementierung ab. Beide sind oft komplex und meist nicht offen zugänglich, so dass es sehr schwierig ist, eine theoretische Bestimmung der logischen Blockgröße vorzunehmen. Deshalb wird ein empirisches Verfahren eingesetzt, um eine gute logische Blockgröße durch experimentell ermittelte Leistungskurven festzulegen.

Durch die vielen Einflussfaktoren auf die Leistung der DGEMM-Funktion ergibt sich ein neues Problem bei der experimentellen Bestimmung der logischen Blockgröße. Die logische Blockgröße bedingt z. B. die Netzwerkperformance des Algorithmus, die Performance der DGEMM-Funktion und die Möglichkeit Kommunikations- und Berechnungszeit zu überlappen. Deshalb ist der Wert auch abhängig von Netzwerkparametern, vom Prozessorgitter, der Anzahl der Prozessoren und der BLAS-Implementierung. Wollte man eine optimale Anpassung der logischen Blockgröße an System- und Algorithmusparameter (Matrixdimensionen, Prozessorenanzahl) erreichen, müsste man für jede einzelne Problem- und Plattformkonfiguration eine neue logische Blockgröße experimentell ermitteln. Da der Suchraum aber extrem groß ist, ist solch eine Lösung in der Praxis nicht rentabel.

Das hier vorgestellte Verfahren verwendet eine Heuristik, um die logische Blockgröße für parallele Routinen durch Tests mit einer Einprozessorfunktion zu bestimmen. Wie im vorherigen Abschnitt gezeigt, bezeichnet die logische Blockgröße im parallelen Algorithmus die verkettete Matrixdimension für die lokalen Updates. Die beiden anderen Matrixdimensionen für die lokalen Updates werden durch das Prozessorgitter und die Eingabematrix definiert. Diese Werte muss der Nutzer zu Beginn der Optimierung festlegen. Er spezifiziert, wie viele Prozessoren im Normalfall genutzt werden und wie groß die zu multiplizierenden



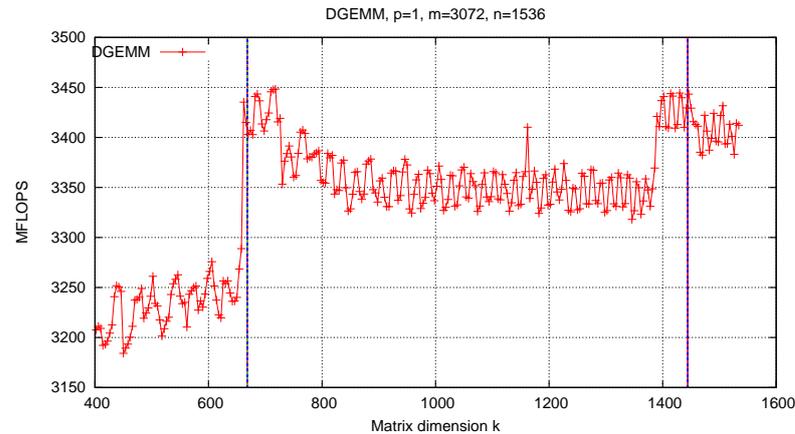
**Abb. 2.23:** (a) Erzeugung eines DGEMM-Profiles ( $C = A \cdot B$ ,  $A \in R^{m \times k}$ ,  $B \in R^{k \times m}$ ) durch Variation der Matrixdimension  $k$ . (b) Beispiel eines DGEMM-Profiles für die IBM Regatta p690.

Matrizen im typischen Anwendungsfall sind. Aus diesen Parameter lassen sich dann die Dimensionen der lokalen Matrizen berechnen. Dabei entspricht z. B. die Anzahl der Zeilen der Matrix  $A_{\text{lokal}}$  (in  $C_{\text{lokal}} = A_{\text{lokal}} \times B_{\text{lokal}}$ ) dem Wert  $M/p_r$ , wobei  $M$  die Anzahl der Zeilen der Gesamtmatrix  $A$  und  $p_r$  die Zeilen des Prozessorgitters bezeichnet. Nun wird die verkettete Dimension  $k_{\text{lokal}}$  variiert und die resultierenden MFLOPS von DGEMM für diese Parameter bestimmt. Die Werte von  $k_{\text{lokal}}$  bewegen sich zwischen 32 (Standardgröße von ScaLAPACK) und einem durch die Konfiguration gegebenen Maximum  $k_{\text{max}}$ . Für quadratische Matrizen  $A$  und  $B$  ergibt sich  $k_{\text{max}}$  als  $K/p_c$ , wobei  $K$  für Anzahl der Spalten der Ausgangsmatrix  $A$  und  $p_c$  für die Anzahl der Spalten des Prozessorgitters steht. Ist  $p_r > p_c$ , wird  $p_r$  für die Berechnung von  $k_{\text{max}}$  verwendet. Die Abb. 2.23(a) zeigt schematisch die Erzeugung eines solchen DGEMM-Profiles und Abb. 2.23(b) enthält ein Beispielpprofil für die IBM Regatta p690.

Mit Hilfe des DGEMM-Profiles lässt sich dann die logische Blockgröße für die parallele Plattform ablesen. Zuerst wird der Wert  $k_{\text{best}}$  ermittelt, mit dem die maximale DGEMM-Leistung erzielt werden konnte. Ausgehend von dieser Leistung  $l_{\text{best}} = MFLOPS(k_{\text{best}})$  wird der kleinste Wert  $k'$  gewählt, so dass mindestens eine Leistung  $x \cdot l_{\text{best}}$ ,  $0 < x < 1.0$  erreicht wird. Das Ziel ist dabei, die logische Blockgröße ausreichend groß zu wählen, um eine hohe Performance der lokalen DGEMM-Routine zu ermöglichen. Andererseits darf die logische Blockgröße auch nicht zu groß sein, um eine mögliche Überlappung von Kommunikation und Berechnung nicht zu gefährden. Damit lässt sich  $lb$  wie folgt definieren:

$$lb = \min_{32 \leq k \leq k_{\text{max}}} \{k : MFLOPS(k) \geq MFLOPS(k_{\text{best}}) \cdot x\}. \quad (2.26)$$

Nun stellt sich noch die Frage, wie  $x$  zu wählen ist. In den durchgeführten Experimenten hat sich gezeigt, dass ein Wert von  $x = 0.98$  zu einer guten Wahl von  $lb$  führt.

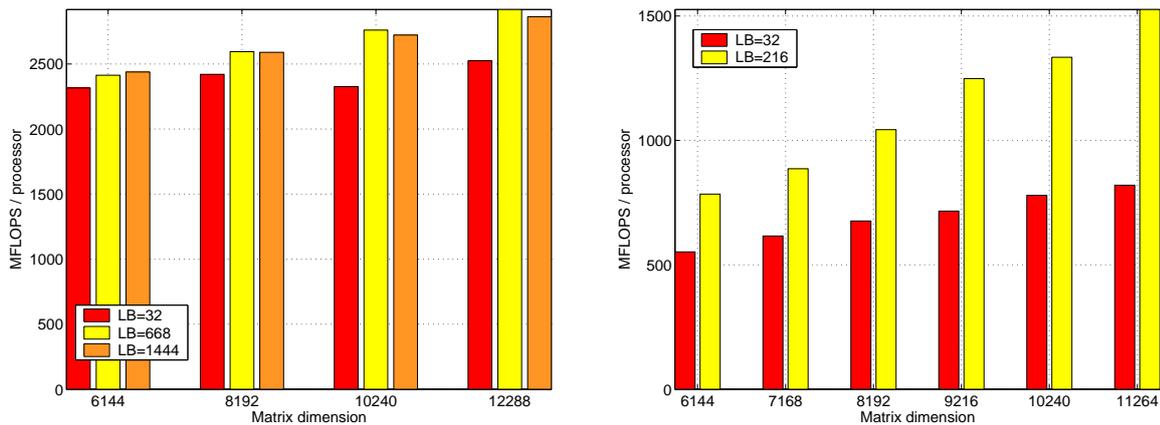


**Abb. 2.24:** Leistungsspektrum von DGEMM (ATLAS) für die Matrixdimension  $k$ . Die vertikalen Linien bezeichnen die Werte 668 und 1444.

## 2.5.6 Experimentelle Auswertung

Nach der Entwicklung der Methode zur automatischen Bestimmung der logischen Blockgröße wird diese im Folgenden experimentell ausgewertet. Dieser Evaluation liegt das Cluster-System aus Tab. 2.9 als Zielplattform zu Grunde. In den Experimenten werden 32 Prozessoren, angeordnet in einem rechteckigen Prozessorgitter (4 Zeilen, 8 Spalten), eingesetzt. Außerdem müssen die Matrixgrößen definiert werden, die einen typischen Anwendungsfall charakterisieren sollen. Dazu wurden quadratische Matrizen mit Dimension 12 288 gewählt. Bei dem  $4 \times 8$ -Prozessorgitter hält damit jeder Prozessor  $3072 \times 1536$  Elemente. Daraus ergibt sich, dass die lokalen Matrixmultiplikationen von PDGEMM Teilmatrizen der Größe  $3072 \times lb$  (Matrix  $A$ ) sowie  $lb \times 1536$  (Matrix  $B$ ) verwenden. Der Tuning-Algorithmus erstellt für diese Konfiguration ein DGEMM-Profil für  $8 \leq lb \leq 1536$ . Ungerade Werte von  $lb$  werden dabei nicht betrachtet. Das DGEMM-Profil für 32 Prozessoren auf dem **Opteron-Cluster** ist in Abb. 2.24 dargestellt. Für eine detaillierte Ansicht wurde auf kleinere Werte von  $lb = k < 400$  verzichtet. Das DGEMM-Profil weist eine maximale MFLOPS-Rate bei  $k = 1444$  auf. Gemäß der Gleichung (2.26) ergibt sich dann, dass der kleinste Wert von  $k$ , welcher mindestens 98 % der maximalen Leistung erreicht, bei 668 zu finden ist. Dieser Wert  $k = 668$  ist demnach die optimierte logische Blockgröße für den **Opteron-Cluster**. Der optimierte Wert liegt interessanterweise nahe dem vorher diskutierten kritischen Wert von 656, bei dem die DGEMM-Leistung noch weit vom Optimum entfernt war.

Mit der optimierten logischen Blockgröße kann die Leistung von PDGEMM vor und nach dem Tuning verglichen werden. Die linke Grafik in Abb. 2.25 zeigt die Leistung von PDGEMM auf dem **Opteron-Cluster** mit drei verschiedenen Werten von  $lb$ , der Standardgröße von ScaLAPACK  $lb = 32$ , der optimierten Blockgröße  $lb_{opt} = 668$  und der Blockgröße 1444, für welche DGEMM die maximale Leistung erreichte. Es ist deutlich zu erkennen, dass PDGEMM bei Verwendung der optimierten Blockgröße von 668 eine Verbesserung der Leistung erfährt. Man kann außerdem feststellen, dass der relative Leistungsunterschied zwischen optimierter und nicht optimierter Variante mit steigender Matrixdimension anwächst. Für die Matrixgrößen von 10 240 und 12 288 liegt der Leistungsgewinn der



**Abb. 2.25:** Leistung von PDGEMM bei 32 Prozessoren in MFLOPS. Links: Opteron-Cluster (Infiniband), logische Blockgrößen  $lb = 32, 668, 1444$ ; Rechts: XEON-SCI-Cluster (SCI-Netzwerk), logische Blockgrößen  $lb = 32, 216$ .

optimierten Variante gegenüber der nicht optimierten Variante zwischen 15 und 18%. Interessant ist aber auch die Betrachtung der Leistung von PDGEMM mit  $lb = 1444$ . Trotz einer besseren lokalen Leistungsfähigkeit bei DGEMM lässt sich keine Leistungssteigerung von PDGEMM mit  $lb = 1444$  feststellen. Das untermauert die These, dass ein zu großer Wert von  $lb$  zu schlechteren Kommunikationseigenschaften des Algorithmus führt.

Ein weiterer Test der automatischen Bestimmung der logischen Blockgröße wurde auf einem anderen Linux-Cluster durchgeführt, welcher aus 16 Knoten mit je 2 Xeon-Prozessoren (2 GHz) besteht (siehe XEON-SCI-Cluster in Anhang B). Als Kommunikationsnetzwerk kam ein SCI-Netzwerk zum Einsatz. Die rechte Grafik in Abb. 2.25 vergleicht die MFLOPS pro Prozessoren von PDGEMM mit optimierter ( $lb = 216$ ) und nicht optimierter ( $lb = 32$ ) logischen Blockgröße. Auch in diesem Test übertrifft die optimierte Variante die Standardvariante deutlich. Bei einer Matrixgröße von 11 264 Elemente pro Dimension wurde sogar ein Geschwindigkeitsgewinn von 47% erzielt.

## 2.5.7 Fazit

Ein wichtiges Ergebnis dieser Studie ist, dass die Leistung der PBLAS-Routinen aus ScaLAPACK, wie PDGEMM, stark von der gewählten logischen Blockgröße abhängt. Es wurde gezeigt, wie man die logische Blockgröße automatisch an das Zielsystem anpassen kann. Die Laufzeittests haben belegt, dass es zu einem signifikanten Leistungsgewinn durch die Optimierung der logischen Blockgröße kommen kann.

## 2.6 Zusammenfassung und Fazit

In diesem Kapitel wurden verschiedene taskbasierte gemischt-parallele Algorithmen zur Matrixmultiplikation für homogene verteilte Parallelrechner vorgestellt. Durch das Umstrukturieren von Algorithmen in task- und datenparallele Anteile kann ein Algorithmus in manchen Fällen besser auf eine parallele Zielplattform abgebildet werden. Im Fall von tpMM lassen sich so Kommunikationskosten durch eine verbesserte Ausnutzung von Multicore- oder SMP-Systemen reduzieren. Es wurden außerdem hierarchische Algorithmen zur Matrixmultiplikation entworfen, die sich aus verschiedenen algorithmischen Bausteinen zusammensetzen. Für jede algorithmische Schicht wird dabei ein Algorithmus gewählt, der für eine gegebene Anzahl von Prozessoren und Größe der Eingabematrizen eine gute Laufzeit erreichen kann. Die durch Kombination verschiedener Algorithmen entstandenen hierarchischen Verfahren erzielten für viele Testfälle sehr gute und sogar bessere Ergebnisse als die besten datenparallelen Implementierungen wie PDGEMM. Eine Kombination, die zu einer sehr guten parallelen Leistungsfähigkeit führte, besteht aus einer taskparallelen Variante des Algorithmus von Strassen, der nachfolgenden Anwendung der Variante „Ring“ ( $k$ -Panel-Updates) und dem abschließenden Aufruf der BLAS-Routine DGEMM (z. B. aus ATLAS). Des Weiteren wurde gezeigt, wie die Leistung der PBLAS-Routine PDGEMM für eine Zielplattform verbessert werden kann. Dazu wurde ein Verfahren angegeben, dass die von PDGEMM verwendete logische Blockgröße empirisch bestimmt. Durch diese Optimierung der logischen Blockgröße konnte die Leistungsfähigkeit von PDGEMM in allen betrachteten Fällen verbessert werden.

Die durchgeführten Experimente auf Beowulf-Clustern, speziell mit tpMM, haben auch gezeigt, dass die räumliche Lokalität von M-Tasks eines gemischt-parallelen Programms noch besser ausgenutzt werden kann. Eine mögliche Verbesserung wäre, eine schnelle Intra-Knoten-Kommunikation zu gewährleisten, wenn eine M-Task auf einen SMP-Knoten abgebildet wird. Dazu wird im nächsten Kapitel die Laufzeitumgebung vShark vorgestellt, die speziell zur Reduzierung der Kommunikationszeit zwischen Prozessoren eines SMP-Knotens entwickelt wurde.

Eine Formulierung von Algorithmen der linearen Algebra mittels Tasks oder M-Tasks bietet auch die Möglichkeit, jede Task auf unterschiedlichen parallelen Systemen im Grid auszuführen. Es wäre z. B. denkbar, dass jede der vier Tasks ( $T_{C_{ij}}, 1 \leq ij, \leq 2$ ) der vorgestellten Matrixmultiplikation nach Strassen auf einem dedizierten Cluster ausgeführt wird. Um solch eine verteilte Ausführung eines taskparallelen Programms im Grid zu erlauben, bedarf es einer Laufzeitumgebung, die eine notwendige Co-Allokation von Ressourcen für ein Programm ermöglicht. Aus diesem Grund wird in Kapitel 4 die Grid-Laufzeitumgebung TGrid vorgestellt, die speziell zum Ausführen dieser taskbasierten Algorithmen in heterogenen Umgebungen entworfen wurde.

# Kapitel 3

## Laufzeitunterstützung für M-Task-Programme auf homogenen SMP-Systemen

*Writing in C or C++ is like running a chain saw with all the safety guards removed.*

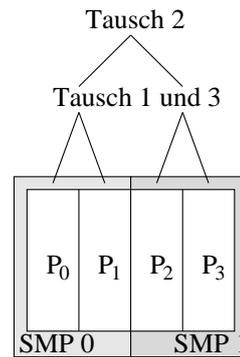
---

BOB GRAY

Dieses Kapitel beschreibt die Laufzeitumgebung vShark, die eine effiziente Ausführung von M-Task-Programmen auf Clustern von SMP- oder Multicore-Knoten ermöglicht.

### 3.1 Motivation und Ziele

Ein wichtiges Kriterium für die performante Ausführung eines M-Task-Programms auf homogenen parallelen Rechnersystemen ist die speicherlokale Abbildung der Tasks auf die Prozessoren. Beim Programmieren mit der TLib-Bibliothek, vgl. Abschnitt 1.1.2, kann der Programmierer auf die Zuordnung der Prozessoren zu den Tasks Einfluss nehmen, indem man z. B. Prozessoren mit Hilfe von Farbgebung in Gruppen anordnet. Das Ausnutzen der räumlichen Lokalität kann die Laufzeit vieler M-Task-Programme verbessern, z. B. können davon die M-Tasks der taskparallelen Matrixmultiplikation (tpMM, Kapitel 2.3) profitieren. Die Abb. 3.1 veranschaulicht noch einmal die Arbeitsweise von tpMM bei der Ausführung mit vier Prozessoren. Da die Berechnungs- und Kommunikationsschritte einer Baumstruktur entsprechen, werden zwischen benachbarten Prozessoren Daten öfter ausgetauscht als zwischen entfernten Prozessoren. Die „Entfernung“ entspricht aber der logischen Identifikationsnummer der Prozessoren. Es ist durchaus möglich, dass zwei Prozessoren von zwei unterschiedlichen SMP-Knoten einer M-Task zugeordnet werden. In diesem Fall würden Transferoperationen – in Abb. 3.1 als Tausch 1 und Tausch 3 bezeichnet – über SMP-Grenzen hinaus benötigt werden. Es sei aber angenommen, dass die Prozessoren  $P_0$  und  $P_1$  auf eine gemeinsame M-Task abgebildet werden. In dieser Konfiguration werden mehr Intra-SMP-Transfers ausgeführt, was zu einer besseren Laufzeit führen kann. Trotzdem ist dies aus technischer Sicht oft nicht effizient durchzuführen. Das Problem liegt sehr oft in der Implementierung der MPI-Bibliothek. In den meisten Fällen ist die erreichte Bandbreite der Intra-Knotenkommunikation von MPI-Implementierungen im



**Abb. 3.1:** Datenverteilung der Matrix  $B$  mit zwei SMP-Knoten und vier Prozessoren bei tpMM.

Vergleich zu den Speicherbandbreiten der einzelnen SMP-Knoten relativ gering. Für Intra-Knotenkommunikationen werden z. B. gemeinsame Speicherbereiche beim Betriebssystem reserviert oder die Kommunikation wird über das eigene Ethernet-Interface zurück gekoppelt. Beide Varianten weisen deutliche Overheads auf, die speziell bei M-Task-Algorithmen wie tpMM negativen Einfluss haben. Bei einem Ping-Pong-Experiment mit MPICH 1.2.5 auf einem Dual-Xeon-System mit 2 GHz wurde bei einer Nachrichtengröße von 1 MB lediglich ein Durchsatz von ca. 270 MB/s erzielt. Da die Speicherbandbreiten der Prozessoren oft deutlich über 1 GB/s liegen, kann eine verbesserte Intra-Knotenkommunikation die Gesamtperformance des parallelen Programms verbessern.

Aufgrund dessen wird in dieser Arbeit eine Software-Bibliothek mit dem Ziel entwickelt, die Intra-Knotenkommunikation zu optimieren. Es gibt mehrere mögliche Ansätze dies zu realisieren:

1. Man kann eine freie Kommunikationsbibliothek (z. B. MPICH für MPI) an die eigenen Anforderungen anpassen, oder
2. man entwickelt eine neue Kommunikationsbibliothek, die auf verfügbaren Bibliotheken aufsetzt und über spezielle Adapter existierenden Code wiederverwendet.

Die Variante 1 hat den Vorteil, dass man die Software für die Zielplattform optimieren kann. Es gibt aber auch den Nachteil, dass sie eben genau für diese Zielplattform entwickelt wurde. Deshalb lässt sich die Software nur schlecht auf andere Systeme portieren. Außerdem muss die eigene Weiterentwicklung immer dann angepasst werden, wenn die Ursprungssoftware in einer neuen Version freigegeben wird. Diese Nachteile werden von Variante 2 behoben, allerdings auf Kosten der Anpassungsmöglichkeiten an das Zielsystem. Nach Abwägen der Vor- und Nachteile beider Optionen wurde Variante 2 als Implementierungsziel ausgewählt.

Im Kontext dieser Arbeit heißt das, dass eine Kommunikationsbibliothek entwickelt werden soll, welche bestehende Kommunikationsdirektive wie die MPI-API benutzt. Da die für M-Task-Programme verwendete TLib-Bibliothek MPI nutzt, ist das erste Ziel, die neue Kommunikationsbibliothek auf einer MPI-Bibliothek aufzusetzen. Das Hauptziel beim Design der Kommunikationsbibliothek ist die Verbesserung der Intra-Knotenkommunikation

von MPI. Das soll dadurch erreicht werden, dass der Datenaustausch auf einem Knoten direkt über den gemeinsamen Speicher eines Prozesses stattfindet. Dazu müssen die ursprünglichen MPI-Prozesse als Threads abgebildet werden. Jedem Prozessor wird deshalb eine Anzahl von Berechnungsthreads zugeordnet. Müssen Daten zwischen zwei Berechnungsthreads auf einem SMP-Knoten transferiert werden, können die Daten schnell im gemeinsamen Speicherbereich kopiert werden. Sollen Daten zwischen zwei Berechnungsthreads ausgetauscht werden, die nicht auf demselben SMP-Knoten liegen, werden die Daten via MPI über das Netzwerk übertragen. Da aktuelle MPI-Implementierungen keine Unterstützung für die Adressierung von Threads bieten, ist es eine Hauptaufgabe dieser Arbeit, die neue Kommunikationsschicht so zu gestalten, dass sie auch mit prozessorientierten MPI-Bibliotheken korrekt arbeitet.

## 3.2 Verwandte Arbeiten

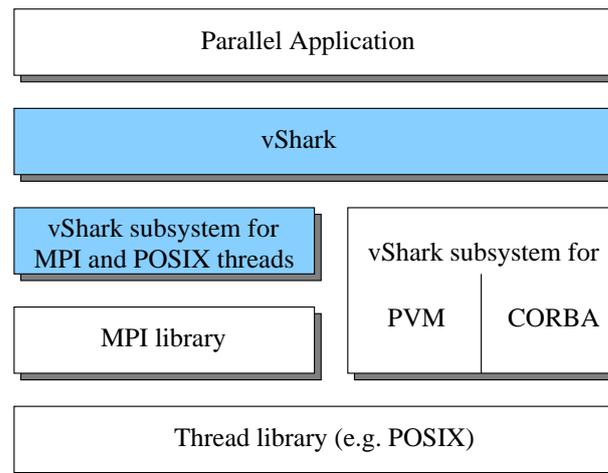
Die Verwendung und die Vorteile von Threads in Message-Passing-Bibliotheken wie MPI wurde bereits einschlägig untersucht. Ein Beispiel ist die Thread-sichere MPI-Bibliothek für Solaris [102] von Sun Microsystems, bei der Threads gleichzeitig MPI-Funktionen aufrufen können. Allerdings dürfen als Sender oder Empfänger nur Identifikationsnummern von Prozessen verwendet werden. Deshalb müssen Threads Nachrichten eindeutig identifizieren (mit Tags versehen, *tagging*), um Konflikte zu vermeiden. Verschiedene Ansätze zur Verwendung von Threads innerhalb von MPICH werden in [84] vorgestellt. Die Arbeit [53] zeigt, wie man durch die Kombination von MPI und Threads die Geschwindigkeit von irregulären Algorithmen auf verteilten Systemen verbessern kann. Es können zwei grundsätzliche Typen der Thread-Programmierung auf verteilten Systemen unterschieden werden. Eine Möglichkeit ist, einen virtuell gemeinsamen Adressraum des verteilten Systems zu erzeugen. Dabei sieht der Entwickler nur einen großen gemeinsamen Speicher und damit sinkt die Komplexität der Programmierung, speziell von Message-Passing-Code. MuPC ist ein Beispiel für einen solchen Ansatz [94]. Einen anderen Ansatz verfolgt die Erweiterung des POSIX-Threadmodells mit Message-Passing-Funktionen wie z. B. Chant [52]. Eine Implementierung von MPI, die komplett auf Threads aufsetzt, wurde in [33] beschrieben. Diese Implementierung ist aber eher zum Testen und Debuggen von MPI-Anwendungen auf einem einzelnen Rechner gedacht als für einen produktiven Einsatz auf Großrechnern. Die Arbeiten [82] und [105] gehen einen Schritt weiter und ändern einige Teile von MPICH, um das Prozessmodell komplett durch ein Threadmodell zu ersetzen. Ihr Nachteil ist die Abhängigkeit vom Betriebssystem und der MPI-Bibliothek. Eine weitere Multi-Thread-Implementierung von MPI ist AMPI [54]. AMPI verwendet eine ähnliche Notation wie die hier vorgestellte Bibliothek vShark. AMPI benutzt auch virtuelle Prozessoren, welche durch leichtgewichtige Threads realisiert werden und einen eigenen privaten Adressraum besitzen. AMPI optimiert die Zuordnung der virtuellen Prozessoren zu realen Prozessoren des Systems. AMPI versucht die Komplexität der Programmierung von parallelen Systemen zu reduzieren, wenn sich die Anzahl der Prozessoren nicht gut für eine einfache Umsetzung von Algorithmen eignet (z. B. rechteckiges Prozessorgitter nicht möglich). Ferrari et al. entwickelten TPVM als Multi-Thread-Variante von PVM [42]. Ähnlich wie

vShark benutzt TPVM Threads als parallele Basiseinheiten und die Kommunikation zwischen den Threads wird durch explizites Message-Passing realisiert. Die Threads werden dabei durch eine eindeutige Nummer identifiziert. Wie bei den veränderten Varianten von MPICH ist TPVM an eine spezifische PVM-Implementierung und an ein Betriebssystem gebunden. Das Virtual Machine Interface (VMI) [78] kann wie vShark verschiedene Kommunikationspfade zur Kommunikation auswählen. Es werden z. B. die Verwendung des gemeinsamen Speichers, TCP/IP oder Myrinet als Kommunikationsschichten unterstützt. Im Unterschied zu dem hier vorgestellten vShark ist VMI eine Middleware, die zwischen MPI und Netzwerkkarte sitzt, für welche Adapter geschrieben werden müssen.

### 3.3 Architektur der Bibliothek

Da vShark die Intra-Knotenkommunikation von Message-Passing-Bibliotheken – vorrangig MPI – zu optimieren versucht, muss die vShark-Bibliothek dem Programmierer dieselbe Sicht auf das System geben, wie die Bibliothek auf die es aufsetzt. Der Programmierer braucht explizite Kontrolle über ein verteiltes paralleles System, in dem jeder Prozessor seinen eigenen privaten Speicher besitzt. Der Hauptunterschied besteht dann in der internen Behandlung der SMP-Knoten. In den meisten MPI-Implementierungen werden auf jedem Knoten zu Beginn mehrere Prozesse gestartet. Demzufolge muss für den Austausch von Daten auf einem Knoten zwischen diesen Betriebssystemprozessen kommuniziert werden. Im Gegensatz dazu wird bei vShark für jeden SMP-Knoten eine entsprechende Anzahl von Threads erzeugt. Diese Threads werden im Folgenden als *virtuelle Prozessoren* oder Arbeitsthreads bezeichnet. Ein vShark-Programm in Ausführung besteht deshalb aus einer Anzahl von virtuellen Prozessoren, die Daten mittels Message-Passing austauschen können. Um z. B. ein MPI-Programm in ein vShark-Programm zu überführen, hat der Programmierer nur die MPI-Methodenaufrufe durch die äquivalenten Funktionen von vShark zu ersetzen. In Abhängigkeit von der Zielarchitektur bildet das vShark-Laufzeitsystem die virtuellen Prozessoren auf reale Prozessoren ab.

Systeme mit verteiltem Speicher benötigen ein Message-Passing-Protokoll, um Daten zwischen Prozessoren zu übertragen. In der Laufzeitumgebung von vShark wird die Kommunikation durch einen separaten Kommunikationsthread, dem *Kommunikator*, gesteuert. Ein Cluster-Knoten hat genau einen Kommunikator, welcher den Kommunikationskanal zwischen den virtuellen Prozessoren bildet. Der Vorteil von vShark ist die Thread-basierte Programmierumgebung. Der Programmierer braucht deshalb sein Message-Passing-Programm nicht an eine spezielle Zielplattform anzupassen. Oft werden MPI-Programme für größere SMP-Knoten hybridisiert, d. h. um schnelleren Datenaustausch auf einem SMP-Knoten zu erreichen, werden mehrere Threads gestartet. Diese komplexe Anpassung von Programmen entfällt bei der Anwendung von vShark, da vShark den besten Kommunikationskanal für zwei virtuelle Prozessoren automatisch wählt. Da vShark die Lage der virtuellen Prozessoren im verteilten System kennt, kann der Kommunikationsoverhead für Intra-Knotenkommunikation verkleinert werden. Anstatt teure Inter-Prozess-Kommunikation auf einem SMP-Knoten nutzen zu müssen, werden Daten direkt zwischen Threads kopiert.



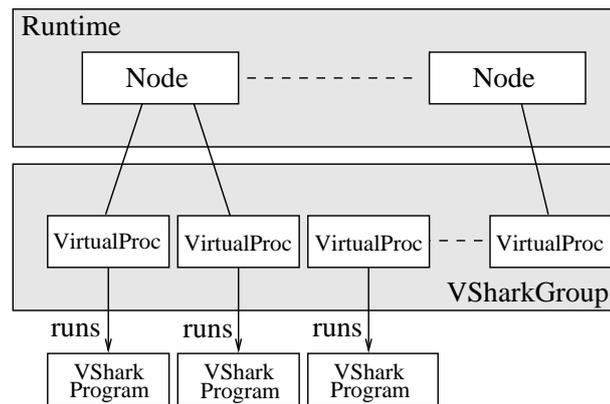
**Abb. 3.2:** vShark-Architektur. Die dunkel gefärbten Schichten sind bereits im vShark-Stack implementiert.

Die Abb. 3.2 zeigt den hierarchischen Aufbau der vShark-Laufzeitumgebung. Das parallele Programm in der obersten Schicht nutzt direkt Funktionen der vShark-API. Der Programmierer sollte nur diese obere vShark-API nutzen, um z. B. andere vShark-Subsysteme nutzen zu können. Neben einer Version für MPI und POSIX-Threads könnte z. B. auch eine für CORBA und POSIX-Threads implementiert werden. Die Programme müssten somit nicht neu angepasst werden, wenn eine andere Zielplattform und ein anderes Subsystem gewählt wird. Die vShark-Bibliothek verwendet ein objektorientiertes Design. Besonders wichtig ist die von vShark verwendete Typsicherheit der API, was besonders für die Korrektheit von verteilten Applikationen von entscheidender Bedeutung ist.

Die Ausführung eines vShark-Programms beginnt mit dem Starten einer vorher definierten Anzahl von virtuellen Prozessoren auf jedem physikalischen SMP-Knoten. Jeder virtuelle Prozessor wird mit dem zu dem SMP-Knoten gehörenden Kommunikator verbunden, über den er mit anderen virtuellen Prozessoren im System kommunizieren kann. Damit eine Kommunikationsoperation durchgeführt werden kann, muss ein virtueller Prozessor eine Anfrage beim Kommunikator stellen. Diese Anfrage wird intern an eine Warteschlange des Kommunikators angehängt. Der entsprechende Kommunikator arbeitet die Anfragen nacheinander ab und benachrichtigt den virtuellen Prozessor, wenn die Operation beendet ist. Wie jedes Subsystem die Kommunikation intern realisiert, kann von Fall zu Fall unterschiedlich sein. Eine detaillierte Beschreibung der Funktionsweise der Implementierung mit MPI und POSIX-Threads wird in Abschnitt 3.5 gegeben.

### 3.4 Die vShark-Programmierschnittstelle

Der folgende Abschnitt gibt einen Überblick über die zentralen Klassen der C++-Bibliothek vShark und ihre Verwendung bei der Programmierung von parallelen Anwendungen. Die Hauptklassen von vShark sind `Node`, `VirtualProc`, `VSharkProgram`, `VSharkGroup` und `Runtime`. Die Abb. 3.3 stellt die logische Interaktion dieser Klassen innerhalb der vShark-



**Abb. 3.3:** Interaktion der vShark-Komponenten im Überblick.

Laufzeitumgebung dar. Für den Programmierer sind vor allen die Klassen `VSharkProgram`, `VSharkGroup` und `Runtime` interessant, da diese direkt zugreifbar sind. Die Klassen haben die folgenden Funktionen:

**Runtime** Die Klasse `Runtime` ist das zentrale Objekt, über welches der Programmierer Zugriff auf das parallele vShark-Laufzeitsystem erhalten kann. `Runtime` ist für die Initialisierung des Laufzeitsystems verantwortlich, z. B. für das Lesen der Konfigurationsdateien und für das Erzeugen der relevanten Entitäten wie `Node`.

**Node** Diese Klasse bildet einen physikalischen SMP-Knoten virtuell im Laufzeitsystem ab. Ein `Node` besitzt einen oder mehrere virtuelle Prozessoren. Die Aufgabe des Objekts `Node` ist, den Kommunikator-Thread und die definierte Anzahl an virtuellen Prozessoren zu erzeugen.

**VirtualProc** In realen parallelen verteilten Systemen besteht jeder SMP-Knoten aus einer festen Anzahl an Prozessoren. Die Klasse `VirtualProc` ist die abstrakte Modellierung dieser Prozessoren in der vShark-Umgebung. Es kann jedoch von Vorteil sein, dass in einer Multi-Thread-Umgebung mehr virtuelle Prozessoren erzeugt werden als es physikalische gibt. Das könnte die Leistung steigern, wenn z. B. ein virtueller Prozessor mit I/O-Aufgaben beschäftigt ist und gleichzeitig ein anderer diese freie CPU-Zeit ausnutzen kann. Deshalb ist es in der vShark-Laufzeitumgebung möglich eine beliebige Anzahl an virtuellen Prozessoren auf einem SMP-Knoten zu starten.

**VSharkProgram** Alle vShark-Programme sind von der abstrakten Klasse `VSharkProgram` abgeleitet. Um ein paralleles Programm zu schreiben, muss der Entwickler nur die `run()`-Methode von `VSharkProgram` überschreiben. Das Programm wird dann von der Laufzeitumgebung an die virtuellen Prozessoren übergeben, welche die `run()`-Methode ausführen. Damit führt jeder virtuelle Prozessor das gleiche Programm aus.

**VSharkGroup** Die Klasse `VSharkGroup` hat die äquivalenten Aufgaben eines MPI-Kommunikators. Mit den Funktionen lassen sich Nachrichten schicken und empfangen. Außerdem kann der Entwickler Informationen über den Status der Gruppen des Systems abfragen, z. B. die Gesamtzahl der virtuellen Prozessoren. Die Grundfunktionen zur Kommunikation beinhalten u. a. `send`, `recv`, `bcast`, `gather` und `barrier`.

Bei allen hier genannten Klassen handelt es sich um abstrakte Klassen, d. h. sie definieren lediglich die zu verwendende Schnittstelle für vShark-Programme. Da diese Implementierung eine C++-Schnittstelle besitzt, heißt das, dass die Klassen mittels virtueller Funktionen die vShark-API festlegen. Die darunter liegende Kommunikationsschicht muss für jede Kommunikationsschnittstelle (z. B. MPI) implementiert werden.

Ein Beispielprogramm ist in Abb. 3.4 dargestellt. Zuerst wird die neue Klasse `Exp1` von `VSharkProgram` abgeleitet. In der zu überschreibenden `run()`-Methode steht dann das parallele Programm, welches von den virtuellen Prozessoren ausgeführt wird. Wie in MPI üblich, wird zu Beginn des Programms die Größe der Kommunikatorgruppe und die eigene Identifikationsnummer bestimmt. Ein Prozessor mit gerader ID schickt danach eine Nachricht zu dem Prozessor mit der nächst größeren ID (mit wrap-around). In der `main`-Funktion wird die vShark-Laufzeitumgebung instantiiert und das Programm `exp` an die virtuellen Prozessoren via `execute` übergeben. An diesem Beispiel lässt sich sehr gut die Nähe zu MPI erkennen. Dies ist gewollt, da MPI in einem sehr großen Teil der parallelen Programme zum Einsatz kommt. Außerdem soll darauf hingewiesen werden, dass die darunter liegende Kommunikationsschicht wie MPI komplett gekapselt wurde. Damit ist ein späteres Austauschen der Kommunikationsbibliothek leichter möglich.

### 3.5 Fallstudie mit MPI und POSIX-Threads

In dieser Arbeit wurde vShark unter Verwendung der POSIX-Threads und MPI in C++ implementiert. Im folgenden Abschnitt werden die Details dieser Realisierung beschrieben.

**Kommunikationsschema** Die MPI-Standards 1.1 und 2.0 gewährleisten keine Thread-Sicherheit in den Programmen. Der Standard legt lediglich die zur Verfügung stehenden Funktionen fest und gibt Implementierungshinweise für die Verwendung von Threads. Der Standard definiert die Kommunikation auf der Basis von Prozessen, d. h. die Sender und Empfänger werden mittels Prozessnummern identifiziert. Die MPI-API deckt somit nicht den Fall ab, dass ein Prozess aus mehreren Threads besteht, zumindest sind sie nicht direkt adressierbar. Für eine Thread-sichere Kommunikation muss allerdings sichergestellt werden, dass die Nachrichten auch zu den richtigen Threads zugestellt werden. Passiert dies nicht, kommt es zu sich überholenden Nachrichten und damit im schlimmsten Fall zu Speicherzugriffsfehlern, da die Nachrichten unterschiedliche Größe haben.

Aus diesem Grund muss vShark die Aufgabe der Sicherstellung der Thread-sicheren Kommunikation zwischen virtuellen Prozessoren übernehmen. Thread-Sicherheit kann in diesem Kontext erreicht werden, wenn zu einem Zeitpunkt nur ein Thread pro Knoten I/O-Operationen mit MPI durchführt. Verschiedene mögliche Lösungsansätze wurden bereits in der Literatur erwähnt, z. B. das Schützen aller MPI-Aufrufe durch globale Locks, um wechselseitigen Ausschluss (*mutual exclusion*) sicherzustellen [53]. Ein anderer Ansatz ist es einen Hilfsthread zu verwenden, der alleinig die Aufgabe der Kommunikation übernimmt. Dieser Thread ist damit der einzige Thread pro Knoten, der Zugriff auf die Kommunikationsschicht (MPI) hat. vShark benutzt solch einen Kommunikator-Thread. Dieser ausgezeichnete Kommunikator stellt nicht nur die Threadsicherheit der Kommu-

```

class Expl : public VSharkProgram {
public:
    void run(VirtualProc *proc);
};

void Expl::run(VirtualProc *proc) {
    Runtime& re = get_runtime();
    VSharkGroup *group = re.get_group();

    int rank = group->get_rank();
    int p = group->get_size();
    Message *msg;

    if( rank % 2 == 0 ) {
        msg = new IntMessage(&rank, 1, 0);
        group->send(group->create_envelope
                    (msg, rank, (rank+1)%p));
    } else {
        int neighbour;
        msg = new IntMessage(&neighbour, 1, 0);
        group->recv(group->create_envelope
                    (msg, (rank-1+p)%p, rank));
        cout << "left_neighbour_of_" << rank
              << "_has_rank_" << neighbour << endl;
    }
    delete msg;
}

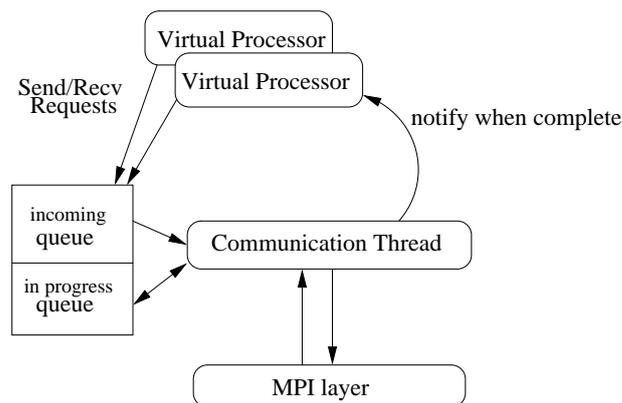
int main(int argc, char *argv[]) {
    Runtime re(argc, argv);
    Expl exp;
    re.execute(exp);
    re.shutdown();
    return 0;
}

```

**Abb. 3.4:** Einfaches Send/Recv-Programm, implementiert mit der vShark-API.

nikation sicher, er erlaubt es auch, die Kommunikationskanäle zur Laufzeit anzupassen. Damit ist es möglich, dass er zwischen Intra-Thread-, shared-memory- oder Socketkommunikation auswählt, je nachdem, welche virtuellen Prozessoren verbunden werden sollen. Dies wäre mit einem globalen Lock-Mechanismus nicht möglich gewesen.

Eine abstrakte Sicht auf das Zusammenspiel der virtuellen Prozessoren mit dem Kommunikator ist in Abb. 3.5 dargestellt. Wenn ein virtueller Prozessor Daten senden oder empfangen soll, stellt er diese Anfrage an den Kommunikator. An dieser Stelle wird darauf hingewiesen, dass vShark die Nachrichten nicht in einen separaten Puffer kopiert, was den Speicherverbrauch extrem erhöhen würde. Stattdessen übergibt der virtuelle Prozessor die entsprechende Speicheradresse an den Kommunikator. Nachdem der Datentransfer vom Kommunikator durchgeführt wurde, benachrichtigt er die virtuellen Prozessoren über die Fertigstellung der Anfrage. Dazu wird der gemeinsamen Bedingungsvariable, welche an die Kommunikationsoperation geknüpft ist, ein Signal geschickt.



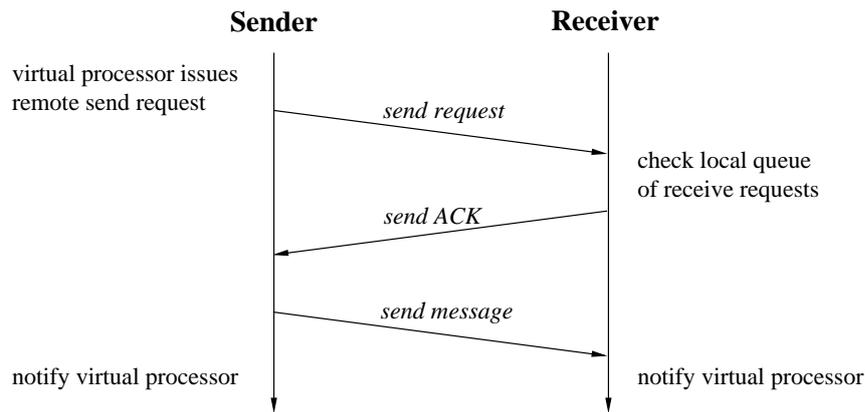
**Abb. 3.5:** Interner Aufbau der vShark-Implementierung mit MPI und POSIX-Threads.

**Datentransferprotokoll** Es gibt zwei kritische Entscheidungen bei der Realisierung des Datenübertragungsprotokolls zu treffen. Einerseits gilt es zu klären, wie oft der Kommunikator ein Polling nach neuen Nachrichten durchführen muss und andererseits, ob eine Zwischenpufferung von Daten genutzt werden sollte.

Wie schon angeklungen ist, verwendet vShark keinen zusätzlichen Datenpuffer, um den Speicherbedarf klein zu halten. Außerdem lässt sich damit ein gesonderter Aufwand für weitere Deadlock-Vermeidungsstrategien vermeiden. Solche Strategien müssten bei zusätzlicher Pufferung bedacht werden, da eine vollständig gefüllte Nachrichtenschlange zu blockierenden Threads führen kann. Außerdem könnte es zu Puffer-Überläufen kommen, wenn keine Vorkehrungen getroffen werden. Speziell bei Applikationen der linearen Algebra mit Nachrichten im Megabyte-Bereich würde eine Pufferung den Speicherverbrauch stark beeinflussen.

Das von vShark implementierte Kommunikationsprotokoll, welches Thread-Sicherheit garantiert und keine zusätzlichen Speicheranforderungen hat ist in Abb. 3.6 veranschaulicht. Der Datentransfer wird immer vom sendenden Kommunikator initiiert. Dieser Kommunikator sendet eine Anfragenachricht (mit `MPI_Send`) an den Kommunikator, der für den empfangenden virtuellen Prozessor zuständig ist. Eine solche Nachricht enthält die Identifikationsnummern des sendenden und des empfangenden virtuellen Prozessors. Der empfangende Kommunikator überprüft, ob sich schon die zugehörige Anforderung vom empfangenden virtuellen Prozessor in seiner Schlange befindet. Ist dies der Fall, schickt der Kommunikator eine Bestätigungsnachricht (*acknowledgement* = *ACK*) an den anfragenden Kommunikator und beginnt in den Modus des Datenempfangs zu wechseln (realisiert mit `MPI_Irecv`). Als Empfangspuffer wird direkt der Speicherbereich des virtuellen Prozessors verwendet. Falls der virtuelle Prozessor der Empfangsseite die Daten noch nicht angefragt hat, wird die Anfrage in eine Warteliste eingefügt. Sobald der virtuelle Prozessor die Daten anfordert, schickt der Kommunikator der Empfangsseite direkt eine *ACK*-Nachricht zur Gegenseite. Um die *ACK*-Nachrichten eindeutig zuordnen zu können, enthalten sie die Identifikatoren der virtuellen Prozessoren.

Zusammenfassend können zwei Hauptvorteile dieser Lösung angegeben werden: (1) Das Protokoll hat einen sehr kleinen Speicheroverhead (Speichern der Requests liegt im Byte-



**Abb. 3.6:** vShark/MPI-Protokoll für den Datentransfer. Rendezvous-Strategie zur Sicherstellung der Thread-sicheren Kommunikation zwischen virtuellen Prozessoren. **Sender** und **Receiver** bezeichnen jeweils einen Kommunikator.

Bereich). (2) Der sendende Kommunikator hat die Möglichkeit die ausgehenden Nachrichten umzuordnen. Dadurch lässt sich der Durchsatz optimieren, wobei natürlich die Randbedingungen wie die (von MPI vorgegebene) Nachrichtenreihenfolge und Fairness beachtet werden müssen (Nachrichten zwischen einem Prozessorpaar dürfen sich nicht überholen).

**Realisierung des Kommunikators** Wie vorher schon angeklungen, muss der Kommunikator für MPI ständig den virtuellen Postkasten auf ankommende Nachrichten überprüfen (*polling*). In einem System mit gemeinsamem Speicher lässt sich das Problem durch Bedingungsvariablen und Signalgebung lösen. Leider kann diese Lösung nicht auf verteilte Systeme übertragen werden, zumindest nicht oberhalb der MPI-Schicht. Mit MPI kann man nur aktiv testen, ob eine Nachricht eingetroffen ist. Da dieser Test CPU-Zeit erfordert, sollte er möglichst selten eingesetzt werden. Daraus ergibt sich die zentrale Frage, die die Leistung von vShark nachhaltig beeinflusst: Wie oft muss der Kommunikator auf neue Nachrichten testen oder wie lange darf er sich im Schlafmodus befinden? Die Schlafzeit muss kurz genug sein, um kleine Latenzzeiten zu ermöglichen, aber auch groß genug, damit nicht zu viele CPU-Zyklen von den Arbeitsthreads abgezogen werden. Um den Warteoverhead zu minimieren, wird die Schlafzeit (*sleep time*) als neue Variable in vShark eingeführt. Der Kommunikator schläft eine definierte Zeitspanne, wenn alle lokalen Bearbeitungswarteschlangen abgearbeitet und keine entfernten Anfragen eingetroffen sind. Wie später genauer experimentell untersucht wird, ist der Wert dieses Parameters entscheidend für die Gesamtleistung von vShark.

Während der Laufzeittests mit vShark hat sich gezeigt, dass ein fester Wert für die Wartezeit des Kommunikators nicht für alle Programme ein gutes Ergebnis liefert. Deshalb wurde ein adaptives Verfahren zum dynamischen Anpassen der Wartezeit implementiert. Die Zeit, die der Kommunikator zu einem bestimmten Zeitpunkt während der Ausführung wartet, hängt vom Kommunikationsmuster des Programms und von den gewählten Grenzen der Wartezeit ab. Diese Grenzen (minimum, maximum, default) können in Millisekunden in der Konfigurationsdatei `vshark_mpi.conf` definiert werden. Die Abb. 3.7 zeigt

```
[comm]
type=dynamic
#type=static

[comm_static]
timeout=100

[comm_dynamic]
min_timeout_ms=10
max_timeout_ms=50
std_timeout_ms=10
step_width_ms=5
```

**Abb. 3.7:** Konfigurationsdatei des vShark-MPI-Adapters.

exemplarisch eine solche Konfigurationsdatei für das MPI-Subsystem. Es gibt zwei mögliche Definitionen der Wartezeit. Man kann die Wartezeit des Kommunikators statisch festlegen. Wurde der Kommunikatortyp als `type=static` angegeben, wird die statische Wartezeit `timeout` verwendet. Die flexiblere Variante ist der Typ `type=dynamic`. Die Parameter `min_timeout_ms` und `max_timeout_ms` definieren die globalen Grenzen, zwischen denen sich die Wartezeit bewegen kann. Die Variable `std_timeout_ms` bezeichnet die Wartezeit, die zu Beginn des Programms benutzt wird. Die Schrittweite `step_width_ms` gibt die Zeit in Millisekunden an, um die die aktuelle Wartezeit vergrößert oder verkleinert werden soll. Die Wartezeit wird erhöht, wenn der Kommunikator aus dem Schlafmodus aufwacht und keine neuen Anfragen vorfindet. Ist eine Anfrage eingetroffen, wird die Wartezeit wieder verkleinert, da es wahrscheinlich ist, dass noch weitere Anfragen eintreffen werden.

**Ausführen von vShark-Programmen mit MPI-Bibliotheken** Ein vShark-Programm, welches auf einer MPI-Implementierung aufsetzt, kann wie üblich mittels `mpirun` auf jedem Knoten gestartet werden. vShark initialisiert dann auf jedem Knoten das Runtime-System, welches die vShark-Konfiguration aus der Datei `vshark.conf` verarbeitet. Diese Datei ist im Grunde äquivalent zu dem „machine file“ von MPI, d. h. sie enthält die Anzahl der virtuellen Prozessoren pro Knoten (`node: #processors`). Das Laufzeitsystem startet die angegebene Anzahl an virtuellen Prozessoren und einen Kommunikator für jeden Knoten. Das eigentliche Programm wird im Anschluss an die virtuellen Prozessoren zur Abarbeitung übergeben.

## 3.6 Experimentelle Auswertung

Die Leistungsfähigkeit der vShark-Bibliothek wird durch einen Vergleich mit äquivalenten MPI-Programmen in zwei unterschiedlichen Applikationsklassen evaluiert. Zuerst wurde das Abschneiden von vShark gegenüber MPI in unterschiedlichen Benchmark-Tests untersucht. Als Benchmarks wurden die Programme aus der ParkBench-Sammlung verwendet [79]. Da diese Benchmark-Programme in Fortran 77 geschrieben sind, mussten sie

**Tab. 3.1:** Konfiguration für den COMMS1-Benchmark.

Fall	Fortran/MPI	C++/vShark
a (1 SMP-Knoten)	2 Prozesse auf einem Knoten	2 Threads (virtuelle Prozessoren) auf einem Knoten
b (2 SMP-Knoten)	2 Prozesse 1 Prozess pro Knoten	2 virtuelle Prozessoren 1 Thread (virtueller Proz.) pro Knoten

zuerst nach C++ portiert und die MPI-Aufrufe durch die entsprechenden vShark-Funktionen ausgetauscht werden.

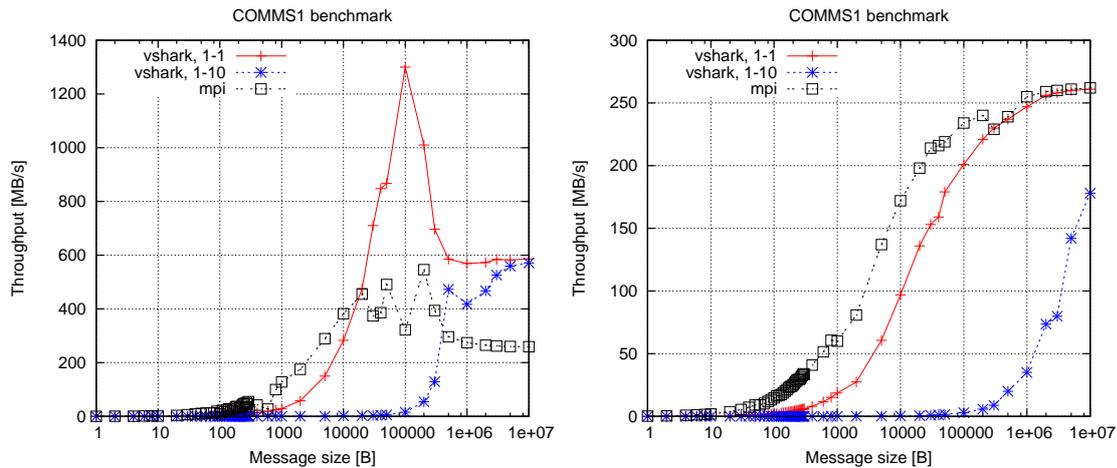
Die Tests wurden auf einem Dual-Xeon-Cluster durchgeführt (16 Knoten, 2 GHz, siehe XEON-SCI-Cluster in Anhang B). Jeder Knoten ist mit zwei Netzwerkkarten ausgestattet, einer SCI- und einer Fast-Ethernet-Karte. Als MPI-Implementierungen stehen jeweils die Bibliotheken ScaMPI [114] für SCI und MPICH für Fast-Ethernet zur Verfügung.

In den folgenden Diagrammen werden die Ergebnisse des Parkbench-Benchmarks mit „mpi“ bezeichnet. Der Bereich (x-y) hinter der Beschriftung der vShark-Programme steht für die gewählten Minima und Maxima der Wartezeit des Kommunikators, z. B. bei 1–10 wartet der Kommunikator mindestens eine und höchstens zehn Millisekunden.

**Der COMMS1-Benchmark** Der Benchmark COMMS1 realisiert einen so genannten Ping-Pong-Benchmark und gibt damit Auskunft, wie schnell eine Nachricht zwischen zwei Prozessen verschickt werden kann. Ein Master-Prozess sendet eine Nachricht mit variabler Länge zu einem Worker-Prozess, welcher die Nachricht umgehend nach Erhalt zurück schickt. Damit kann die Durchsatzrate zwischen den Prozessen bestimmt werden. Es werden zwei unterschiedliche Konfigurationen untersucht, welche in Tab. 3.1 zusammengefasst werden. Im ersten Fall werden zwei virtuelle Prozessoren auf genau einem SMP-Knoten gestartet, um den Durchsatz auf einem Knoten zu analysieren. Der zweite Test evaluiert den Durchsatz zwischen zwei virtuellen Prozessoren, die auf unterschiedliche Knoten abgebildet wurden.

Die linke Grafik in Abb. 3.8 veranschaulicht den Durchsatz, der von vShark und MPI bei der Intra-Node-Kommunikation erreicht wurde. Es ist zu sehen, dass die Kommunikation zwischen zwei MPI-Prozessen bei kleineren Nachrichten schneller als von vShark durchgeführt wird. Steigt jedoch die Nachrichtengröße, kann vShark durch die Thread-basierte Kommunikation punkten und der Overhead des Kommunikators kommt weniger zum Tragen. Die Verwendung von sehr geringen Wartezeiten (1–1) führt zwangsläufig zu einer besseren Leistung. Dieser sehr geringe Wert für das Maximum der Wartezeit ist aber für reale Applikationen nicht realistisch, da sonst zu viel Zeit für das Polling verwendet wird. Aber auch bei längeren Wartezeiten (1–10) kann vShark einen besseren Durchsatz für größere Nachrichten erzielen. Schon bei Nachrichtengrößen, die deutlich unter 1 MB liegen, weist vShark einen höheren Durchsatz auf.

Im Kontrast dazu wurde der Durchsatz zwischen zwei unterschiedlichen Knoten gemessen. Die Ergebnisse dieses Tests sind in der rechten Grafik von Abb. 3.8 veranschaulicht. Da vShark ein zusätzliches Kommunikationsprotokoll verwendet, kann der ursprüngliche



**Abb. 3.8:** Durchsatz in MB/s beim COMMS1-Benchmark: links: Intra-Node-Kommunikation (SMP), rechts: Inter-Node-Kommunikation. Plattform: XEON-SCI-Cluster.

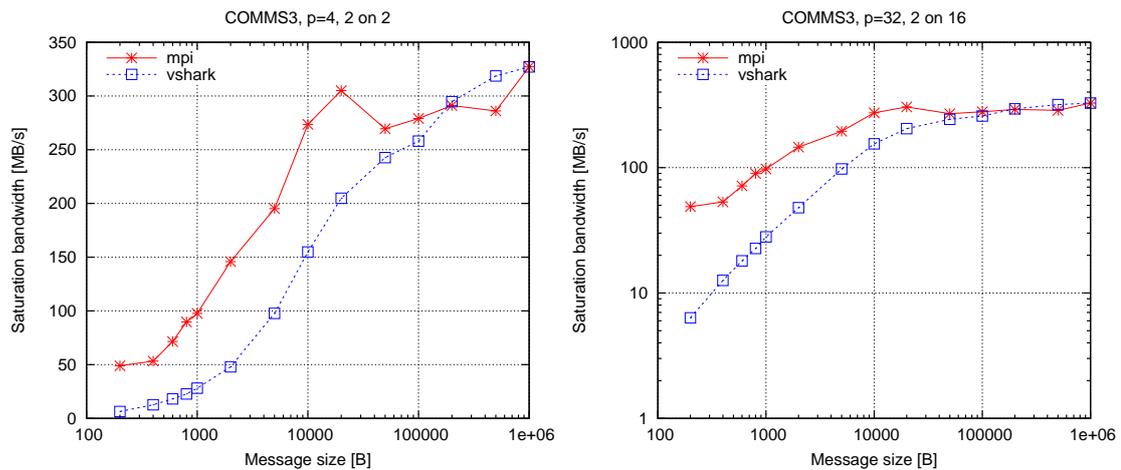
MPI-Benchmark für alle Nachrichtenlängen einen besseren Durchsatz erzielen. Das ist nicht verwunderlich, da man bedenken muss, dass der Kommunikator für jede Inter-Knotenkommunikation eine Anfragenachricht der Größe 4 Bytes + MPI-Overhead verschickt. vShark kann den Leistungsunterschied mit zunehmender Nachrichtengröße ausgleichen, da die kleinen Protokollnachrichten dort eine geringere Rolle spielen.

Nach Analyse beider Tests kann festgestellt werden, dass eine Anwendung dann von vShark profitiert, wenn viele Nachrichten innerhalb des Knotens verschickt werden und diese Nachrichten entsprechend lang sind ( $> 10$  KB).

**Der COMMS3-Benchmark** Die Website [www.top500.org](http://www.top500.org) beschreibt den COMMS3-Benchmark wie folgt: Jeder Prozessor eines Systems mit  $p$  Prozessoren schickt eine Nachricht der Länge  $n$  zu jedem der anderen  $(p - 1)$  Prozessoren. Jeder Prozessor wartet dann, bis alle  $(p - 1)$  Nachrichten an ihn eingetroffen sind. Die Zeitnahme endet, wenn alle Nachrichten von allen Prozessoren empfangen wurden.

Die Abb. 3.9 vergleicht die Bandbreiten, die mit MPI und vShark bei diesem Benchmark erreicht wurden. Wie in Abb. 3.9 (links) zu sehen, ist die mit vShark mögliche Bandbreite bei der Verwendung von vier Prozessoren (2 virtuelle Prozessoren auf 2 Knoten) nur geringfügig kleiner als die von MPI. Trotzdem kann vShark bei größeren Nachrichten ( $> 50$  KB) in dieser Konfiguration zu MPI aufschließen. Werden deutlich mehr Prozessoren eingesetzt, lässt der zusätzliche Aufwand im Kommunikationsprotokoll von vShark nur für große Nachrichten vergleichbare Bandbreiten zu, siehe Abb. 3.9 (rechts).

**Testen einer realen Applikation** Da die taskparallele Matrixmultiplikation (tpMM) eine Zielanwendung für die Entwicklung von vShark war, wird die Leistungsfähigkeit von vShark anhand dieses Algorithmus evaluiert. Die Laufzeittests wurden mit dem Dual-Xeon-Cluster und Fast-Ethernet als Verbindungsnetzwerk durchgeführt. Als MPI-Implementierung wurde MPICH 1.2.5.2 eingesetzt, bei der zuvor durch die Konfigurationsopti-

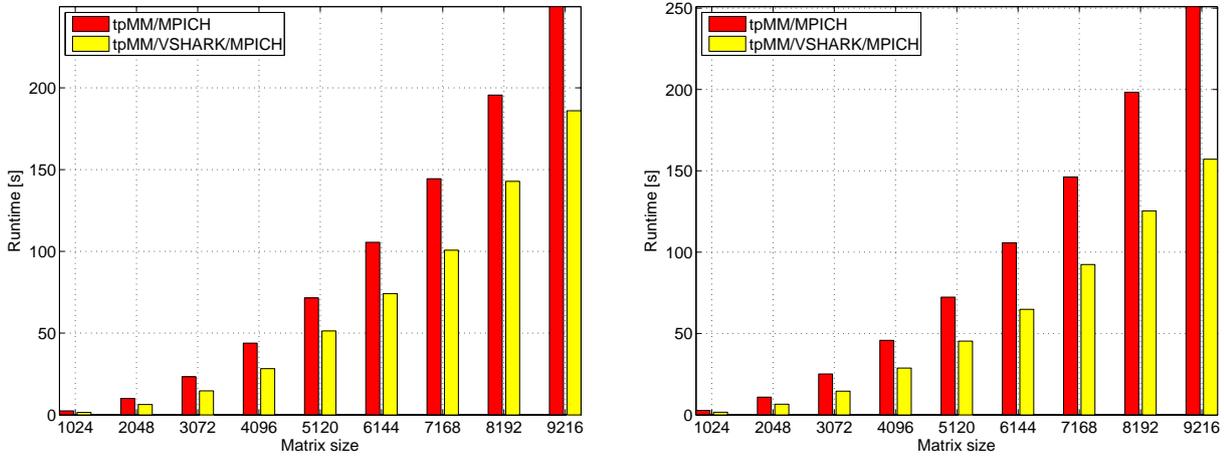


**Abb. 3.9:** Sättigungsbandbreite in MB/s (COMMS3-Benchmark, 2 Prozessoren pro Dual-SMP-Knoten, Plattform: XEON-SCI-Cluster, Polling-Timeout (statisch): 1 ms); links: für 2 Knoten ( $p=4$ ), rechts: für 16 Knoten ( $p=32$ ).

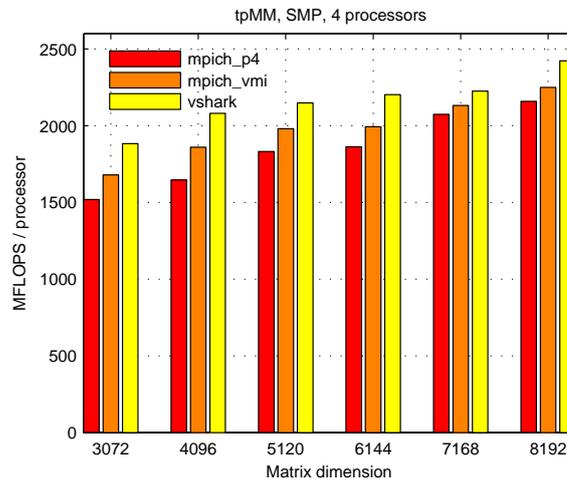
on `-with-comm=shared` der Einsatz des gemeinsamen Speichers für Intra-Knotentransfer aktiviert wurde. Die Leistung von tpMM wird aus zwei Gründen nicht mit ScaMPI verglichen. Erstens wurde vShark speziell für Beowulf-Cluster entwickelt, die meist nur eine freie MPI-Implementierung wie MPICH zur Verfügung haben. Der zweite Grund liegt in den beobachteten Leistungsunterschieden von tpMM mit vShark und MPI. Diese waren bei ScaMPI so gering, dass eine genaue Analyse nicht möglich war. Eine genauere Betrachtung wäre nur möglich gewesen, wenn man die Matrizen sehr stark vergrößert hätte. Dies war aber wiederum durch eine sehr geringe Speichergröße pro Knoten (1 GB) nicht möglich.

In den Experimenten wurden zwei Implementierungen von tpMM verglichen. Die Erste ist in C++ implementiert und verwendet vShark als Kommunikationsbibliothek. Die Zweite ist die ursprüngliche Implementierung von tpMM, die in C geschrieben wurde und direkt auf TLib und MPI aufsetzt. Die Laufzeitresultate der beiden Varianten für 8 und 16 Prozesse (MPI) oder virtuelle Prozessoren (vShark) sind in Abb. 3.10 veranschaulicht. Es ist deutlich zu erkennen, dass vShark die Laufzeit von tpMM in beiden Testfällen signifikant reduziert. Dem Algorithmus kommt speziell die schnellere Intra-Knotenkommunikation von vShark entgegen.

Die Laufzeitumgebung vShark wurde auf einem weiteren parallelen System getestet. Als Testplattform kam dazu eine 4-Wege-Xeon mit 2 GHz zum Einsatz (Linux, MPICH 1.2.5). Auf diesem System stand neben dem älteren langsamen P4-Treiber von MPICH auch der aktuellere VMI-Treiber zur Verfügung. Die Abb. 3.11 verdeutlicht, dass vShark speziell auf Maschinen mit mehreren Prozessoren pro Knoten, oder wie in dem Fall von reinen Shared-memory-Rechnern, besser als die MPI-Bibliotheken abschneidet.



**Abb. 3.10:** Vergleich der Laufzeit von tpMM mit vShark und direkt mit MPI. Links: vShark mit 2 Threads auf 4 Knoten (8 virtuelle Prozessoren) und 8 MPI-Prozessen. Rechts: vShark mit 2 Threads auf 8 Knoten und 16 MPI-Prozessen. System: XEON-SCI-Cluster, MPICH.



**Abb. 3.11:** Leistungsvergleich von tpMM in MFLOPS/Prozessor für verschiedene Kommunikationsbibliotheken (MPICH-P4, MPICH-VMI, vShark).

## **3.7 Zusammenfassung und Fazit**

In diesem Kapitel wurde die C++-Bibliothek vShark vorgestellt, die eine hybride Implementierung (Message-Passing und Threads) zur schnellen Datenkommunikation auf einem SMP-System anbietet. Trotz eines Programmiermodells, welches einen verteilten Speicher voraussetzt, kann Intra-Knotenkommunikation zwischen leichtgewichtigen Threads schnell ausgeführt werden, ohne externe Funktionsaufrufe und teure Betriebssystemroutinen nutzen zu müssen. Die experimentellen Ergebnisse haben gezeigt, dass vShark eine deutliche Leistungssteigerung bei Intra-Knotenkommunikation erzielt. Ein Hauptvorteil von vShark ist das objektorientierte Design und die Realisierung oberhalb von Message-Passing-Bibliotheken. Deshalb ist es leicht möglich vShark mit vielen verschiedenen MPI-Implementierungen zu benutzen. Da vShark schon einen Adapter für den MPI-Standard 1.1 besitzt, kann es mit jeder konformen Bibliothek verwendet werden.

## Kapitel 4

# TGrid: Laufzeitumgebung für Multiprozessor-Tasks und Middleware für heterogene verteilte Systeme

*A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*

---

LESLIE LAMPORT

### 4.1 Einführung Grids

Die dichte Vernetzung von rechenstarken und preiswerten Computersystemen eröffnet die Möglichkeit, die zur Verfügung stehende Rechenleistung gebündelt auszunutzen, um berechnungsintensive Applikationen ohne teure dedizierte Supercomputer auszuführen. Um diese verteilt lagernde Rechenleistung nutzen zu können, bedarf es einer Infrastruktur, dem so genannten Grid [61]. Zwei Pioniere des Grid-Computing – Ian Foster und Carl Kesselman – haben das Grid im Jahr 1998 wie folgt definiert:

*„A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.“*

Dieser Definition liegt vor allem der Wunsch nach einer Orchestrierung der Ressourcen zu Grunde. Grid-Infrastrukturen, wie sie damals angesprochen wurden, haben in den letzten Jahren den Status von Testprojekten hinter sich gelassen. Heutzutage werden Computer-Grids im großen Maßstab für astronomische, biologische, geologische und medizinische Simulationen eingesetzt. Sie bilden das Rückgrat für berechnungsintensive Anwendungen in der Krebsforschung oder bei Wettersimulationen.

Trotzdem ist es noch ein langer Weg, um möglichst vielen Forschern Zugang zu rechenstarken Grids zu ermöglichen. Speziell die Nutzbarkeit der Computer-Grids als Serviceleistung ist eine wichtige Fragestellung, die Ian Foster wie folgt in [43] zusammenfasste:

*„In a future in which computing, storage, and software are no longer objects that we possess, but utilities to which we subscribe, the most successful scientific communities are*

*likely to be those that succeed in assembling and making effective use of appropriate Grid infrastructures and thus accelerating the development and adoption of new problem solving methods within their discipline.*“

Die starke Heterogenität der einzelnen Ressourcen verstärkt die Problematik der gemeinsamen Nutzung. Ein Hauptziel für die Entwicklung von Rechengrids ist damit die koordinierte, flexible und sichere Aufteilung der dynamisch bereitgestellten Ressourcen unter einer sich kontinuierlich ändernden Anzahl von Nutzern und Institutionen. Es wäre damit auch möglich, virtuelle und skalierbare Organisationen zu schaffen, die sich geographisch verteilte Ressourcen teilen.

Die Bereitstellung der Grid-Infrastruktur stellt aber auch die Frage, wie Anwendungen zu entwickeln und zu programmieren sind, um diese Infrastruktur gewinnbringend auszunutzen. Die ohnehin schon erhöhte Komplexität der Entwicklung von parallelen Anwendungen für MPPs oder Cluster wird im Grid weiter verstärkt. Es gilt deshalb, neue Programmiermodelle und Frameworks zu entwickeln, um Anwendungen auch im Grid lauffähig zu machen (*grid-enable*, *gridify*).

Die in den vorangegangenen Kapiteln vorgestellten gemischt-parallelen Applikationen weisen sehr gute Eigenschaften auf, um sie effizient im Grid ausführen zu können. Es wäre zum Beispiel denkbar, dass die einzelnen M-Tasks eines solchen taskbasierten Programms auf unterschiedlichen Service-Providern im Grid ausgeführt werden. Es ist deshalb das Ziel dieses Kapitels, eine Grid-Middleware und ein Laufzeitsystem zu realisieren, mit deren Hilfe M-Task-Programme verteilt im Grid abgearbeitet werden können.

## 4.2 Motivation und Entwicklungsziel

Viele größere Applikationen, welche Hochleistungsrechner zur Ausführung benötigen, besitzen eine modulare Struktur von kooperierenden Tasks. Beispielanwendungen sind Modellierungen der Atmosphäre, der Wasser- und Erdoberfläche, Flugzeugsimulationen oder Crashtests. Die Realisierung solcher Applikationen als gemischt-parallele Programme, die Ausnutzung von task- und datenparallelen Anteilen, hat sich für viele wissenschaftliche Applikationen als vorteilhaft erwiesen [101, 25]. Speziell konnte eine Verbesserung der Laufzeit bei mathematischen Lösungsverfahren durch die Verwendung von hierarchisch aufgebauten M-Tasks für Systeme mit verteiltem Speicher festgestellt werden [89]. Die Programme profitieren dabei oft von der Reduzierung des Kommunikationsoverheads durch die gruppenbasierte Ausführung von Kommunikationsoperationen, speziell bei kollektiven Operationen wie Broadcast oder Gather. Hierarchische M-Task-Programme entstehen, wenn die Applikation in unterschiedliche Teile (M-Tasks) zerlegt wird und diese dann parallel oder bei entsprechenden Abhängigkeiten sequenziell ausgeführt werden. Jede M-Task kann dabei wieder aus mehreren M-Tasks bestehen, wodurch eine Hierarchie von M-Tasks gebildet wird. Die M-Task-Programme lassen sich unter Zuhilfenahme der Daten- und Vorrangabhängigkeiten als gerichteter azyklischer Graph (DAG) darstellen. In Kapitel 2 dieser Arbeit wurde gezeigt, wie M-Task-Programme auf homogenen Rechnern mit verteiltem Speicher, insbesondere auf Clustersystemen, zu guten und teilweise sogar besseren Laufzeiten führen als rein datenparallele Programme.

In den letzten Jahren wurden viele große Anwendungen umgeschrieben, um Grid-Services nutzen zu können. Demzufolge stellt sich auch die Frage, ob und wie modular aufgebaute M-Task-Programme im Grid ausgeführt werden können. Daraus ergaben sich mehrere Zielstellungen, die im Rahmen dieser Arbeit verfolgt wurden. Ein zentraler Schwerpunkt liegt in der Realisierung einer grundlegenden Infrastruktur, die es ermöglicht, gemischt-parallele Programme im Grid auszuführen. Weitere Fragen betreffen u. a. die effiziente Regelung des Datentransfers zwischen den M-Tasks und das Scheduling der ausführbaren Tasks.

Aus den genannten Zielen ergeben sich für das hier vorgestellte Grid-Software-System namens TGrid folgende Anforderungen:

- Realisierung einer *Laufzeitumgebung für M-Task-Programme* im Grid. Dazu ist eine API zu entwickeln, um die Tasks und die Abhängigkeiten geeignet zu implementieren.
- Die Grid-Middleware soll hauptsächlich auf Multiclustern (*Cluster aus Clustern*) zu Einsatz kommen. Da jede Grid-Site eine unterschiedliche Architektur aufweisen kann, ist auch die Plattformunabhängigkeit der Grid-Programme eine wichtige Fragestellung.
- Das System sollte DAGs aus M-Tasks effizient abarbeiten. TGrid sollte Tasks, die mehr als einen Prozessor zur Ausführung benötigen, nur auf homogenen Untergruppen des gesamten heterogenen Grids ausführen. Beim Zuweisen der M-Tasks an bestimmte Prozessorgruppen sollte eine vorhandene *Kommunikationsinfrastruktur* (MPI-Bibliotheken, Infiniband-Treiber) *berücksichtigt* werden.
- Das Laufzeitsystem soll speziell die *Co-Allokation von Ressourcen* [30, 41] während der Abarbeitung eines Programms im Rechengrid erlauben. Mittels *Co-Allokation* von Ressourcen ist es möglich, Tasks (oder M-Tasks) eines einzigen Programms bei verschiedenen Service-Providern (Cluster, Supercomputer) auszuführen.
- Es soll ein *Kommunikationsmodul* entwickelt werden, welches die Datenübertragung zwischen den Prozessoren im System transparent gestaltet, so dass der einzelne Prozessor nicht wissen muss, in welcher Grid-Site sich der Empfänger befindet.
- Das Kommunikationsmodul soll flexibel gestaltet werden, so dass *unterschiedliche Protokolle zum Datenaustausch* eingesetzt werden können. Protokolle, die je nach Firewall-Situation eingesetzt werden können, sind z. B. TCP, HTTP oder SSH.
- Das Kommunikationsmodul soll des Weiteren die Fähigkeit haben, *Datenumverteilungen* automatisch vorzunehmen (Umverteilungskomponente / MxN). Damit würde sich die Komplexität bei der Programmierung von Grid-Programmen reduzieren.

## 4.3 Überblick über verwandte Arbeiten

Bei der Umsetzung der oben genannten Ziele floss die Arbeit [88] von Rauber und Rüniger in das Design des TGrid-Systems ein, in welcher ein Konzept zur Ausführung von M-Tasks in heterogenen Umgebungen vorgeschlagen wird. Des Weiteren wurden in [90]

Strategien zum Scheduling von M-Tasks untersucht, die den Kommunikationsoverhead zwischen zwei datenabhängigen Tasks bei der Abarbeitung verringern. Unter Berücksichtigung dieser Ergebnisse und der definierten Anforderungen aus dem letzten Abschnitt wurden verschiedene Grid-Systeme auf ihre Verwendbarkeit hin überprüft.

Die Globus-Alliance schuf mit dem Globus-Toolkit<sup>1</sup> eines der ersten frei verfügbaren Grid-Systeme [44, 45]. Das Globus-Toolkit ist eine Open-Source-Software, die es ermöglicht, Rechenleistung und Datenbanken über Instituts- oder Firmengrenzen hinweg gemeinsam zu nutzen. Das Toolkit enthält Module zur Überwachung, Sicherung und zum Management von Grid-Applikationen. Da das Toolkit sehr viele Aspekte des Grid-Computing abdeckt und sich an abstrakte Schnittstellen halten muss, ist es als Middleware für TGrid zwar durchaus eine Alternative, hat aber ein sehr lange Liste von Abhängigkeiten und ist aufwendig zu installieren, um damit in kurzer Zeit zu experimentieren. Eine Möglichkeit der Programmierung mit dem Globus-Toolkit wäre die Verwendung von MPI-Bibliotheken, die auf dem Toolkit aufsetzen, z. B. MPICH-G2<sup>2</sup> [60]. Die Bibliothek MPICH-G2 ermöglicht das Programmieren und Ausführen von Programmen in verteilten Systemen, wobei die einzelnen Maschinen von unterschiedlicher Architektur sein können. Sie konvertiert Daten zwischen den einzelnen Formaten und Architekturen (Endian) automatisch und unterstützt verschiedene Protokolle zur Kommunikation zwischen den teilnehmenden Rechnern. Für eine schnelle Kommunikation wird die beste Schnittstelle ausgewählt, d. h. TCP für die Kommunikation im WAN (Wide Area Network) oder die von Herstellern gestellten MPI-Bibliotheken auf einem Cluster oder MPP. Als Middleware-Kandidat für TGrid wäre MPICH-G2 durchaus geeignet, hat aber ein paar Nachteile, weswegen TGrid vorerst nicht darauf zurückgreift. Zum einen bietet MPI ein möglichst transparentes API, was die Programmierung erleichtert, aber die Abbildung von M-Tasks auf räumlich zusammenhängende Prozessoren erschwert. Das liegt daran, dass MPI nicht über portable Schnittstellen verfügt, mit denen Informationen über die Rechnersysteme der teilnehmenden Prozessoren ausgelesen werden können. Andererseits kommt wieder die Komplexität des Globus Toolkit zum Tragen. Dieses muss erst aufwendig installiert und administriert werden, um Tests mit MPICH-G2 zu ermöglichen.

Eine komplett in Java geschriebene Grid-Middleware bietet Ibis<sup>3</sup> [111]. Das Ziel des Ibis-Projekts besteht darin, eine effiziente und Java-basierte Plattform für das Grid bereitzustellen. Das Kernstück von Ibis ist die IPL (Ibis Portability Layer), die über eine allgemeine Schnittstelle zum Monitoring (z. B. NWS), zum Ressourcen-Management (z. B. GRAM) oder verschiedene Netzwerkprotokolle (TCP, UDP, MPI, GM) verfügt. Aufbauend auf der IPL wurden unterschiedliche Frameworks zum parallelen Programmieren entwickelt. Satin [110] ist die prominenteste dieser Umgebungen und unterstützt die Programmierung von Applikationen, die eine *Divide-and-Conquer*-Strategie zur Lösung einsetzen. Bei dieser Art von Programmen werden Probleme durch Aufteilung und Lösung von Teilproblemen effizient ausgeführt. Satin bietet neben der einfachen Programmierschnittstelle auch Algorithmen zum Task-Scheduling und implementiert *Stealing*-Strategien zur verbesserten

---

<sup>1</sup><http://www.globus.org/>

<sup>2</sup><http://www3.niu.edu/mpi/>

<sup>3</sup><http://projects.gforge.cs.vu.nl/ibis/>

Verteilung der Arbeitslast im Grid. Ibis verfügt über viele Funktionen, die es als Middleware für TGrid interessant machen, wie z. B. die bereitgestellten Kommunikationsprotokolle und die Möglichkeit zur Überbrückung von Firewalls. Allerdings abstrahiert die IPL den Prozess vom Ort der Maschine, d. h. es ist schwer herauszufinden, welche Prozesse auf welchen Rechnern liegen. Durch die IPL lassen sich auch nur schwer die von Herstellern mitgelieferten Kommunikationsbibliotheken der benutzten Parallelrechner verwenden.

Das System OurGrid<sup>4</sup> [4] implementiert ein freies Peer-to-Peer-Grid (P2P), an dem jeder Nutzer mit einem Internetzugang und einem Linux-Rechner teilnehmen kann. Dadurch kann auf eine immense Rechenleistung zugegriffen werden. Da es sich aber um ein sehr dynamisches P2P-Netz handelt, kann nur eine bestimmte Art von parallelen Applikationen effizient berechnet werden. Prädestiniert für ein sich dynamisch änderndes Netz sind *Parameter Sweep*-Applikationen (auch Bag-of-Tasks-Applikationen (BoT) genannt). Dabei wird dasselbe Programm mehrfach mit unterschiedlichen Eingabedaten (Parametern) gestartet. Diese Art von Problem lässt sich sehr gut parallelisieren, da die einzelnen Aufrufe keine Abhängigkeiten untereinander aufweisen. Deshalb bedarf es auch keiner komplexen Funktionen, die z. B. Task-Migration oder Checkpointing bereitstellen.

Ein weiteres Grid-System zur parallelen Abarbeitung von vielen Tasks ist Condor<sup>5</sup> [106]. Wie OurGrid hat auch Condor das Ziel, viele verfügbare Arbeitsplatzrechner zu einem leistungsstarken Supercomputer zusammenzufügen. Die Condor-Entwickler sprechen selbst von einem System für High Throughput Computing (HTC), was im Grunde BoT-Applikationen entspricht. Die Zielapplikationen sind Programme, die eines extrem hohen Rechenaufwands zur Lösung bedürfen. Der allgemeine Ablauf im Condor-System ist wie folgt: Zuerst übergibt der Nutzer einen Job an Condor. Das System versucht dann via Ressourcen-*Matchmaking* eine passende Workstation (Betriebssystem, Architektur) zu finden und führt den Job dort aus. Nach Ende des Jobs wird der Nutzer über die Terminierung informiert. Condor realisiert Funktionen zum Checkpointing und zur Migration von Jobs. Mit Condor können auch Abhängigkeiten zwischen Tasks mit Hilfe von DAGMan (Directed Acyclic Graph Manager) modelliert werden. DAGMan ist ein Meta-Scheduler für Condor und regelt die Abhängigkeiten zwischen Jobs auf einer höheren Ebene als die lokalen Condor-Scheduler für jedes angeschlossene System. Die Abarbeitung der Jobs erfolgt dann mit Condor, wobei Daten über Ein- und Ausgabedateien zwischen den abhängigen Tasks übergeben werden. Condor bietet auch eine Schnittstelle zum Globus Toolkit (Condor-G), um Jobs auf Globus-Rechnern ausführen zu können. Als Middleware für M-Task-Programme kommt Condor aber nicht in Frage, da es keinen direkten Datenaustausch zwischen Mehrprozessor-Jobs erlaubt. DAGMan ist außerdem nicht leistungsfähig genug, um komplexe Taskgraphen (z. B. mit einer dynamischen Struktur) zu beschreiben.

Eine verteilte Umgebung zur parallelen Lösung von komplexen wissenschaftlichen Problemen wird durch NetSolve [5, 96] (jetzt GridSolve<sup>6</sup>) realisiert. Das System erlaubt, dass Nutzer auf im Netzwerk verfügbare Ressourcen zugreifen können. Eine Ressource entspricht dabei einer bereitgestellten Funktion auf einem ausgezeichneten Rechner. NetSolve

---

<sup>4</sup><http://www.ourgrid.org/>

<sup>5</sup><http://www.cs.wisc.edu/condor/>

<sup>6</sup><http://icl.cs.utk.edu/netsolve/>

überwacht das Netzwerk von Rechnern, die Berechnungsfunktionen zur Verfügung stellen, und wählt bei einer Nutzer-Anfrage die bestmögliche aus. Um NetSolve im Grid verfügbar zu machen, wird GridRPC [95] verwendet. GridRPC ist eine standardisierte und portable Programmierschnittstelle, um Grid-Computing zu vereinfachen. Im GridRPC-Modell ruft ein Client eine Funktion beim Service-Provider mittels Remote Procedure Call (RPC) auf. Der Server führt die Funktion mit den übergebenen Parametern aus und liefert das Ergebnis an den Client zurück. Im Fall von NetSolve bieten die Server Funktionen zur numerischen Berechnung mathematischer Probleme an, z. B. DGSEV zur Lösung eines Gleichungssystems. Die Ursprungsimplementierung von GridRPC hatte den Nachteil, dass Daten immer zum aufrufenden Client zurück transportiert wurden. Wird aber eine weitere Funktion auf dem gleichen Server aufgerufen, welche die Daten der vorangegangenen Funktion benötigt, führt das zu einem großen Kommunikationsoverhead. Für dieses Problem wurde eine Lösung implementiert [34], die Daten auf dem Server solange vorhält, bis sie von der abhängigen Task gebraucht werden. Damit werden die Daten direkt zu dem Server transferiert, der die Daten als Eingabe benötigt, und dadurch wird der zusätzliche Kommunikationsschritt über den aufrufenden Client vermieden. Der Hauptvorteil der GridRPC-Architektur liegt in der potenziellen Leistung der angebotenen Funktionen. Mathematische Funktionen aus ScaLAPACK [14] können auf der Zielplattform an die Caches [112] angepasst werden. In der Bereitstellung und Optimierung der einzelnen Funktionen auf den Servern liegt auch der Nachteil von GridRPC für die Verwendung innerhalb von TGrid. Mit diesem Modell lassen sich nur schlecht beliebige Programme im Grid ausführen, da alle Funktionen erst auf einem Server bereitgestellt werden müssen.

Eine relativ neue Grid-Middleware wird von ProActive<sup>7</sup> [55] bereitgestellt. Das Open-Source-Projekt ProActive bietet eine Java-Bibliothek, die das Programmieren von parallelen und verteilten Multi-Thread-Anwendungen ermöglicht. Das Einsatzspektrum reicht von Local Area Networks, über Cluster aus Workstations, P2P-Desktop-Grids bis hin zu Internet-Grids. ProActive besitzt eine Architektur, die es erlaubt, mit unterschiedlichen De-facto-Standards der Datenkommunikation zu interagieren, z. B. Schnittstellen für Web Services, HTTP, RMI, SSH, SCP, Globus GT3, Globus GT4. ProActive bietet eine einfache Programmierumgebung durch eine Integration in Eclipse. Außerdem existiert die Möglichkeit MPI-Programme in die ProActive-API zu portieren. Die Bibliothek ProActive bietet eine sehr umfangreiche API um Grid-Anwendungen zu erstellen. Für die Abarbeitung von M-Taskgraphen ist auch hier die Abstraktion von der Ausführungsplattform ein Nachteil.

Die Diet-Bibliothek<sup>8</sup> [20] ist eine, im Vergleich zu den anderen Systemen, recht leichtgewichtige Grid-Middleware. Die Software verwendet einen Client-Server-Ansatz, um Grid-Applikationen verteilt auszuführen. Über die Diet-Clients kann ein Nutzer eine Task an das System übergeben. Ein Master-Agent dient als Scheduling-Komponente und platziert die Task auf einem Zielsystem. Der entsprechende Server bearbeitet die Task und benachrichtigt den Client über den Status. Die Bibliothek verwendet CORBA, um Daten zwischen verschiedenen Architekturen und Betriebssystemen auszutauschen. Die Verwendung von CORBA ist aber für eine Kommunikation im WAN problematisch, da Firewallprobleme

---

<sup>7</sup><http://proactive.inria.fr/>

<sup>8</sup><http://graal.ens-lyon.fr/DIET/>

auftreten können. Außerdem bietet Diet nur eine C++-Schnittstelle, was die Portabilität von möglichen TGrid-Programmen einschränken würde.

Erst in den letzten Jahren wurden verstärkt Systeme zum Abarbeiten von wissenschaftlichen Workflows entwickelt, so dass sie zu Beginn des TGrid-Projektes nicht als Middleware in Frage kommen konnten. Trotzdem verfolgen diese Ansätze ähnliche Ziele wie TGrid, denn die wissenschaftlichen Workflows werden als DAGs beschrieben und sollen im Grid abgearbeitet werden. Die meisten dieser System sind für die Ausführung von biotechnologischen und physikalisch-chemischen Verfahren entwickelt worden. Viele dieser Anwendungen lassen sich gut parallelisieren, wie z. B. das Alignment von Proteinsequenzen. Eine Software zum Design und Ausführen dieser wissenschaftlichen Workflows ist Taverna<sup>9</sup> [56]. Taverna ist Teil des MyGrid-Projekts<sup>10</sup> und benutzt die Sprache Scufl (Simple Conceptual Unified Flow language), um Workflows zu beschreiben. Taverna ist sehr stark mit der Abarbeitung von Workflows im Bereich der Bioinformatik und Biotechnologie verzahnt. Eine weitere Umgebung zur Abarbeitung von Workflows auf dem Grid ist Pegasus<sup>11</sup> [32]. Dabei handelt es sich um ein System, mit dem Workflows entworfen und im Grid ausgeführt werden können. Des Weiteren besitzt Pegasus eine Scheduling-Komponente. Als Eingabe verwendet Pegasus einen abstrakten Workflow, der die Daten und Transformationen auf einer logischen Ebene beschreibt. Als Workflow-Ausführungsplattform kommt Condors DAGMan zum Einsatz, wodurch der Datenaustausch zwischen den Tasks über Dateien realisiert wird.

## 4.4 Laufzeitumgebung für M-Task-Programme

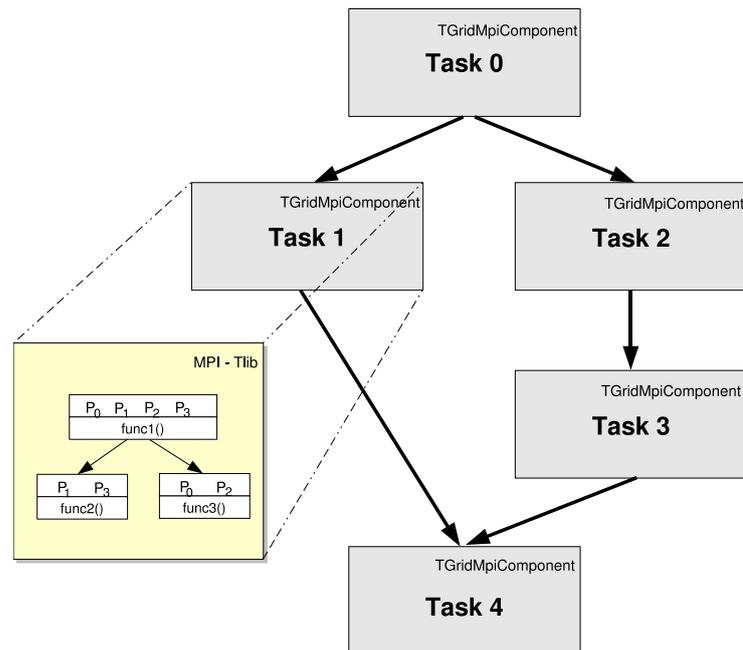
In den folgenden Abschnitten wird unter Berücksichtigung der Implementierungsziele und der verwandten Arbeiten der Entwurf des Grid-Software-Systems TGrid und dessen Funktionsweise beschrieben. Das Ziel von TGrid ist die Bereitstellung einer Infrastruktur, um Graphen von M-Tasks auf heterogenen Systemen abzuarbeiten. Ein Hauptaugenmerk liegt dabei auf der Ausnutzung der möglicherweise vorhandenen Kommunikationsbibliotheken, wie z. B. herstellerspezifischen MPI-Implementierungen.

Für eine genauere Betrachtung der Bestandteile von TGrid werden zuerst die Zielprogramme sowie die Zielplattformen genauer umrissen. Ein typisches M-Task-Programm besteht aus einer Reihe von Tasks, die parallel ausgeführt werden können. Jede M-Task soll dabei eine effizient implementierte Komponente darstellen. Auf einem „normalen“ Supercomputer (Cluster, MPP) werden solche Tasks üblicherweise mit MPI oder PVM realisiert. Damit eine hohe parallele Effizienz bei der Bearbeitung des Taskgraphen im Grid erzielt werden kann, ist es von entscheidender Bedeutung, dass die einzelnen Tasks auf rechner-spezifische Bibliotheken zurückgreifen können. Die bestehenden MPI-Programme sollen nach Möglichkeit relativ einfach für die Benutzung im Grid aufbereitet werden können. Es ist deshalb bei der Entwicklung von TGrid wichtig, die Integrationsmöglichkeiten von MPI-Programmen in die Betrachtungen mit einzubeziehen. Die Abb. 4.1 stellt

<sup>9</sup><http://taverna.sourceforge.net/>

<sup>10</sup><http://www.mygrid.org.uk/>

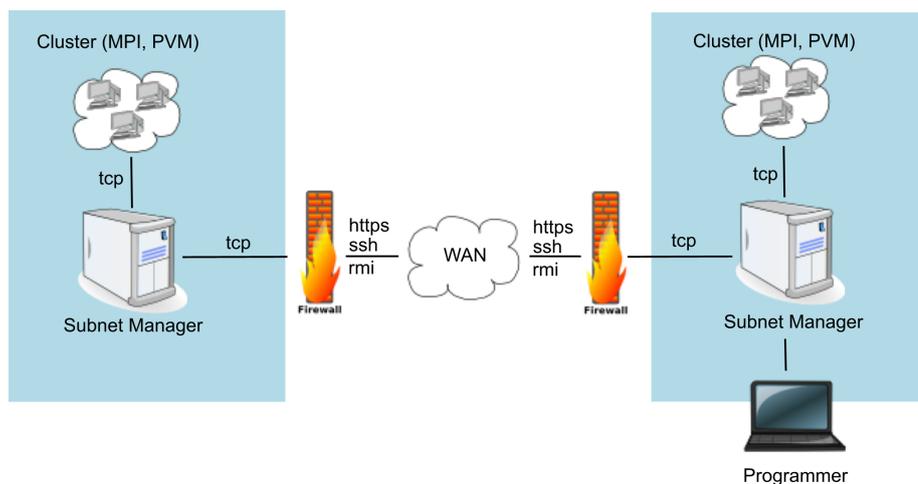
<sup>11</sup><http://pegasus.isi.edu/>



**Abb. 4.1:** Exemplarische Struktur eines TGrid-Programms. Einzelne TGrid-Komponenten bilden einen Graph, wobei jede Komponente (M-Task) mehr als einen Prozessor zur Ausführung verwenden kann.

exemplarisch einen M-Taskgraphen dar, welcher mittels TGrid auf einer heterogenen Ansammlung von Rechnern ausgeführt wird. Die Knoten des Graphen bestehen aus den M-Tasks, die im Weiteren auch als Komponenten bezeichnet werden. Der Begriff „Komponente“ wird deshalb gewählt, da sich die TGrid-Tasks ähnlich zu Komponenten im Software-Engineering verhalten sollen. Die Komponenten von TGrid sollen nach außen definierte und standardisierte Schnittstellen besitzen und ihre interne Implementierung vor den Benutzern weitestgehend verstecken.

Als Zielplattform für die Ausführung solcher Taskgraph-basierten Programme werden vor allem Multicluster (Cluster aus Clustern) betrachtet. Viele wissenschaftliche Institute haben sich in den letzten Jahren kleinere Clustersysteme (oft Beowulf-Cluster, 16–128 Prozessoren) angeschafft, um rechenintensive Probleme in den Arbeitsgruppen lösen zu können. Oft sind die einzelnen Cluster mit schnellen Verbindungsnetzwerken wie Myrinet, Infiniband oder SCI ausgestattet. Zur effizienten Ausführung einer M-Task ist es deshalb sinnvoll, diese Infrastruktur auszunutzen und die Abarbeitung einer Task auf einen einzelnen Cluster zu limitieren. Aus diesem Grund wird in TGrid eine M-Task nicht über Cluster Grenzen hinaus verteilt, d. h. zwei unterschiedliche Tasks können auf unterschiedlichen Clustern ausgeführt werden, aber einer Task können nur Prozessoren eines einzelnen Clusters zugewiesen werden. Auch andere Forscher, z. B. aus dem Umfeld des M-Task-Scheduling, verwenden dieses Ausführungsmodell für Multicluster [23]. Eine denkbare Konfiguration einer TGrid-Umgebung ist in Abb. 4.2 dargestellt. In dem Beispiel sollen zwei Cluster-Systeme, die beide MPI und PVM anbieten, über ein WAN verbunden werden, um eine gemeinsame Plattform zur Ausführung von gemischt-parallelen Programmen



**Abb. 4.2:** TGrid-Beispielkonfiguration.

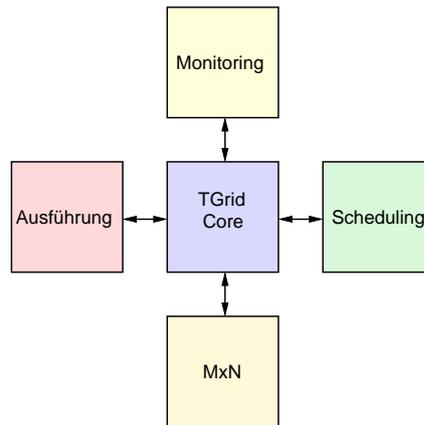
anzubieten. Der Programmierer oder Benutzer hat Zugriff auf eines dieser beiden Teilnetze (subnets). Jedes Teilnetz wird von einem Subnet-Manager verwaltet. Die gesamte Anzahl dieser Subnet-Manager bildet die Ausführungsplattform von TGrid. Nachdem der Nutzer das taskbasierte Programm an einen Subnet-Manager übergeben hat, kann das Programm parallel auf allen angeschlossenen Teilnetzen bearbeitet werden. In der Grafik wird die Problematik des Verbindungsaufbaus deutlich, die auch das Hauptproblem dieser Vorgehensweise darstellt. Die Clustersysteme sind im Normalfall durch eine Firewall abgesichert. Darüber hinaus verwenden die Cluster zur Intranet-Kommunikation nur interne private IP-Adressen, wodurch sich die einzelnen Prozessoren im Grid nicht „sehen“ können. Die von TGrid zu lösende Aufgabe ist deshalb, Teilnetze durch geeignete Protokolle (http, ssh) miteinander zu verbinden und für die Zustellung von Nachrichten zwischen Prozessoren verschiedener Teilnetze zu sorgen.

Nach Beschreibung der Zielapplikationen und -plattformen wird im folgenden Abschnitt die Software-Architektur von TGrid beschrieben.

## 4.5 Definition der TGrid-Architektur

Im Rahmen dieser Arbeit wurde eine grundlegende Software-Architektur TGrid entwickelt, die eine Abarbeitung von M-Taskgraphen in einer heterogenen Umgebung ermöglicht. Die Basisarchitektur von TGrid, dargestellt in Abb. 4.3, besteht aus folgenden Software-Komponenten:

- dem Systemkern (core),
- dem Ausführungsmodul,
- dem Monitoring-Modul,
- dem Scheduling-Modul und
- dem Umverteilungsmodul (MxN).



**Abb. 4.3:** Kernmodule von TGrid.

Der *Systemkern* enthält alle notwendigen Funktionen zum Aufbau und zum Überwachen der verbundenen Rechner. Er stellt außerdem Routinen zum Nachrichtenaustausch zwischen den beteiligten Maschinen zur Verfügung. Das *Ausführungsmodul* wird genutzt, um die M-Tasks eines TGrid-Programms auf einem Teilnetz auszuführen. Informationen über die aktuell in Ausführung befindlichen Tasks werden in TGrid durch das *Monitoring-Modul* bereitgestellt. Während der Abarbeitung eines M-Task-Programms werden Tasks dann ausführbar, wenn alle Abhängigkeiten dieser Tasks aufgelöst wurden. Die Entscheidung, wie viele Prozessoren von welchem Teilnetz einer Task zugewiesen werden, trägt das *Scheduling-Modul*. Besteht zwischen zwei aufeinanderfolgenden Tasks eine Datenabhängigkeit, werden die Ausgabedaten der Elterntask als Eingabedaten für die Kindtask bereitgestellt. Diese Aufgabe des Datentransfers und ggf. einer Datenumverteilung übernimmt das *Umverteilungsmodul*.

**Ausführen von M-Task-Programmen** Für die Ausführung eines M-Task-Programms sind alle vorgestellten Module notwendig. Das genaue Zusammenspiel der einzelnen Entitäten wird anhand der Abb. 4.4 erläutert. Der Entwickler oder Benutzer von TGrid hat Zugriff auf einen lokalen Subnet-Manager, an den er sein M-Task-Programm übergibt. Jedes M-Task-Programm ist, wie vorher erklärt, aus verschiedenen M-Tasks (Komponenten) aufgebaut, wobei sich jede Komponente auf einem Teilnetz (Subnet) ausführen lässt. Ein Subnet hat immer genau einen dedizierten Subnet-Manager, welcher die Belegung der Prozessoren in diesem Subnet überwacht. Typischerweise ist ein Subnet mit einem verfügbaren Cluster-System gleichzusetzen. Ein Subnet kann aber auch nur einen einzelnen Rechner enthalten, was zu einer Art Desktop-Grid führen würde. Nachdem das Programm an den Subnet-Manager übertragen wurde, startet dieser einen TaskGraph-Walker. Dabei handelt es sich um ein Objekt, welches den aktuellen Status und den Abarbeitungsfortschritt des Taskgraphen enthält. Man könnte es als eine abstrakte Form eines Programmzählers für Grid-Applikationen betrachten. Dieser TaskGraph-Walker extrahiert die ausführbaren Tasks des Graphen und übergibt diese an den Scheduler, hier als *Component Scheduler* bezeichnet. Der Scheduler kann aktuelle Informationen zur Lastverteilung im gesamten

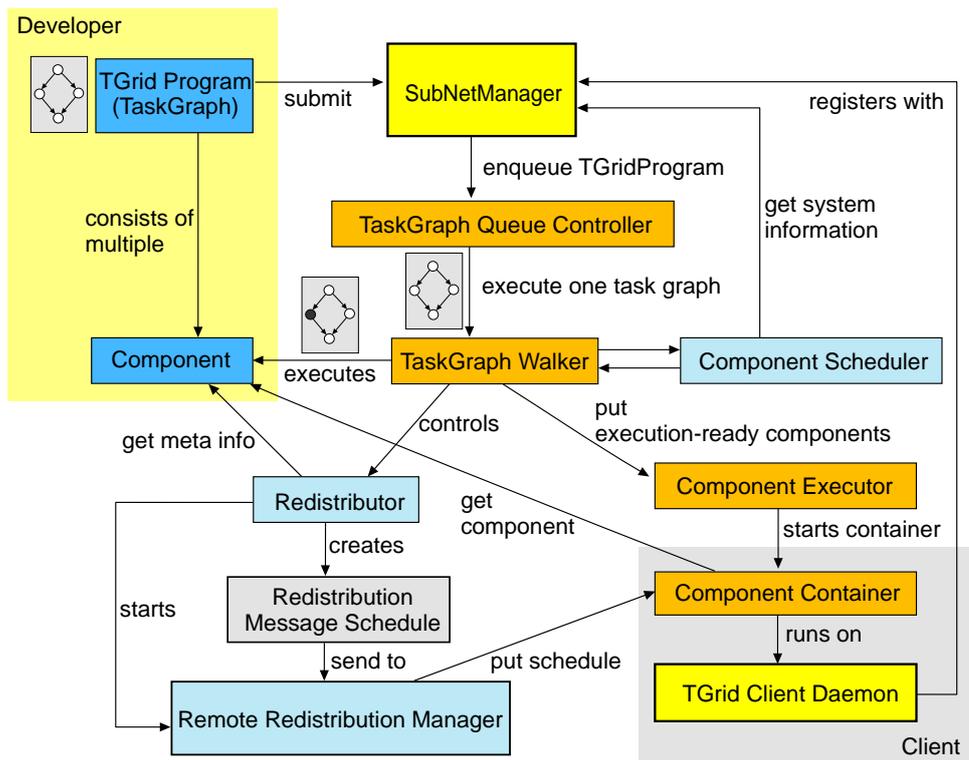


Abb. 4.4: Abarbeitung eines TGrid-Programms.

Grid über den Subnet-Manager beziehen. Damit lässt sich unter Zuhilfenahme von speziellen Algorithmen (siehe Kapitel 5) ein Ablaufplan (*schedule*) für die ausführbaren Tasks erstellen. Der Schedule spezifiziert für jede Task, auf welche Subnets und Prozessoren sie abgebildet wird. Der TaskGraph-Walker startet auf den Subnets einen Task-Container (*Component-Container*), in den die M-Tasks abgebildet werden sollen. Dieser Container dient dazu, über standardisierte Schnittstellen Informationen von der Task zu empfangen und Befehle an die Task weiterzuleiten. Der Component-Container empfängt die auszuführende Task vom TaskGraph-Walker und startet sie auf dem Client-System. Jeder einzelne Host eines Subnets, der an einer TGrid-Konfiguration teilnehmen soll, muss über einen laufenden Client-Daemon verfügen. Ähnlich zu MPI kann der Subnet-Manager mit Hilfe der Daemons die Verfügbarkeit von Hosts im eigenen Teilnetz prüfen. Wenn die Ausführung einer Task in einem Container auf der Seite der Clients beendet ist, informiert der Container den entsprechenden TaskGraph-Walker. Dieser validiert dann die Abhängigkeiten und übergibt ggf. neue ausführbar gewordene Tasks an den Scheduler. Sollte eine Datenabhängigkeit zu einer nachfolgenden Task bestehen, muss evtl. eine Umverteilung der Daten stattfinden. Dazu startet der TaskGraph-Walker mit dem *Redistributor* eine Entität, welche die Datenkommunikation bei der Umverteilung überwacht. Der Redistributor erzeugt mit Hilfe der Scheduling-Information über Quell- und Zieltask eine Liste von Nachrichten, die zwischen den Prozessoren, welche die Tasks ausführen, geschickt werden müssen. Die genaue Funktionsweise der komplexen Datenumverteilung in TGrid wird de-

tailliert in Abschnitt 4.8 vorgestellt. Sind die Umverteilungen der Daten zwischen zwei Komponenten abgeschlossen, kann die beendete Elterntask abgebaut werden. Dazu gehört auch die Beendigung des zugehörigen Komponentencontainers auf Client-Seite. Nachdem alle Tasks des Programms im System abgearbeitet wurden, wird der Nutzer über das Ende der Ausführung benachrichtigt. Dies geschieht über ein Software-Handle, welches er bei der Übergabe des Programms erhalten hat.

**Implementierung von TGrid mit Java und MPI** Die Middleware und das Laufzeitsystem von TGrid wurden in Java geschrieben. Die Entscheidung für Java als Programmiersprache hatte mehrere Gründe. Möchte man beliebigen Code auf einem Zielsystem ausführen, haben Programme die noch in Maschinencode zu übersetzen sind, z. B. in C oder C++ geschrieben, verschiedene Nachteile. Es ist äußerst schwierig, komplexe C-Programme mit abhängigen Bibliotheken zur Laufzeit zu übersetzen oder Abhängigkeiten aufzulösen. Das Senden und Ausführen einer einzigen Binärdatei ist darüber hinaus für unterschiedliche Prozessor-Architekturen, Betriebssysteme und Binärformate nicht möglich. Deshalb standen nur interpretierte Sprachen (Skriptsprachen wie Python oder Perl) oder Sprachen mit einer Laufzeitumgebung wie Java und C# (.NET) zur Wahl. Die Skriptsprachen erreichen ohne den Aufruf von maschinenabhängigen Modulen nicht die Geschwindigkeit, die Java und C# zu leisten im Stande sind. Andererseits ist eine Typisierung von Funktionen speziell beim Programmieren von komplexen nebenläufigen Anwendungen von großer Bedeutung, da Laufzeitfehler durch leicht erkennbare Fehler im Code ausgeschlossen werden können. Die Skriptsprachen bieten keine solche Typisierung. C# hat den Nachteil, dass es speziell für das Windows-Betriebssystem entwickelt wurde, obwohl mittlerweile auch freie Implementierungen von .NET für Linux existieren. Java ist speziell im Serverbereich sehr weit verbreitet und akzeptiert. Darüber hinaus gibt es eine Reihe exzellenter freier Java-Entwicklungsumgebungen für Linux.

Die aktuelle Implementierung von TGrid verwendet MPI, um M-Tasks auf verfügbaren Clients auszuführen und um innerhalb der M-Tasks zu kommunizieren. Das TGrid-System ist transparent implementiert, d. h. für die Verwendung von MPI innerhalb der M-Tasks müssen nur entsprechende Adapter implementiert werden. Der für MPI mitgelieferte Adapter für die Ausführung von M-Tasks bewirkt serverseitig, dass der TaskGraph-Walker mit MPI-Unterstützung gestartet wird. Auf der Client-Seite wird der mitgelieferte MPI-Container instantiiert und ausgeführt. Eine M-Task, die MPI als Kommunikationsschnittstelle verwendet, kann dann vom TaskGraph-Walker zum Client-Container kopiert und dort gestartet werden. Das TGrid-Programm, welches die M-Tasks orchestriert, ist hingegen völlig abstrakt gehalten und kennt die jeweilige interne Implementierung und die verwendete Kommunikationsbibliothek der M-Tasks nicht. Das Design von TGrid lässt es deshalb auch leicht zu, verschiedene Kommunikationsbibliotheken innerhalb von M-Tasks zu verwenden. Um den MPI-Container auf der Client-Seite zu starten, wird die Information aus dem Task-Ablaufplan (Schedule) in eine MPI-Maschinendatei (*machine file*) geschrieben. Diese Maschinendatei und eine eindeutige Nummer der auszuführenden Komponente (M-Task) wird dann zum Subnet-Manager des Zielsystems geschickt. Dieser startet mittels `mpirun` die MPI-Container als Prozesse auf allen angegebenen Client-Hosts.

Das Container-Programm registriert sich beim Subnet-Manager und fordert über die erhaltene Komponenten-ID die zu bearbeitende Task an. Der Subnet-Manager sendet die Komponente an den MPI-Container, welcher diese ausführt. Das Container-Programm ist wichtig, um Funktionen wie Datenumverteilung und Monitoring von Tasks bereitzustellen. Da die Funktionen von allen Tasks benötigt werden, stellen die Container diese modular bereit und sie müssen somit nicht von jeder Task neu implementiert werden.

Da die M-Tasks in Java geschrieben sind, ist eine Java-kompatible Schnittstelle von MPI für die Ausführung notwendig. In der prototypischen Realisierung von TGrid wird mpiJava (Version 1.2.5) als Implementierung der MPI-Funktionen in Java verwendet. mpiJava ist nicht komplett in Java geschrieben. Stattdessen werden die Funktionen einer nativen MPI-Bibliothek über das *Java Native Interface* verfügbar gemacht. Die mpiJava-Implementierung stellt sicher, dass die Daten aus den Java-Programmen in das von der jeweiligen Architektur verwendete Datenformat konvertiert werden. Als MPI-Bibliothek kam auf den verwendeten Clustern MPICH zum Einsatz. Damit konnten proprietäre Netzwerktreiber für Infiniband und SCI verwendet werden, was mit einer reinen Java-Implementierung zum Zeitpunkt der Entwicklung nicht möglich war. Die mpiJava-Bibliothek musste für die Verwendung für die x86\_64-Architektur angepasst werden, um eine korrekte Datentypkonvertierung zu gewährleisten. Mit MPJ<sup>12</sup> haben die ursprünglichen Entwickler von mpiJava auch eine reine Java-Implementierung von MPI realisiert, welche als zukünftige Basis für die M-Tasks in TGrid dienen könnte.

**Problematik der freien Ports** Theoretisch können Subnet-Manager und Clients auf demselben Rechner laufen. Dies ist speziell für die Entwicklung von TGrid-Programmen sinnvoll, da man auf einem einzigen Rechner ein komplettes Teilnetz simulieren kann. Außerdem können, wie auch bei MPI möglich, mehrere Clients auf einem Host laufen. Werden jetzt TGrid-Tasks auf dem Teilnetz ausgeführt, muss die TGrid-Laufzeitumgebung jede Task eindeutig ansprechen können, um den Datentransfer durchzuführen. Aus diesem Grund besitzt TGrid einen Portmanager, der ausgehend von einem für ein System definierten Startport, z. B. 3500, freie Ports an einen M-Tasks ausführenden Prozess vergibt. Ist eine M-Task abgearbeitet, werden die der Task zugeordneten Prozesse beendet und die verwendeten Ports wieder freigegeben.

**Schnittstellen der M-Tasks** Die M-Task ist der zentrale Baustein aller hierarchisch aufgebauten gemischt-parallelen Programme. Jede M-Task kann einem beliebigen Teilnetz von TGrid zugewiesen und dort ausgeführt werden. Deshalb muss es möglich sein, die Tasks über das Netzwerk zu verschicken und auf entfernten Rechnern auszuführen. Wie bereits mehrfach angesprochen, existieren zwischen den M-Tasks oft Kontroll- und Datenabhängigkeiten. Damit Abhängigkeiten zwischen Tasks definiert werden können und um sie zur Laufzeit auflösen zu können, muss jede Task in TGrid bestimmte Schnittstellen implementieren.

Als Beispiel soll die Datenumverteilung einer Matrix von einer Elterntask  $T_P$  auf eine Kindtask  $T_C$  betrachtet werden. In den meisten taskparallelen Programmen müsste der

---

<sup>12</sup><http://mpj-express.org>

Programmierer selbst für die Datenumverteilung und die Kommunikation zwischen Eltern- und Kindtasks sorgen. Dies ist aber für große taskbasierte Programme keine portable Lösung, da die Zielplattform und die Zuordnung der Tasks zu Teilnetzen variieren kann. Überdies ist es eine sehr zeitaufwendige Arbeit, eine funktionstüchtige Umverteilungsroutine für eine bestimmte Datenstruktur zu erstellen. Das TGrid-Framework verfolgt deswegen das Ziel, die Datenumverteilung für den Entwickler zu übernehmen. Das kann aber nur gelingen, wenn jede M-Task die entsprechenden Schnittstellen besitzt, um Daten aus einer Task zu lesen und in einer andere zu schreiben. Mit den definierten Abhängigkeiten zwischen den M-Tasks (Abbildung der Variablen zwischen Eltern- und Kindtasks) kann TGrid nun unterstützend arbeiten und die Daten von der Quelltask an die Zieltask übermitteln. Da das TGrid-Laufzeitsystem die zugewiesenen Prozessoren der Quell- und Zieltasks kennt, kann TGrid die benötigten Daten zwischen den Tasks austauschen.

Um diese Funktionalität innerhalb von TGrid zu gewährleisten, muss jede M-Task die folgenden Schnittstellen implementieren, wobei einige Funktionen exemplarisch beschrieben sind:

- Das *Scheduling Interface* dient dazu, Informationen über bevorzugte Parameter und Restriktionen der Task abzurufen. Solche Informationen sind z. B. die minimale und die maximale Anzahl von Prozessoren, die von der Komponente unterstützt werden.

```
public int getMinimumNumberOfProcs()
public int getMaximumNumberOfProcs()
```

- Das *Data Mapping Interface* ist speziell für die Datenumverteilung konzipiert worden. Diese Schnittstelle bietet Funktionen, um Informationen über die Datenverteilung innerhalb der Komponente und über das verwendete Prozessgitter zu erlangen.

```
public DataDistributionSpec getDistributionSpec(String varID, int procNum)
public TGProcLayout getProcLayout(int procNum)
public void setGridData(String varID, GridData data)
public GridData getGridData(String varID)
```

- Das *Property Interface* einer M-Task dient der Deklaration von Variablen. Mit diesen Funktionen lassen sich Variablenbezeichner der Task mit Datenstrukturen verbinden, z. B. kann der Variable mit dem symbolischen Namen *a* ein Vektor mit 10 Fließkommazahlen zugewiesen werden.

```
public void setDataDeclaration(final String varID, final TGridDataDecl decl)
public TGridDataDecl getDataDeclaration(final String varID)
```

- Ein *Execution Interface* dient der eigentlichen Ausführung der M-Task. Über die *execute*-Methode kann das Container-Programm die M-Task auf den Clients starten. In dieser Methode befindet sich das eigentliche (daten)parallele Programm, wobei die zuvor genannten Schnittstellen für die Ein- und Ausgabe der Daten benötigt werden.

```
public int execute(SubsystemHandle subsys) throws TGridRunException
```

## 4.6 Konfiguration von TGrid

Zum Ausführen von taskparallelen TGrid-Programmen müssen auf den miteinander verbundenen Clustern (Subnets) die entsprechenden TGrid-Prozesse laufen, damit die Laufzeitumgebung benutzt werden kann. Wie in Abb. 4.2 dargestellt, wird jedes einzelne Teilnetz (Rechnerwolke) von einem Subnet-Manager verwaltet. Auf jedem ausgezeichneten Rechner, welcher als Subnet-Manager von TGrid fungieren soll, wird deshalb auch ein softwareseitiger Subnet-Manager-Prozess gestartet. Jeder dieser Prozesse benötigt die Informationen, welche Client-Rechner er überwachen soll und welche anderen Teilnetze (Subnets) noch im Gesamtgrid verfügbar sein können. Diese Information kann jeder Subnet-Manager der TGrid-Konfigurationsdatei entnehmen, in der für jedes Teilnetz die Adressen der Manager und der Clients definiert sind. Die Konfigurationsdatei in Abb. 4.5 zeigt beispielhaft die Definition zweier Teilnetze. In der XML-Beschreibung der Grid-Umgebung wird jedem Teilnetz eine eindeutige Identifikationsnummer zugewiesen, damit Beziehungen zwischen den Teilnetzen konfiguriert werden können. Zu den Beziehungen gehören in erster Linie die Anzahl und die Art der Verbindungen zwischen den einzelnen Subnets. Die Verbindungen zwischen den Teilnetzen werden im Abschnitt `connections` definiert. In der Beispielkonfiguration werden hierfür Verbindungen von Subnet `ai2` zu allen anderen Subnets (\*) über den Router `router1` definiert. Im entsprechenden XML-Abschnitt `routers` ist für diesen Router definiert, dass er das Teilnetz `clusternet` über direkten Weg via TCP oder über einen SSH-Tunnel über Port 5000 erreicht. Wird ein Subnet-Manager gestartet, liest dieser die Konfigurationsdatei und versucht die anderen angegebenen Subnet-Manager über die entsprechenden Verbindungen zu erreichen.

Des Weiteren existiert mit `ports.properties` eine wichtige Konfigurationsdatei. In dieser Datei lassen sich die zu verwendenden Ports für Router, Subnet-Manager und Client-Daemon an die Zielplattformen anpassen. Außerdem wird in dieser Datei auch der Bereich der vom Portmanager zu vergebenden Ports definiert (siehe Abschnitt 4.5).

## 4.7 Programmierung von TGrid-Applikationen

Die Programmierung der gemischt-parallelen Applikationen mit TGrid erfolgt in zwei Schritten. Zuerst muss für jeden Knoten des Taskgraphen eine M-Task definiert und implementiert werden. Die Implementierung einer M-Task für TGrid umfasst neben der Programmierung des datenparallelen Codes auch die Bereitstellung der erforderlichen Schnittstellen, um die Task innerhalb der Laufzeitumgebung benutzen zu können. Wie bereits beschrieben, arbeitet die Laufzeitumgebung von TGrid keine statisch definierten Taskgraphen ab, wie es bei vielen Workflow-Engines der Fall ist. Da in Abhängigkeit der zur Laufzeit dynamisch ausgewerteten Bedingungen Tasks neue Tasks erzeugen können, ist der entstehende Taskgraph vorher nicht bekannt. Das bedeutet auch, dass es keine externe Beschreibung des Taskgraphen gibt, wie z. B. bei DAGMan. Die Definition des Taskgraphen ist demnach direkt im TGrid-Programm enthalten. Dadurch wird eine sehr große Flexibilität geschaffen, da man mit den Mitteln einer Programmiersprache zur Laufzeit beliebig komplexe Bedingungen auswerten kann, welche die Abarbeitung der Tasks beeinflussen

```

1 <tgridconf>
2   <subnetmanagers>
3     <subnet id="ai2">
4       <manager>
5         <address>aranha</address>
6       </manager>
7       <clients>
8         <client>equinox</client>
9       </clients>
10      <env>
11        <property name="prunjava" value="prunjava.sh"/>
12      </env>
13    </subnet>
14
15    <subnet id="clusternet">
16      <manager>
17        <address>clust03</address>
18      </manager>
19      <clients>
20        <client>192.168.2.4</client>
21        <client>192.168.2.5</client>
22        <client>192.168.2.6</client>
23        <client>192.168.2.7</client>
24      </clients>
25      <env>
26        <property name="prunjava" value="prunjava.x86_64.suse93.cluster.sh"/>
27      </env>
28    </subnet>
29  </subnetmanagers>
30
31  <connections>
32    <source subnet="ai2">
33      <destination subnet="*" type="routed">
34        <property name="router" value="router1"/>
35        <property name="protocol" value="tcp"/>
36      </destination>
37    </source>
38    ...
39  </connections>
40
41  <routers>
42    <router id="router1">
43      <property name="address" value="clust01"/>
44      <route subnet="clusternet">
45        <connection type="tcp">
46          <property name="target_host" value="clust02"/>
47        </connection>
48        <connection type="ssh">
49          <property name="target_host" value="localhost"/>
50          <property name="target_port" value="5000"/>
51        </connection>
52      </route>
53    </router>
54    ...
55  </routers>
56 </tgridconf>

```

**Abb. 4.5:** Beispiel einer TGrid-Konfigurationsdatei in XML.

können. Im Gegensatz dazu, ist eine statische Definition von Taskgraphen, z. B. mit XML, stark an die angebotenen XML-Elemente gebunden. Die bei TGrid gewonnene Flexibilität und Unabhängigkeit zieht aber einen größeren Aufwand bei der Programmierung nach sich.

Im TGrid-Programm müssen alle Daten- und Kontrollabhängigkeiten, die zwischen den Tasks bestehen, vom Entwickler mit Hilfe der TGrid-API definiert werden. Außerdem ist der Programmierer dafür verantwortlich, im TGrid-Programm das Scheduling zu veranlassen, die Tasks zu starten und zu überwachen. Darüber hinaus müssen der Datenfluss für die Umverteilung und der Aufruf der Datenumverteilungskomponente von TGrid explizit angegeben werden.

Ein einfaches TGrid-Beispielprogramm wird in Abb. 4.6 veranschaulicht. In diesem Beispiel soll eine Matrix mit gegebener Dimension im Grid erstellt und danach ausgegeben werden. Dazu werden zwei Tasks definiert, eine zum Erzeugen der Matrix (*MatrixGenerate*) und eine zum Ausgeben der Task (*MatrixPrint*). Zuerst wird die Task *MatrixGenerate* instantiiert und für diese angegeben, dass die zu erzeugende Matrix 16 Zeilen und 16 Spalten besitzen soll (Zeilen 5–6). Danach wird diese Task vom Scheduler auf ein Subnet von TGrid abgebildet. Mit der Übergabe der Task-Referenz und des Ablaufplans (*schedule*) kann dann die erste Task gestartet werden (Zeilen 8–11). Nachdem die Task ausgeführt und damit eine Matrix im Grid erzeugt wurde, wird eine Instanz von *MatrixPrint* erstellt. Wie bei der vorherigen Task wird ein Ablaufplan vom Scheduler erzeugt (Zeilen 13–18). Um die Matrix von der Ursprungs- zur Zieltask übertragen zu können, wird die Umverteilung vorbereitet. Dies wird erreicht, indem die logische Abbildung der Matrixelemente festgelegt wird. Im Beispiel soll die rechte obere Teilmatrix an Position (0, 8) übertragen werden (Zeilen 22–24). Die eigentliche Umverteilung kann im Anschluss angestoßen werden (Zeilen 26–29). Es sei nochmals darauf hingewiesen, dass der Programmierer nur die zu übertragenden Elemente angibt. Die eigentliche Kommunikation zwischen den Tasks erledigt die Umverteilungskomponente, die im nächsten Abschnitt beschrieben wird.

## 4.8 Datenumverteilung – Die MxN-Komponente

Damit taskbasiertes Rechnen auf dem Grid erfolgreich sein kann, ist es von entscheidender Bedeutung für Frameworks wie TGrid, dass die Komplexität der Datenumverteilung und des Datentransfers zwischen Tasks durch ein Verlagern in das Framework vor dem Programmierer versteckt wird. Der Aufwand der Programmierung von Grid-Programmen verringert sich damit erheblich und außerdem werden Fehlerquellen ausgeschlossen. Deshalb lag ein Hauptaugenmerk bei der Entwicklung der TGrid-Architektur auf der Bereitstellung einer Umverteilungskomponente, welche die Kopplung (coupling) von M-Tasks und die damit verbundene Datenkommunikation ermöglicht.

In diesem Abschnitt wird, nach Betrachtung von verwandten Ansätzen zur Datenumverteilung, die entsprechende TGrid-Komponente vorgestellt und deren Funktionsweise beschrieben.

### 4.8.1 Verwandte Ansätze

Durch die zunehmende Verbreitung von Grid-Computing und verteilten heterogenen Umgebungen wurden in den letzten Jahren verschiedene Ansätze zur Datenumverteilung vorgestellt. Das Framework PAWS (Parallel Application WorkSpace) bietet ein komponenten-

```

1 public class MatrixTest implements TGridProgram {
2
3     public void run(TGridRuntime runtime) throws TGridRunException {
4
5         TGridComponent matGen = new MatrixGenerate();
6         matGen.setDataDeclaration(MatrixGenerate.GEN_MATRIX, new ArrayDecl(16, 16));
7
8         Schedule[] sched = TGridScheduler.getSchedulerInstance().schedule(matGen);
9
10        TGComponentHandle matGenHandle = runtime.execute(matGen, sched[0]);
11        matGenHandle.waitFor();
12
13        TGridComponent printTask = new MatrixPrint();
14        matPrintComp.setDataDeclaration(MatrixPrint.PRT_MATRIX, new ArrayDecl(8, 8));
15
16        Schedule[] sched2 = TGridScheduler.getSchedulerInstance().schedule(printTask);
17
18        TGComponentHandle matPrintHandle = runtime.execute(printTask, sched2[0]);
19
20        // information required to perform a redistribution
21        // (schedules, component handles, unique id of variables)
22        RedistConfigObject redistConfig = new RedistConfigObject(
23            MatrixGenerate.GEN_MATRIX, matGenHandle, new ArraySelection(0, 8, 8, 8),
24            MatrixPrint.PRT_MATRIX, matPrintHandle, new ArrayMapping(0, 0));
25
26        Redistributor2 redist = new Redistributor2(runtime.getSubnetManager(),
27            redistConfig);
28        redist.start();
29        redist.waitFor();
30
31        // ...
32    }
33 }

```

**Abb. 4.6:** TGrid-Beispielprogramm. Die Komponente MatrixGenerate instantiiert eine Matrix, welche an die Komponente MatrixPrint übertragen und ausgegeben wird.

tenbasiertes Modell, um parallele Applikationen miteinander zu verbinden (*data coupling*) [10]. PAWS verwendet eine andere Abstraktionsstufe als TGrid. Bei PAWS muss der Programmierer selbst die Daten zwischen Sender und Empfänger übertragen und es hat eher zum Ziel, bestehende parallele Programme zu verbinden. Im Gegensatz dazu ist bei gemischt-parallelen Applikationen für TGrid die Aufteilung in einzelne Teilprogramme (Tasks) schon beim Design vorgesehen. Das Model Coupling Toolkit (MCT) [64] ist eine Fortran-Bibliothek, die Funktionen zum Umverteilen von Datenstrukturen zwischen datenparallelen (*message passing*) Programmen bereitstellt. Die Bibliothek ist aber speziell für die Entwicklung von taskparallelen Applikationen in homogenen Umgebungen konzipiert wurden. InterComm [65] ist ein Framework zum Verbinden von wissenschaftlichen Programmen, die auf verteilten Systemen komplexe Datenverteilungen verwenden. Zielapplikationen von InterComm sind vor allem physikalische Simulationen auf homogenen Clustern, die in Fortran und C++/P++ geschrieben sind. Ähnlich zum Ansatz in TGrid arbeitet InterComm mit zwei Abstraktionsstufen, wobei eine die Verteilung der Daten (*distribution*) und die andere die Abbildung zwischen zwei Verteilungen (*linearization*) beschreibt. Ein weiteres aktuelles Forschungsprojekt wird vom CCA-Forum<sup>13</sup> (*common*

<sup>13</sup><http://www.cca-forum.org/>

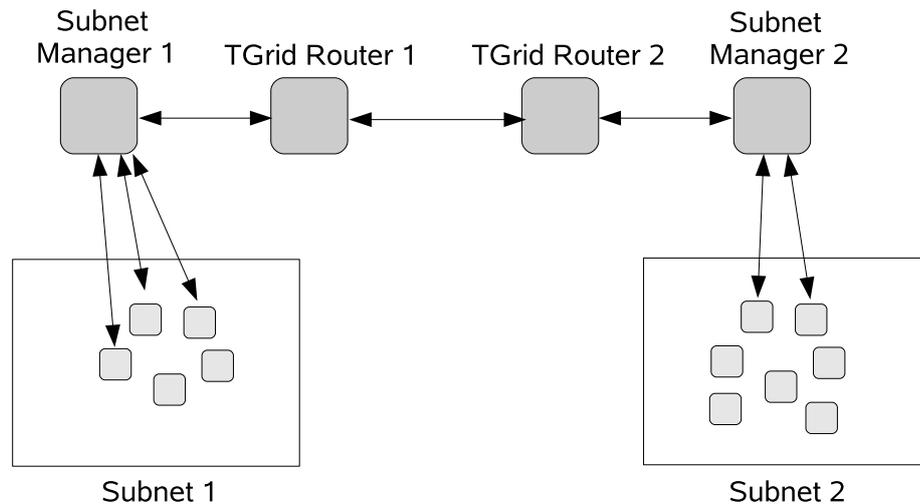
*component architecture*) vorangetrieben, welche speziell an Standards für die Entwicklung von komponentenbasierten Architekturen zur Unterstützung des Hochleistungsrechnens arbeitet. Ein wichtiger Teil entfällt dabei auf die MxN-Arbeitsgruppe, die an der Definition und der Implementierung von Schnittstellen arbeitet, um Daten zwischen Komponenten auszutauschen, die auf einer unterschiedlichen Anzahl von Prozessoren und auf verschiedenen Rechnern ausgeführt werden [12]. Eine CCA-kompatible Implementierung einer MxN-Komponente für *Ccaffeine* [2] wurde durch das Framework Seine [116] bereitgestellt. Seine arbeitet wie TGrid mit einer geometrischen Methode zur Bestimmung der notwendigen Nachrichten. Aus der Definition der Datenverteilung von Quell- und Zielkomponente bestimmt Seine die überlappenden Regionen und generiert daraus die Datenpakete, die dann vom Framework zwischen den Komponenten ausgetauscht werden. Andere zu CCA kompatible Frameworks, welche das Verbinden von verteilt arbeitenden Komponenten unterstützen, sind DCA [11] und XCAT [63]. Auch die Erstellung von Kommunikationsschedules zum Umverteilen von Daten zwischen zwei parallelen Programmen war bereits Gegenstand aktueller Forschung. In [58] wurde ein effizientes Verfahren entwickelt, um bei der Umverteilung von Daten eine möglichst hohe Ausnutzung der zur Verfügung stehenden Bandbreite (vor allem im Backbone) zwischen den Prozessoren zu gewährleisten.

## 4.8.2 Aufbau der Datenumverteilungskomponente

Eine Datenumverteilung – auch MxN genannt – ist für das Verbinden von verteilt arbeitenden parallelen Programmen (oder Komponenten) notwendig, wenn Datenabhängigkeiten zwischen ihnen bestehen. MxN (gesprochen „M by N“) steht dabei für die Datenumverteilung von  $M$  Prozessoren des Ausgangsprogramms auf  $N$  Prozessoren des Zielprogramms. Bei Betrachtung der verwandten Ansätze stellt man fest, dass die Datenumverteilung meist mit einem ähnlichen Muster arbeitet. Die Basisschritte dieses Musters sind: Identifikation der Daten, Generierung der zu verschickenden Nachrichten, Planung (Scheduling) des Nachrichtenaustauschs und letztendlich die Kommunikation. Innerhalb des TGrid-Frameworks ist die Datenumverteilung in folgende Schritte gegliedert:

1. Festlegen der Quell- und Zielvariablen der Tasks (*specification*). Die Quellvariable definiert die Daten, welche von der sendenden Komponente bereitgestellt werden. Demzufolge gibt die Zielvariable die Datenstruktur der empfangenden Task an.
2. Auswahl der Elemente der Quelldaten, die zur Zielkomponente übertragen werden sollen (*selection*).
3. Definition der Abbildung der Daten auf die Zielkomponente (*mapping*).
4. Erstellung eines Ablaufplans (*schedule*) der Nachrichten, der die korrekte Reihenfolge der Nachrichten zum Transfer der Datenstrukturen festlegt (*scheduling*).
5. Starten des Nachrichtenaustauschs (*communication*).

Eine Umverteilung kann gestartet werden, wenn die Quelltask die Ausführung beendet hat und die Zieltask auf den zugewiesenen Prozessoren gestartet wurde. Die MxN-Komponente erstellt aus den Meta-Informationen (*specification, selection, mapping*) den Ablaufplan des Nachrichtenverkehrs. Die für die Erstellung und das Versenden der Nachrichten



**Abb. 4.7:** Beispielkonfiguration einer Inter-Subnet-Kommunikation in TGrid.

notwendigen Informationen sind leicht aufzulisten (z. B. Zielnetz, Zielprozessor, Zieltask, etc.). Es ist jedoch weitaus schwieriger, die MxN-Komponente als einfach zu benutzendes und erweiterbares Software-Modul zu gestalten. Um der Erweiterbarkeit Sorge zu tragen, verwendet TGrid generische Schnittstellen für die Methoden der Umverteilungskomponente. Damit haben Entwickler die Möglichkeit, nicht vorhandene Datentypen und Datenverteilungen selbst hinzuzufügen (z. B. ein blockzyklisch verteiltes 2-dimensionales Feld von Ganzzahlen). Beim Entwurf der MxN-Komponente ist neben der einfachen Verwendbarkeit und den Erweiterungsmöglichkeiten die Leistung und der Durchsatz von Bedeutung. In TGrid wird deshalb zwischen zwei Typen der Datenumverteilung unterschieden: der Umverteilung innerhalb eines Teilnetzes (*intra-subnet*) und der Umverteilung zwischen zwei Teilnetzen (*inter-subnet*). Die Intra-Subnet-Umverteilung kann eingesetzt werden, wenn sich die Quell- und Zieltask im selben Teilnetz befinden. In diesem Fall gehören die Prozessoren beider Tasks zum gleichen IP-Adressraum, wodurch ein direktes Senden und Empfangen der Prozessoren möglich ist und ein hoher Durchsatz garantiert wird. Liegen die Tasks und damit die zugeordneten Prozessoren in verschiedenen Teilnetzen, kann normalerweise keine direkte Verbindung zwischen den Prozessoren aufgebaut werden. Das liegt zum einen an restriktiven Firewalls, zum anderen verwenden viele Cluster-Systeme einen privaten IP-Adressraum, womit eine explizite Übersetzung der Adressen von Routern nötig wird. Diese Einschränkungen bei der Inter-Cluster-Kommunikation erforderten die Entwicklung von TGrid-Softwaremodulen, um einen Datenaustausch zwischen zwei privaten Teilnetzen zu erlauben. Um eine Inter-Cluster-Kommunikation zwischen Teilnetzen durchzuführen, die durch Firewalls geschützt und mit privaten Adressräumen versehen sind, ist beispielsweise die in Abb. 4.7 dargestellte Konfiguration von TGrid erforderlich. Sollen Nachrichten von Prozessoren aus Subnet 1 an Prozessoren aus Subnet 2 geschickt werden, müssen diese erst an den lokalen Subnet-Manager weitergeleitet und von dort zu einem TGrid-Router übermittelt werden. Der Router kennt die möglichen Kommunikationspfade (tcp, ssh, http) zum Zielnetz und leitet die Nachrichten entsprechend weiter.

Die Anzahl dieser Teilstrecken (hops) und die verfügbare Bandbreite der einzelnen Verbindungen beeinflussen die erreichbare Latenzzeit zwischen zwei Prozessoren und ebenso den Datendurchsatz.

Ein weiteres wichtiges Ziel beim Design der Umverteilungskomponente von TGrid ist die Unterstützung von parallel ausgeführten Umverteilungen zwischen Tasks. Dies kommt besonders zum Tragen, wenn eine Task mehrere Kinder besitzt und deren Datenabhängigkeiten erfüllt werden müssen. In diesen Fällen ist das parallele Kopieren der Ergebnisdaten der Elterntask auf die Kindtasks erstrebenswert. Die MxN-Komponente von TGrid unterstützt diese Vorgaben durch eine komplett Thread-basierte Implementierung, womit eine Umverteilung zwischen je zwei Paaren von Tasks zur selben Zeit ermöglicht wird.

### 4.8.3 Protokoll zur Datenumverteilung

Für die Realisierung der Umverteilungskomponente von TGrid wurde ein Protokoll entwickelt, welches von den beteiligten Entitäten (Tasks, Subnet-Manager, Redistributor) implementiert werden muss. Anstatt einer formalen Beschreibung des Protokolls mit API, Ereignissen und Fehlercodes wird die Funktionsweise des Protokolls im Folgenden an einem Beispiel erläutert, welches in Abb. 4.8 illustriert ist. In diesem Beispiel wird ein typischer Anwendungsfall dargestellt, bei dem ein Teil einer Matrix (`MATRIX_A`) aus Subnet A auf eine Matrix (`MATRIX_B`) in Subnet B abgebildet werden soll. Die kontrollierende Instanz der Umverteilung ist der so genannte *Redistributor*. Der Redistributor wird im gleichen Subnet gestartet, in dem auch das Benutzerprogramm gestartet wurde. Im betrachteten Fall läuft der Redistributor auf einem dritten Subnet, dem Subnet C. Der Programmierer gibt unter Zuhilfenahme der Task-Schnittstellen (vgl. Abschnitt 4.5) im TGrid-Programm an, welche Elemente zwischen den Tasks übertragen werden sollen. Zu den notwendigen Informationen gehören die folgenden Angaben:

- Eindeutige Bezeichner (*IDs*) der Variablen, über die TGrid auf die assoziierten Datenstrukturen der Quell- und Zieltask zugreifen kann.
- Eine Auswahl (Selektion) der Elemente der Datenstruktur in der Quelltask und eine Abbildung in die Datenstruktur der Zieltask. Beispielsweise soll der Teil einer Matrix, der durch das Rechteck mit den Punkten (0,0) und (4,4) begrenzt wird, an die Position (1,1) der Zielmatrix abgebildet werden.
- Die Ablaufpläne (Schedules) der Quell- und Zieltask. Diese enthalten die den Tasks zugeordneten Teilnetze und Prozessoren. Diese Angaben ermöglichen es der Umverteilungskomponente, Informationen zu dem verwendeten Datenlayout einer in Ausführung befindlichen Task zu gewinnen.
- Objekt-Referenzen der Quelltask und der Zieltask.

Die Umverteilungskomponente (Redistributor) erstellt aus diesen Informationen den abstrakten Kommunikationsplan (*abstract communication schedule*). Der abstrakte Kommunikationsplan enthält eine Liste von Nachrichten, die zwischen den beiden laufenden Tasks ausgetauscht werden müssen. Diesen Nachrichten liegt eine Berechnung der geometrischen

Überdeckung der Datenverteilungen beider Tasks zu Grunde. Die Nachrichten enthalten noch keine spezifischen Angaben zu den verwendeten Plattformen, wie IP-Adressen oder Rechnernamen. Aufgrund dessen muss der abstrakte Kommunikationsplan in einen systemabhängigen Plan umgewandelt werden, bevor er angewendet werden kann. Dazu werden die abstrakten Nachrichten mit weiteren Informationen (header) versehen, um die Quell- und Zieldatenstrukturen in den Tasks zu bezeichnen. Eine Datenstruktur (Variable / Speicherplatz) kann in TGrid mit folgenden Angaben eindeutig identifiziert werden:

- Der Bezeichner des Teilnetzes (*subnet*).
- Die ID der Task, welche von TGrid pro Subnet eindeutig zur Laufzeit vergeben wird (ähnlich einer Prozess-ID unter UNIX).
- Der Name der Variable innerhalb der Task, die mit der Datenstruktur verknüpft ist.

Die Namen der Variablen und des Teilnetzes sind schon bei der Erzeugung des abstrakten Kommunikationsplans bekannt. Da die Task-ID aber zur Laufzeit vergeben wird, muss der Redistributor eine Anfrage an beide Teilnetze stellen. Die Subnet-Manager der Quell- und Zieltask übermitteln daraufhin diese ID an den anfragenden Redistributor. Danach stehen alle Informationen zur Verfügung, um einen systemabhängigen Kommunikationsplan zu erstellen.

Der Redistributor startet danach auf den Teilnetzen der sendenden und empfangenden Task Umverteilungsmanager (redistribution manager). Diese Umverteilungsmanager übernehmen die Aufgabe der Kontrolle des Datentransfers und ermöglichen das direkte Senden zwischen den Teilnetzen, ohne dass Nachrichten über das Teilnetz des Redistributors geroutet werden müssen. Die lokalen Umverteilungsmanager haben auch den Vorteil, dass Protokollnachrichten zwischen Manager und Client-Prozessoren, z. B. zur Synchronisation und zur Statusänderung, nur innerhalb eines Netzwerks verschickt werden.

Da TGrid mehrere gleichzeitig ablaufende Umverteilungsoperationen unterstützt, ist es notwendig, die einzelnen Umverteilungsprozesse voneinander unterscheiden zu können. Aus diesem Grund wird jedem Umverteilungsmanager bei der Instantiierung eine eindeutige ID zugewiesen. Diese IDs werden beim Start der Manager, wie in Abb. 4.8 dargestellt, untereinander ausgetauscht. Bei diesem Austausch dient der ursprüngliche Redistributor als Vermittler, da nur er die IDs eindeutig zuweisen kann. Nach dem Austausch der IDs können die Manager untereinander kommunizieren, ohne dass der Redistributor involviert werden muss. Der Manager der Zieltask weist zuerst jedem empfangenden Prozessor einen eindeutigen Port zu. Dieser Port ist pro Umverteilungsoperation eindeutig, wodurch eine Task von zwei Elterntasks unter Verwendung von zwei Ports gleichzeitig empfangen kann. Nachdem die Ports zugeordnet wurden, schickt jeder Prozessor eine Nachricht an den lokalen Empfangsmanager, dass er zum Datentransfer bereit ist. Anschließend übermittelt der Empfangsmanager dem Manager der sendenden Seite die Metadaten der Empfangsseite (*hostname, port, rank*). Der Sendemanager erstellt mit diesen Informationen ein Proxy-Objekt für alle sendenden Prozessoren, um das Ziel der Übertragung vor den Prozessoren zu verstecken. Mit dieser transparenten Implementierung der Sendeoperation weiß ein Prozessor nicht, ob sich der Zielprozessor im gleichen Teilnetz oder in einem entfernten Teilnetz

befindet. Mit dem Erhalt des Proxys und der Nachrichtenliste beginnt jeder Prozessor der Quelltask, Daten an das Zielnetz zu übertragen. Nach Abschluss der Kommunikation benachrichtigen die sendenden und empfangenden Prozessoren ihren jeweiligen Manager über das Ende der Operation.

Es wird speziell darauf hingewiesen, dass das Senden und Empfangen der Nachrichten nicht von den Tasks vollzogen wird. Dies würde bedeuten, dass der Entwickler den Code zur Umverteilung für jede Task bereitstellen müsste. Stattdessen bietet der Container die Funktionalität zum Datentransfer zwischen Tasks an, vgl. Abschnitt 4.5. Da ein Container den internen Aufbau der Task und die verwendeten Datenstrukturen nicht kennt, verwendet dieser die vorher beschriebenen abstrakten Schnittstellen zur Identifikation der Variablen, zur Bestimmung des verwendeten Datenlayouts und zum Zugriff auf Datenstrukturen über die eindeutigen Bezeichner der Variablen.

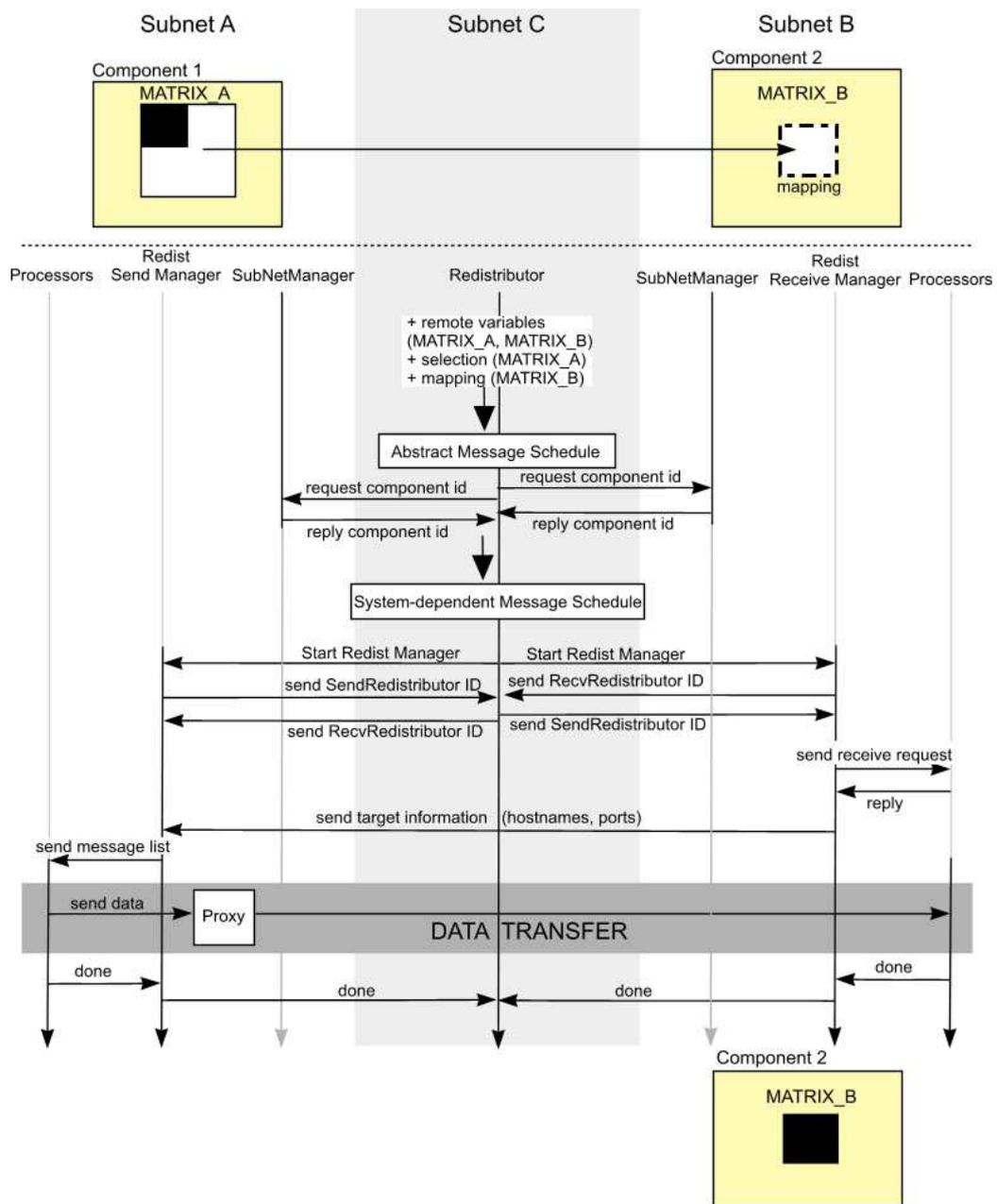
## 4.9 Experimentelle Analyse von TGrid

In diesem Abschnitt wird die Leistungsfähigkeit von TGrid anhand kleinerer Testszenarien mit unterschiedlichen gemischt-parallelen Implementierungen von Algorithmen untersucht. Bei dieser Analyse ist zu klären, ob eine taskparallele Ausführung von Algorithmen im Grid mit einer verbesserten Laufzeit und einer guten Skalierbarkeit gelingt. Des Weiteren werden in diesem Zusammenhang die Umverteilungsroutinen von TGrid experimentell evaluiert.

### 4.9.1 Matrixmultiplikation nach Strassen

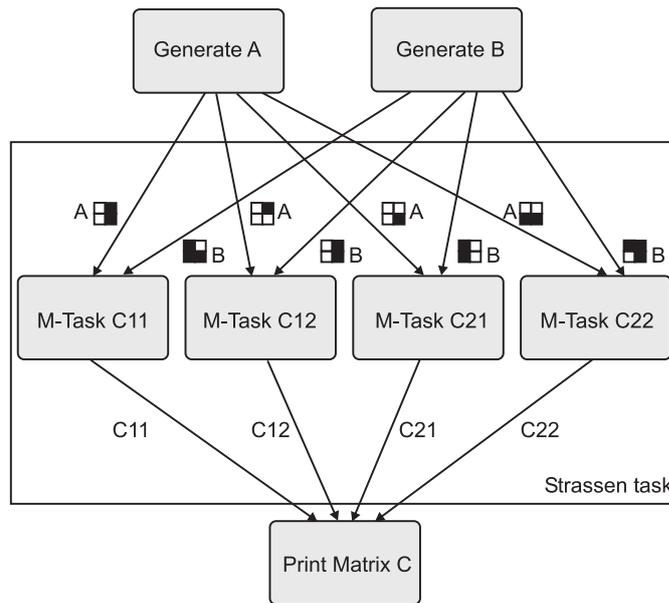
Als erster Algorithmus wurde die Matrixmultiplikation nach Strassen als TGrid-Programm realisiert. Die taskparallele Struktur und damit die Aufteilung entspricht der algorithmischen Variante, mit der in Kapitel 2 gute Laufzeiten auf homogenen Clustern erreicht werden konnten. Die Leistung einer parallelen Implementierung von Strassen ist durch die große Anzahl von Datenabhängigkeiten begrenzt [35]. Demzufolge ist die Aufgabe anspruchsvoll, für die taskparallele Matrixmultiplikation nach Strassen auf einer heterogenen Plattform einen ähnlichen Speedup zu erzielen wie zuvor auf homogenen Systemen.

Die Matrixmultiplikation nach Strassen besitzt verschiedene Eigenschaften, die sie als Applikation für TGrid interessant machen: Erstens lässt sich der Algorithmus wie in Kapitel 2 beschrieben in mehrere Teilaufgaben zerlegen. Zweitens kann der Algorithmus rekursiv definiert werden. Beide Eigenschaften gehen optimal mit dem Programmiermodell von TGrid einher, da TGrid dynamisch erzeugte Taskgraphen unterstützt. Damit hat der Programmentwickler die Kontrolle, wann die Rekursion beendet werden soll, und kann dies dynamisch auswerten. Um gemischt-parallele Algorithmen schnell und möglichst fehlerfrei implementieren zu können, ist es darüber hinaus wichtig, dass das Grid-Framework eine Unterstützung bei der Umverteilung von Datenstrukturen mitbringt. Speziell bei der Matrixmultiplikation nach Strassen werden die Eingabematrizen in vier gleich große Teilmatrizen zerlegt, welche dann an die entsprechenden Tasks übertragen werden müssen. Wie in Abschnitt 4.8 vorgestellt, besitzt TGrid eine solche MxN-Umverteilungskomponente, die



**Abb. 4.8:** Umverteilungsprotokoll der MxN-Komponente.

es möglich macht, Teilmatrizen zu selektieren und in Kindtasks abzubilden. Die TGrid-Applikation zur Matrixmultiplikation nach Strassen wurde, wie in Abb. 4.9 veranschaulicht, mit drei Tasks realisiert. Zu Beginn werden die Matrizen A und B durch die Task „Generate“ verteilt angelegt. Danach wird die Task „Strassen“ ausgeführt, die wiederum aus den vier Teiltasks  $C_{ij}$ ,  $1 \leq i, j \leq 2$ , besteht. Nach Abschluss der Multiplikation wird die Ergebnismatrix C an die Task „Print“ übergeben, welche diese auf die Kommandozei-



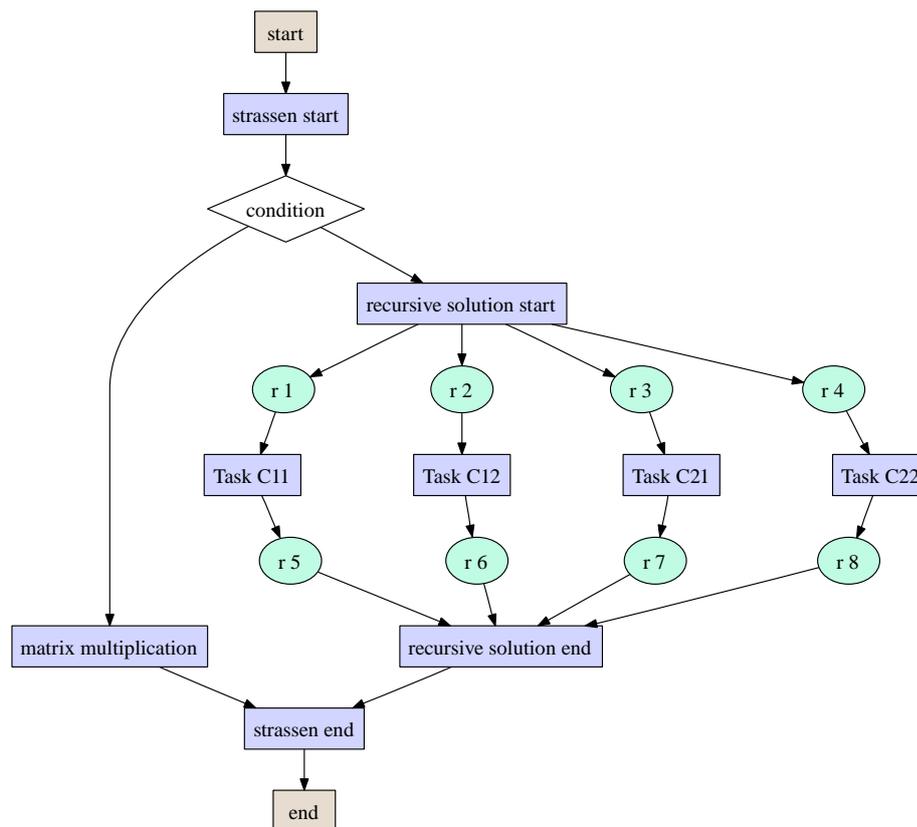
**Abb. 4.9:** Gesamtstruktur des TGrid-Programms für die Matrixmultiplikation nach Strassen.

le oder in eine Datei ausgibt. Die Tasks zum Erzeugen und zum Ausgeben der verteilten Matrizen sind generisch, d. h. sie können für andere Algorithmen wiederverwendet werden.

Die Arbeitsweise der Task „Strassen“ zur Matrixmultiplikation ist in Abb. 4.10 dargestellt. Das Problem der Berechnung  $C = A \times B$  wird rekursiv in vier Teilprobleme zur Bestimmung von  $C_{11}$ ,  $C_{12}$ ,  $C_{21}$  und  $C_{22}$  zerlegt. Für die Lösung jedes Teilproblems wurde eine entsprechende TGrid-Task realisiert, welche die Eingabedaten empfängt und intern die Strassen-Task rekursiv aufruft. Aus diesem Grund wurde ein Bedingungsknoten in den Taskgraph eingefügt, bei dem entweder eine weitere rekursive Zerlegung oder die eigentliche Matrixmultiplikation aufgerufen wird. Für die Berechnung der Teilmatrizen  $C_{ij}$  mit insgesamt sieben Matrixmultiplikationen wurde das Schema-Q7 gewählt, bei dem die Task  $C_{22}$  eine Teilmatrix berechnet und die anderen jeweils zwei, vgl. Abschnitt 2.4.3.

Damit die Laufzeittests nicht von zu vielen Parametern abhängen, wurde nur eine Rekursionsstufe von Strassens Algorithmus angewendet. Demzufolge können jeder M-Task nach Abbruch der Rekursion mehrere Prozessoren zugeordnet sein. Damit muss ein weiterer Algorithmus zur parallelen Berechnung der Matrixmultiplikation zum Einsatz kommen. Da die TGrid-Programme portabel sind, können hierfür nicht wie in Kapitel 2 die C- und Fortran-Schnittstellen der ScaLAPACK-Bibliotheken benutzt werden. Aus diesem Grund wurde ein Broadcast-Multiply-Roll-Algorithmus (ähnlich zu SUMMA [108]) als M-Task implementiert.

Die Experimente wurden auf einer heterogenen Menge von Computern durchgeführt. Die beiden Konfigurationen, die zum Einsatz kamen, sind in Tab. 4.1 zusammengefasst. In der Konfiguration (1) werden zwei Subnets (a und b) benutzt, die beide von je einem Subnet-Manager kontrolliert werden. Die Matrizen A und B werden jeweils auf einem Pentium-4-System in Subnet (a) angelegt. Die eigentliche Berechnung geschieht mit den Tasks  $C_{ij}$ ,  $1 \leq i, j \leq 2$  in Subnet (b). Jeder dieser Berechnungstasks werden vier Prozes-

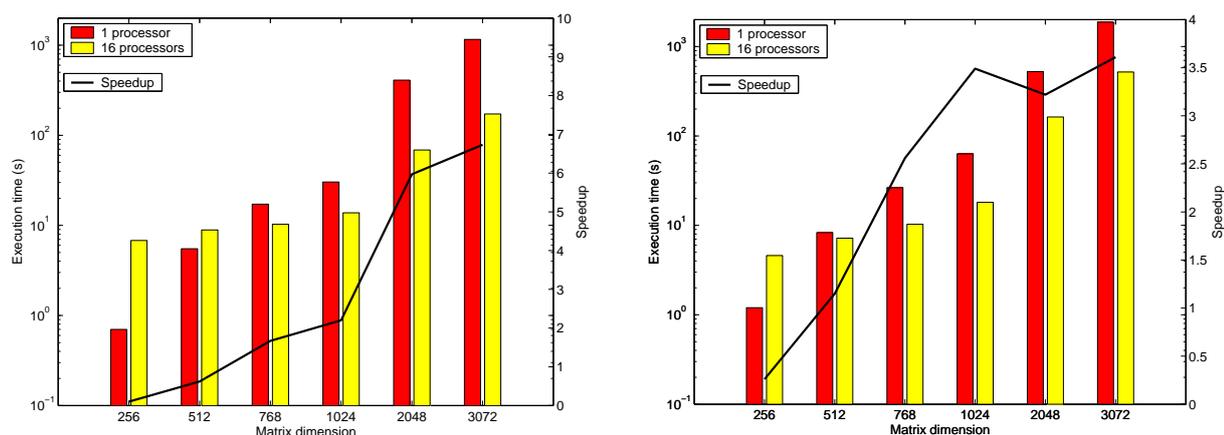


**Abb. 4.10:** TGrid-Task zur Matrixmultiplikation nach Strassen. Die Knoten  $r_1, r_2, \dots, r_8$  bezeichnen Matrizenumverteilungen.

**Tab. 4.1:** Gridkonfigurationen zum Test der Matrixmultiplikation nach Strassen.

Konfiguration (1)				Konfiguration (2)			
Task	Prozesse pro Knoten	Prozessor (GHz)	Subnet	Task	Prozesse pro Knoten	Prozessor (GHz)	Subnet
Generate	1 x 1	P4 (3.0)	a	Generate	1 x 1	P4 (3.0)	a
Generate	1 x 1	P4 (3.0)	a	Generate	1 x 1	P4 (3.0)	a
C11	2 x 2	Opteron (2.0)	b	C11	2 x 2	Opteron (2.0)	b
C12	2 x 2	Opteron (2.0)	b	C12	2 x 2	Opteron (2.0)	b
C21	2 x 2	Opteron (2.0)	b	C21	2 x 2	P4 (3.0)	a
C22	2 x 2	Opteron (2.0)	b	C22	2 x 2	P4 (3.0)	a

soren des Subnets (b) zugewiesen, welche logisch in einem  $2 \times 2$ -Prozessorgitter zum Ausführen der parallelen Matrixmultiplikation angeordnet werden. Die zuvor beschriebenen Tasks zur Erzeugung und zur Ausgabe der Matrizen werden ausgeführt, aber werden nicht in die Laufzeitmessungen einbezogen. Demgegenüber stehen die Laufzeitmessungen mit Konfiguration (2), mit der die Berechnung in zwei verschiedenen Teilnetzen getestet wird. Dies erzeugt eine größere Heterogenität unter den Prozessoren bei der Berechnung der Matrizen als in Konfiguration (1). Die Tasks  $C_{11}$  und  $C_{12}$  werden von Dual-Opteron-Systemen in Subnet (b) ausgeführt, wohingegen die anderen beiden Tasks von Pentium-4-Prozessoren berechnet werden.

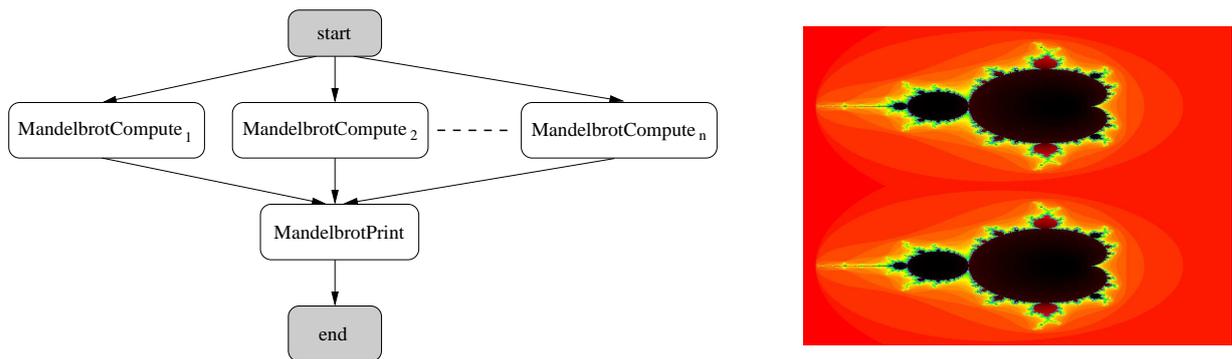


**Abb. 4.11:** Laufzeit und Speedup der Matrixmultiplikation nach Strassen mit TGrid. Links: Berechnung mit einem Subnet, Konfiguration (1). Rechts: Verwendung von zwei Subnets, Konfiguration (2).

Die Dual-Opteron-Prozessoren sind untereinander mit Gigabit-Ethernet und die Pentium-4-Prozessoren mit Fast-Ethernet (100 MBit/s) verbunden. Die Tab. 4.1 enthält auch die Angabe „Prozesse pro Knoten“. Der Wert  $2 \times 2$  bedeutet, dass zwei Prozesse auf zwei Knoten gestartet werden. Für zwei Knoten mit Dual-Opteron-Prozessoren stehen damit insgesamt vier physische Prozessoren zur Verfügung. Die Pentium-4-Maschinen verwenden jedoch die Hyperthreading-Technologie, womit bei einer  $2 \times 2$ -Belegung nur zwei physische Prozessoren benutzt werden.

Die Laufzeitergebnisse für beide Konfigurationen sind in Abb. 4.11 gegenübergestellt. In der linken Grafik sind die Ergebnisse mit Konfiguration (1) und in der rechten mit Konfiguration (2) dargestellt. Für beide Konfigurationen wurde die Laufzeit der gemischt-parallelen Matrixmultiplikation nach Strassen bei unterschiedlichen Matrixgrößen mit einem und mit 16 Prozessoren gemessen. Die Laufzeit ist logarithmisch auf der linken y-Achse aufgetragen; aus dem Verhältnis der Laufzeiten mit einem und mit 16 Prozessoren lässt sich der Speedup berechnen, der auf der rechten y-Achse aufgetragen wurde. Der Speedup basiert auf der Ausführung des Programms in TGrid mit einem Prozessor.

Es ist zu erkennen, dass die parallele Ausführung mit TGrid erst ab einer bestimmten Matrixgröße zu einer verbesserten Laufzeit führt. Für kleinere Matrixdimensionen ( $< 1024$ ) ist der Overhead für das Umverteilen der Matrizen und die damit verbundenen Kommunikationskosten zu hoch, als das sich die Berechnung mit mehreren Prozessoren auszahlen würde. Aus diesem Grund ist es auch nicht überraschend, dass der Speedup mit zunehmender Matrixgröße zunimmt, da die Berechnungszeit stärker zunimmt ( $O(n^3)$ ) als die Kommunikationszeit ( $O(n^2)$ ). Im Fall von Konfiguration (1) wird für 16 Prozessoren ein maximaler Speedup von ca. 7 erreicht, was bei der großen Anzahl von Umverteilungen sehr bemerkenswert ist. Falls mehrere Subnets mit unterschiedlichen Prozessortypen die Matrixmultiplikation bearbeiten, wie in Konfiguration (2), führt dies zu einem kleineren Speedup. Zur Berechnung des Speedups wird die Laufzeit auf einem Opteron-Prozessor zu Grunde gelegt, da dieser in der betrachteten Konfiguration schneller als der P4 war. Ein



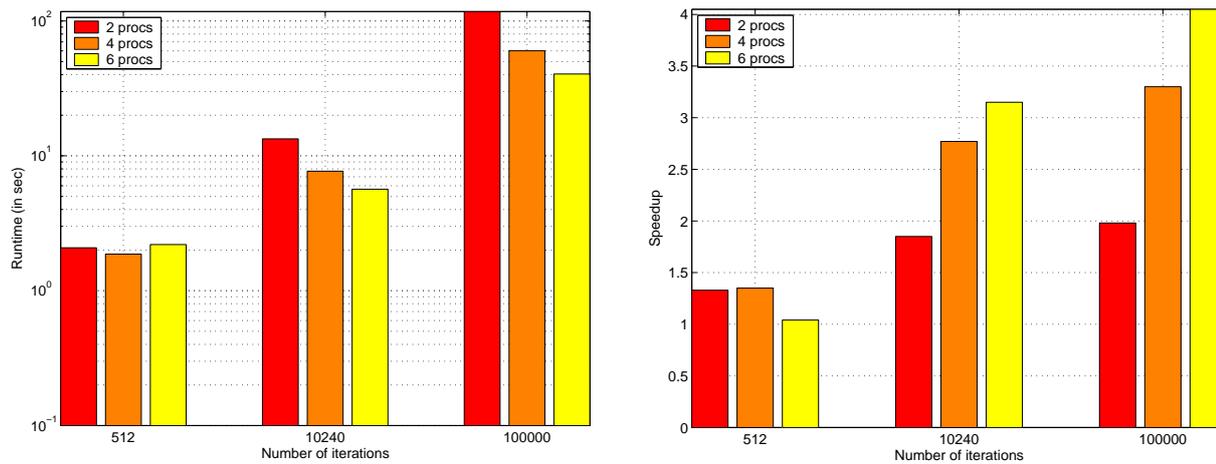
**Abb. 4.12:** Taskgraph des Mandelbrot-Programms (links) und Beispielausgabe bei der Ausführung mit zwei Berechnungstasks.

weiterer Grund für die geringe Leistung dieser Konfiguration ist die kleinere Anzahl an physikalischen Prozessoren, da die P4-Prozessoren nur via Hyperthreading zwei virtuelle Prozessoren aufweisen. Trotzdem zeigte sich, dass die Implementierung auch mit dieser Konfiguration skaliert, da der Speedup mit wachsender Matrixgröße zunimmt.

#### 4.9.2 Parallele Berechnung eines Mandelbrot-Fraktals

Die Berechnung eines Mandelbrot-Fraktals ist als Anwendung sehr gut für eine Ausführung im Grid geeignet, da nur wenige Datenabhängigkeiten zwischen aufeinanderfolgenden Tasks existieren. Im Gegensatz zum Algorithmus von Strassen müssen mit den Teilbildgrenzen nur ein paar Bytes übertragen werden, damit das korrespondierende Bild des Mandelbrot-Fraktals berechnet werden kann. Es ist zu beachten, dass sich die Berechnungszeiten der Teilbilder unterscheiden können, da die Berechnungszeit der Farbwerte der Pixel von der Rekursionstiefe der Mandelbrot-Formel abhängt, die von Pixel zu Pixel variieren kann. Da bei der Realisierung einer taskbasierten Berechnung eines Fraktals nicht die optimale Auslastung aller Prozessoren im Vordergrund steht, werden die zu berechnenden Zeilen des Mandelbrot-Bildes via Round-Robin-Verfahren auf die Berechnungstasks verteilt. Mit dieser Aufteilung der Zeilen können ähnliche Ausführungszeiten der M-Tasks erreicht werden. Das Mandelbrot-Programm für TGrid besteht aus einer variablen Anzahl an Berechnungstasks, die an die Anzahl der Prozessoren geknüpft werden kann. Der zugehörige Taskgraph ist in Abb. 4.12 (links) dargestellt. Die Anzahl der Berechnungstasks kann beim Start des Programms festgelegt werden. Jede Berechnungstask bekommt die zu berechnenden Zeilen und die maximale Iterationstiefe als Startparameter übergeben. Mit der Iterationstiefe wird die Berechnungszeit der Tasks beeinflusst. Eine größere Iterationstiefe führt auch zu einer längeren Berechnungszeit. Zur Ausgabe des Mandelbrot-Bildes dient eine Ausgabekomponente, welche die berechneten Zeilen von den Tasks aufammelt, die Farben zuweist und als Bild ausgibt, siehe Abb. 4.12 (rechts).

In diesem Experiment werden zwei TGrid-Subnets miteinander verbunden. Das erste Subnet besteht aus drei Opteron-Prozessoren mit jeweils 2 GHz, welche über einen Gigabit-Ethernet-Switch verbunden sind. Das zweite Subnet besteht aus drei P4-Prozessoren, die alle mit 3 GHz getaktet und über Fast-Ethernet verbunden sind. In allen Experimen-



**Abb. 4.13:** Leistungsdaten des Mandelbrot-Programms für verschiedene Prozessoranzahlen; links: Ausführungszeit, rechts: Speedup.

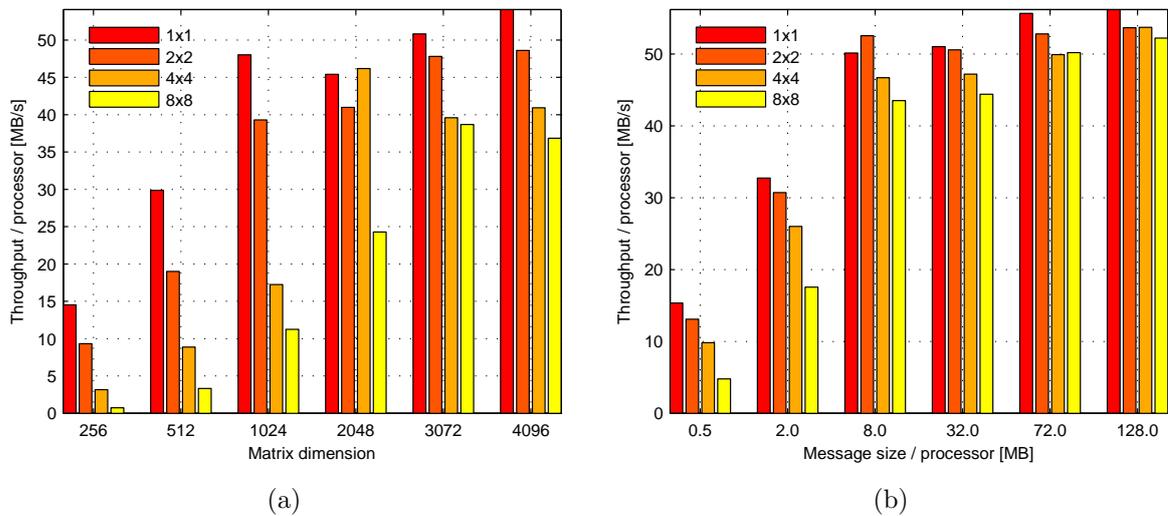
ten wurde ein Mandelbrot-Fraktal mit 1024 Spalten und 768 Zeilen berechnet. Es wurden Tests mit jeweils 2, 4 und 6 Berechnungstasks durchgeführt. Da die Mandelbrot-Tasks keine M-Tasks sind, wurde jeder Task genau ein Prozessor zugewiesen. Die Tasks wurden zuerst auf die P4-Subnets abgebildet. Wurden mehr Prozessoren benötigt, kamen die Opteron-Prozessoren hinzu. Die Abb. 4.13 stellt die Ergebnisse der Laufzeittests für das Mandelbrot-Programm dar. Die Berechnungszeit des Fraktals ist in der linken und der dazugehörige Speedup in der rechten Grafik veranschaulicht. Als Basis für die Berechnung des Speedups diente die Ausführungszeit des Mandelbrot-Programms mit einem Pentium-4-Prozessor. Die Resultate zeigen, dass eine parallele Berechnung eines Mandelbrot-Fraktals sehr gut für eine Grid-Umgebung wie TGrid geeignet ist. Für eine heterogene Zusammensetzung von Prozessoren konnte für sechs Prozessoren immerhin ein Speedup von etwa 4 erzielt werden. Noch bessere Leistungswerte können sicherlich erreicht werden, wenn die Prozessoren ihrer Geschwindigkeit entsprechend, eine unterschiedliche Anzahl von Zeilen berechnen.

### 4.9.3 Experimentelle Evaluation der MxN-Komponente

Um die Leistung der Umverteilungskomponente von TGrid zu evaluieren und den Overhead des TGrid-Protokolls zu quantifizieren, werden unterschiedliche Experimente durchgeführt. Im ersten Experiment wird der Durchsatz pro Knoten ermittelt, der bei einer Intra-Subnet-Kommunikation erzielt werden kann. Das zweite Experiment untersucht die Kommunikationsleistung einer Datenumverteilung zwischen zwei entfernten Teilnetzen (Inter-Subnet-Kommunikation).

Beide Tests wurden auf einem Cluster mit 32 Knoten (je 2 AMD-Opteron-Prozessoren, Modell 246, 2 GHz) durchgeführt (siehe [Opteron-Cluster](#) in Anhang B). Jeder Knoten des Clusters besitzt verschiedene Netzwerkkarten: 100-MBit-Ethernet und Gigabit-Ethernet.

Der Versuchsaufbau ist wie folgt: Das getestete TGrid-Programm besteht aus zwei kooperierenden M-Tasks. Die erste M-Task (T1) erzeugt eine Matrix aus Double-Werten



**Abb. 4.14:** (a) Durchsatz der Datenumverteilung innerhalb eines Subnets. (b) Durchsatz der Datenumverteilung bei konstanter Nachrichtengröße. Gigabit-Ethernet.

mit gegebener Größe. Nachdem T1 die Matrix erstellt hat, wird ein zweite M-Task (T2) gestartet. Diese benötigt die Matrix aus T1 und fordert diese Matrix bei der TGrid-Umverteilungskomponente an. Die Umverteilungskomponente von TGrid erstellt mit den von T1 und T2 gegebenen Informationen über Datenverteilung und Anzahl der Prozessoren einen Ablaufplan der Datenumverteilung. Dieser enthält die zu sendenden Nachrichten zwischen T1 und T2. Beide M-Tasks erhalten diesen Plan und die Datenübertragung wird gestartet. Gemessen wird die Zeit zwischen Datenanfrage von T2 und der erfolgreichen Fertigstellung aller Kommunikationsoperationen auf Sende- und Empfangsseite. Als zu übertragende Datenstruktur wird eine blockverteilte quadratische Matrix verwendet, wie sie in Kapitel 2 benutzt wird (siehe tpMM).

### Intra-Subnet-Kommunikation

In diesem Experiment soll der Durchsatz ermittelt werden, der bei einer Umverteilung von Matrizen in TGrid erreicht werden kann. Damit eine Intra-Subnet-Kommunikation zwischen zwei M-Tasks stattfinden kann, müssen die M-Tasks dem gleichen Teilnetz zugeordnet sein. Da sie somit demselben (privaten) IP-Adressraum angehören, können Nachrichten direkt ausgetauscht und müssen deshalb nicht geroutet werden.

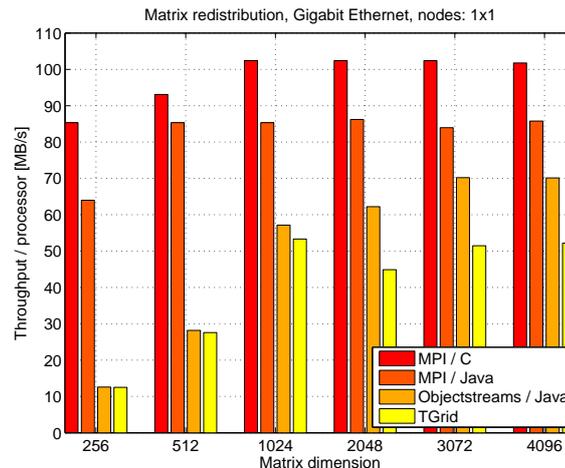
Die Abb. 4.14(a) verdeutlicht den Durchsatz pro Prozessor („Throughput / processor“ in MB/s, der für verschiedene Matrixgrößen und unterschiedliche Umverteilungskonfigurationen mit dem Gigabit-Ethernet-Netzwerk erzielt werden konnte. Der Bezeichner „ $m \times n$ “ gibt die Anzahl der Prozessoren der Quelltask ( $m$ ) und der Zieltask ( $n$ ) an. Beispielsweise bedeutet „ $2 \times 2$ “, dass eine Matrixumverteilung zwischen zwei Prozessoren von T1 und zwei Prozessoren von T2 stattfand. In den Tests wurde jeweils ein TGrid-Prozess auf ge-

nau einen SMP-Knoten abgebildet, d. h. jedem Prozessor stand die gesamte Bandbreite der Netzwerkkarte des SMP-Knotens zur Verfügung.

Wie erwartet, steigt der Durchsatz bei größeren Matrizen an, denn die Nachrichtengröße hängt von der Anzahl der Prozessoren ( $p$ ) ab. Jeder Prozessor speichert  $N^2/p$  Elemente der Gesamtmatrix, wobei  $N$  die Dimension der quadratischen Matrizen bezeichnet. Je mehr Prozessoren eine Matrix speichern, desto geringer ist der Anteil eines einzelnen Prozessors. Aus diesem Grund ist der Durchsatz pro Prozessor für kleinere Matrizen ( $N = 256$ ) bei größeren Prozessorgruppen geringer als bei kleineren Prozessorgruppen. In diesen Fällen ist der Protokoll-Overhead im Vergleich zur Kommunikationszeit noch relativ hoch. Steigt die Größe der Matrizen an, wächst auch der Durchsatz zwischen größeren Prozessorgruppen (z. B. für  $8 \times 8$ ).

Ein weiteres Experiment soll helfen, den Overhead des TGrid-Protokolls in Abhängigkeit von der Anzahl der Prozessoren zu analysieren. Dazu senden wieder alle Prozessoren der M-Task T1 ihren Teil der Matrix zu den Prozessoren von T2. In diesem Test wird jedoch die Nachrichtengröße pro Prozessor fixiert. Das Ziel ist es, dass jeder Prozessor gleich viel Daten sendet, egal wie groß seine zugehörige Prozessorgruppe ist. Aus der Nachrichtengröße pro Prozessor und der Anzahl der Prozessoren kann die Größe der Ausgangsmatrix berechnet werden:  $N = \sqrt{B/8 \cdot P}$ , wobei  $B$  die Anzahl der Bytes bezeichnet und ein Double aus 8 Bytes besteht. Auch für diese Messung wurde nur ein Prozessor pro SMP-Knoten benutzt. Die Resultate dieses Experiments sind in Abb. 4.14(b) veranschaulicht. Der Durchsatz pro Prozessor verringert sich bei kleineren Nachrichten, da der Anteil des Protokolloverheads an der Gesamtlaufzeit steigt. Es lässt sich außerdem feststellen, dass der Durchsatz für größere Nachrichten bei unterschiedlichen Prozessoranzahlen annähernd gleich ist. Demzufolge hat der Protokolloverhead für große Nachrichten keinen gravierenden Einfluss.

Trotzdem ist es noch schwierig, die Güte der Leistung anhand der erzielten Ergebnisse zu beurteilen. Die Messungen wurden mit einem Gigabit-Ethernet-Netzwerk durchgeführt, das eine maximale Bandbreite von 125 MB/s ausweist. In den Tests konnte aber nur ein Durchsatz pro Prozessor von ca. 55 MB/s erreicht werden. Um den Protokoll-Overhead von TGrid genauer zu beziffern, wurde deshalb ein weiteres Experiment durchgeführt. In diesem Ping-Pong-Test werden Daten mit einer vorgegebenen Anzahl an Bytes mit verschiedenen Kommunikationsbibliotheken übertragen. Folgende Konfigurationen wurden getestet: (1) MPI und Programmiersprache C, (2) MPI und Java, (3) Objectstreams und Java und (4) TGrid. Die Auswahl begründet sich aus dem verwendeten Kommunikationsstack von TGrid. Auf der untersten Stufe in der aktuellen Implementierung von TGrid arbeiten MPI Programme. Das Ermitteln des Durchsatzes eines MPI-Programms in C und in Java gibt damit erste Aufschlüsse, wie viel Overhead ein Java-Programm gegenüber einem C-Programm bzgl. der Kommunikationsleistung besitzt. Das MPI-Programm in Java verwendet allerdings primitive Datentypen, die direkt mit der darunterliegenden C-Bibliothek übertragen werden können. Java bietet aber auch die Möglichkeit, beliebige Objekttypen zu übertragen. Dazu müssen die Objekte serialisiert werden. Diese Aufgabe übernimmt der Objectstream, der auch in TGrid zum Einsatz kommt. Deshalb wird auch für diese Kommunikationsvariante der Durchsatz bestimmt. Für alle Tests wurde eine

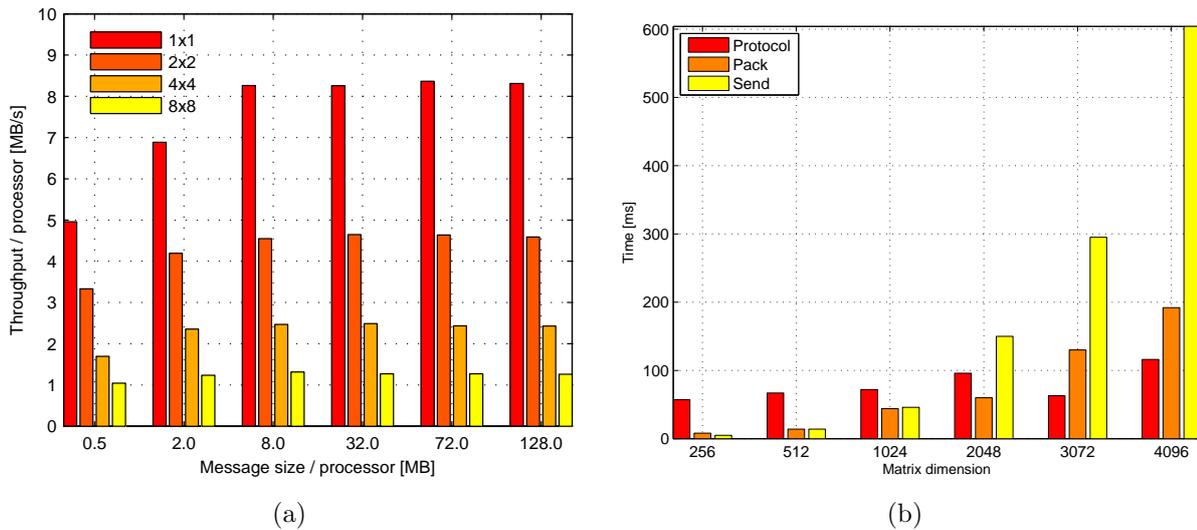


**Abb. 4.15:** Durchsatz bei einer  $1 \times 1$ -Umverteilung mit verschiedenen Programmierschnittstellen.

$1 \times 1$ -Prozessorconfiguration verwendet. Die Ergebnisse dieses Experiments mit den vier verschiedenen Konfigurationen sind in Abb. 4.15 dargestellt. Man kann erkennen, dass der maximale Durchsatz, der einem MPI-Programm in Java zu erreichen ist, bei ungefähr 85 MB/s liegt. Verwendet man zusätzlich noch eine Objekt-Serialisierung, sinkt der Durchsatz auf 70 MB/s. Der Durchsatz des Java-Programms mit Objekt-Serialisierung bildet eine obere Schranke für den mit TGrid erreichbaren Durchsatz. TGrid erzielt in diesem Test ca. 80 % des Durchsatzes des Java-Programms mit Objekt-Serialisierung. Die restliche Zeit entfällt demnach auf das TGrid-Protokoll. Die größte getestete Nachrichtengröße ist 128 MB, womit die Kommunikationszeit relativ gering (ca. 2 s) ist. Deshalb wirkt sich der Protokolloverhead (mehrere 100 ms) hier recht stark auf den Gesamtdurchsatz aus. TGrid soll aber vor allem die Kommunikation zwischen M-Tasks erleichtern. Die maximale Durchsatzrate spielt deshalb bei rechenintensiven Tasks nur eine untergeordnetere Rolle. Der erreichte Durchsatz kann aus den genannten Gründen als akzeptabel angesehen werden. Außerdem könnte man den Overhead der Objektserialisierung für bestimmte Anwendungsfälle reduzieren. Dazu müssten spezielle Serialisierungsfunktionen für einzelne Datentypen von TGrid geschrieben werden. Diese Methode wird z. B. von der Ibis-Middleware [111] verwendet, die ebenfalls in Java geschrieben ist.

### Inter-Subnet-Kommunikation

Wie bereits mehrfach erwähnt, kann TGrid Daten auch zwischen M-Tasks übertragen, die zu zwei entfernten Teilnetze gehören. Dabei stellt sich allerdings das Problem, dass die Prozessoren beider M-Tasks Daten nicht direkt austauschen können, da sie (meist) zu unterschiedlichen privaten IP-Adressräumen gehören. Um dieses Problem zu lösen, können Nachrichten in TGrid über die Subnet-Manager via Routing übertragen werden. Die Nachricht eines Prozessors wird zuerst zum Subnet-Manager übermittelt. Dieser leitet die Nachrichten dann zum entsprechenden Subnet-Manager des Zielnetzes weiter. Im Experiment soll der Durchsatz analysiert werden, der mit einer Inter-Subnet-Kommunikation



**Abb. 4.16:** (a) Durchsatz der Inter-Subnet-Umverteilung über Fast-Ethernet, (b) Protokoll-Overhead pro Umverteilung (4×4), Gigabit-Ethernet.

zu erreichen ist. Um einen möglichst störungsfreien Testablauf zu gewährleisten, wird eine Inter-Subnet-Kommunikation mit einem einzigen dedizierten Cluster simuliert. Dazu wird die Menge der Clusterknoten in zwei unterschiedliche Teilnetze aufgeteilt. Jedes dieser Teilnetze wird von einem eigenen Subnet-Manager kontrolliert. Um die Bandbreite zwischen den Subnet-Managern zu drosseln, verwenden die Subnet-Manager eine Verbindung über Fast-Ethernet (100 MBit/s). Die einzelnen Prozessoren jedes Teilnetzes sind über einen Gigabit-Switch mit dem zuständigen Subnet-Manager verbunden. Die Abb. 4.16(a) vergleicht den erzielten Durchsatz pro Prozessor für unterschiedliche Nachrichtengrößen. Der gemessene Leistungsunterschied zwischen der Umverteilung innerhalb eines Teilnetzes und der Umverteilung zwischen zwei Teilnetzen ist nicht überraschend. Die erreichbare Bandbreite zwischen Prozessoren zweier Teilnetze ist durch die Fast-Ethernet-Verbindung zwischen den Subnet-Manager auf 12.5 MB/s limitiert. Zusätzlich müssen die Nachrichten von einem Prozessor zum Subnet-Manager und vom anderen Subnet-Manager zum Zielprozessor übertragen werden. Der erreichte Durchsatz von ca. 8 MB/s bei einer 1-Processor-Umverteilung ist demnach akzeptabel, bedenkt man den Overhead der drei Hops zum Zielprozessor, den Overhead der Kommunikation mit Java und den Overhead der Objektserialisierung. Die Objektserialisierung wird in der aktuellen Implementierung sogar drei Mal durchgeführt (für jeden Hop einmal). Der Durchsatz für die anderen Prozessorkonfigurationen leitet sich entsprechend ab. Erreicht eine 1×1-Umverteilung einen Durchsatz von 8 MB/s, ist dies die maximale Bandbreite für größere Konfigurationen. Jeder Prozessor der größeren Prozessorkonfigurationen erhält denselben Anteil am Gesamtdurchsatz. Deshalb ist der zu erwartende und auch gemessene Durchsatz bei 2×2: 4 MB/s, bei 4×4: 2 MB/s und bei 8×8: 1 MB/s.

Das Verbindungsprotokoll von TGrid weist deshalb auch noch Optimierungsmöglichkeiten auf. In der aktuellen Implementierung werden Nachrichten erst vom Subnet-Manager

weitergeschickt, wenn dieser sie vollständig empfangen hat. Das kann einerseits zu Speichereingpässen auf dem Subnet-Manager führen, andererseits wird somit die Latenz erhöht und die mögliche Bandbreite reduziert. Eine mögliche Optimierungsmöglichkeit wäre die Realisierung eines Kommunikationsprotokolls, welches in die bestehende Struktur integriert werden kann. Dieses Kommunikationsprotokoll sollte Daten paketweise übertragen und Java-Objekte erst am Zielprozessor wieder deserialisieren.

In einer abschließenden Analyse wurden die partiellen Zeiten der einzelnen Phasen einer Datenumverteilung experimentell bestimmt. Die Abb. 4.16(b) stellt die partiellen Zeiten für unterschiedliche Matrixgröße gegenüber. Der Balken „Protocol“ bezeichnet die Zeit, den die Abarbeitung des Umverteilungsprotokolls in Anspruch nimmt. Dazu gehören u. a. das Zusammenstellen des Nachrichtenplans und das Starten der Umverteilungsmanager. Die Bezeichner „Pack“ und „Send“ charakterisieren die Zeit zum Packen der Nachrichten und die Zeit zum Übertragen der Nachrichten. Das Packen setzt sich aus der Bereitstellung eines Puffers, dem Kopieren der Elemente und dem Schreiben von Header-Informationen zusammen. Im Diagramm lässt sich erkennen, dass der Protokoll-Overhead für alle Matrixgrößen nahezu konstant bleibt. Deshalb hat der Protokoll-Overhead für große Matrizen weniger Einfluss auf den erreichbaren Durchsatz. Wie erwartet, steigt die Zeit zum Packen und Senden der Nachrichten bei größeren Matrizen an.

## 4.10 Zusammenfassung und Fazit

In diesem Kapitel wurde die Laufzeitumgebung TGrid vorgestellt. TGrid ist eine Grid-Middleware, die einzelne Rechnergruppen oder Cluster über das Internet miteinander verbindet. Das Hauptanwendungsgebiet ist die Verbindung von Clustern zu einem Multiclustern. Damit durch Firewall geschützte Cluster-Systeme angebunden werden können, unterstützt TGrid verschiedene Verbindungsprotokolle, wie z. B. das Secure-Shell-Protokoll (ssh). Dadurch entsteht eine gemeinsam nutzbare Grid-Infrastruktur. Da die TGrid-Middleware über die Lage der einzelnen Prozessoren und Prozessorgruppen (Cluster) im gesamten Grid Kenntnis hat (*location-awareness*), kann sie taskbasierte Programme im Grid gut verteilen, indem Tasks an Prozessoren des gleichen Teilnetzes zugewiesen werden.

TGrid ist ein Rechengrid (computational grid), welches die *Co-Allokation* von *Ressourcen* [30] bei der Ausführung eines einzigen gemischt-parallelen Programms *unterstützt*. Das bedeutet, dass ein paralleles Programm und dessen zugehörige Teile verteilt über ein Rechengrid abgearbeitet werden können.

Das primäre Ziel beim Entwurf von TGrid bestand darin, die kumulierte Rechenleistung von Clustern nutzbar zu machen, obwohl diese geografisch verteilt zu verschiedenen Instituten gehören. Damit eine möglichst hohe Geschwindigkeit bei der Co-Allokation von Tasks einer Anwendung erzielt werden kann, beschränkt sich die Abbildung einer Task auf ein Teilnetz. Das ermöglicht es Tasks, die speziellen Eigenschaften der Teilnetze (Cluster) optimal auszunutzen, wie z. B. das verfügbare Netzwerk. Überdies wird dabei auch sichergestellt, dass zwischen den Prozessoren, die einer Task zugewiesen wurden, eine hohe Bandbreite und eine kleine Latenzzeit verfügbar ist.

TGrid wurde komplett in Java geschrieben, um eine möglichst hohe Plattformunabhängigkeit der Middleware zu gewährleisten. Darüber hinaus bietet die TGrid-Laufzeitumgebung eine Java-Schnittstelle an, so dass es sich bei TGrid-Anwendungen um Java-Programme handelt. Dies ist notwendig, um die einzelnen Tasks auf einem beliebigen angeschlossenen Teilnetz ausführen zu können, ohne sie speziell übersetzen zu müssen. Für zukünftige Versionen von TGrid ist es auch denkbar, dass Schnittstellen für andere interpretierte Sprachen angeboten werden (z. B. Python). Außerdem könnten weitere JVM-basierte Programmiersprachen mit wenigen Anpassung verwendet werden, z. B. Groovy oder Jython. Dazu müsste das Executor-Interface von TGrid so erweitert werden, dass die Interpreter-Bibliotheken (für die entsprechende Sprache) beim Ausführen von Tasks mit auf die Client-Rechner übertragen werden.

Die Abarbeitung eines TGrid-Programms kann mit Hilfe eines Taskgraphen modelliert werden. Der Taskgraph ist dabei direkt im TGrid-Programm definiert, d. h. Datenabhängigkeiten zwischen den Tasks werden durch API-Aufrufen spezifiziert. Zur Auflösung der Datenabhängigkeiten besitzt TGrid eine Umverteilungskomponente (MxN), die nach Spezifikation der Eingangs- und Ausgabedatenstrukturen den Transfer zwischen Quell- und Zieltask automatisch vollzieht. Damit wird dem Programmierer die Entwicklung von taskbasierten Anwendungen deutlich erleichtert.

Die experimentellen Analysen der Implementierung der Matrixmultiplikation und der Berechnung eines Mandelbrot-Fraktals haben gezeigt, dass TGrid für gemischt-parallele taskbasierte Anwendungen gut geeignet ist. Die Tests bestätigten aber auch, dass speziell bei der Kommunikationsschicht von TGrid noch unausgeschöpftes Potenzial liegt. Mit einer weiteren Steigerung der Durchsatzraten, vor allem bei Intra-Subnet-Kommunikationen, bietet TGrid sehr gute Voraussetzungen für performante Berechnungen im Grid.

Die Erzeugung der Taskgraphen mit TGrid-Programmen ist dynamisch, da Tasks, in Abhängigkeit der internen Zustände, neue Tasks erzeugen können. Da jede Task mehr als eine Task erzeugen kann, variiert die Anzahl der ausführbereiten Tasks während der Ausführung des Programms. Der TGrid-Scheduler muss entscheiden, in welcher Reihenfolge die ausführbereiten Task auf die Teilnetze abgebildet und viele Prozessoren jeder Task zugewiesen werden. Der verwendete Scheduling-Algorithmus spielt für die Ausführungszeit eines TGrid-Programms eine entscheidende Rolle. Bei der Analyse des TGrid-Systems in diesem Kapitel wurden nur statische Ablaufpläne (*schedules*) verwendet, d. h. es wurde vorher festgelegt, welche Task von welcher Prozessorgruppe auszuführen ist. Für einen produktiven Einsatz als Rechengrid ist dies aber nicht ausreichend, da es die Aufgabe des Schedulers ist, aus der Information über den Systemzustand, den Ablaufplan für die ausführbaren Tasks zur Laufzeit zu generieren. Aus diesem Grund werden im nächsten Kapitel verschiedene Scheduling-Algorithmen untersucht, um eine effiziente und dynamische Ausführung eines TGrid-Programms zu ermöglichen.



## Kapitel 5

# Scheduling von Multiprozessor-Tasks in verteilten Clusterumgebungen

*Good order is the foundation of all great things.*

---

EDMUND BURKE (1729-1797), BRITISH POLITICAL WRITER

Das *Scheduling* (Zeitplanerstellung) spielt in den Bereichen der Informatik eine wichtige Rolle, in denen eine beschränkte Anzahl an Ressourcen konkurrierenden Prozessen zugewiesen werden muss. Beispiele sind das Prozess-Scheduling der Multitasking-Betriebssysteme oder der I/O-Scheduler der Festplatte. Im Bereich des parallelen Rechnens gibt es viele Einsatzgebiete von Scheduling-Algorithmen. Bei der Programmierung von Parallelrechnern mit verteiltem Speicher (SMP, Multicore) werden oft taskbasierte Lösungsverfahren eingesetzt, bei denen der Scheduler über die Zuweisung einer Task zu einer CPU entscheidet. Demzufolge sind gute Algorithmen zum Erstellen eines effizienten *Schedules* (Ablaufplan) für Rechner mit verteiltem Speicher notwendig, um eine gute Laufzeit von taskbasierten Anwendungen zu erzielen.

Wie in den vorherigen Kapiteln vorgestellt, lassen sich gemischt-parallele Anwendungen durch Taskgraphen beschreiben. Die TGrid-Laufzeitumgebung unterstützt die Ausführung von Taskgraphen in einer heterogenen Ansammlung von Rechnern (Grid), wobei jede einzelne Task auf mehreren Prozessoren ausgeführt werden kann (Multiprozessor-Task oder M-Task). Die Laufzeit einer taskbasierten Anwendung in TGrid hängt sehr stark vom verwendeten Scheduling-Algorithmus ab. Der Scheduling-Algorithmus muss dabei eine M-Task auf eine Menge von Prozessoren des Grids abbilden. Die Anzahl der Prozessoren einer M-Task wird als *Allokation* bezeichnet. In diesem Kapitel werden verschiedene Algorithmen zum Scheduling von Taskgraphen in Grid-Umgebungen untersucht. Die Algorithmen werden durch Simulation hinsichtlich ihrer Leistungsfähigkeit in unterschiedlichsten Szenarien verglichen.

## 5.1 Motivation und Zielsetzung

Das Scheduling von Ressourcen ist ein wesentlicher Bestandteil bei der Ausführung jeglicher Programme. Scheduling findet in allen Schichten der Ausführung eines Programms statt: Es beginnt bei der Zuteilung von Rechenzeit an Programme, geht über eine Zuweisung von Prozessoren an Tasks des Programms, bis hin zum Umordnen von Instruktionen auf Prozessorebene. Um die Kosten bei der Ausführung von Programmen zu optimieren (kurze Laufzeit und geringer Ressourcenverbrauch), werden effiziente Scheduling-Algorithmen auf allen Ebenen benötigt.

Auf der Abstraktionsebene der parallelen Systeme entscheiden Scheduler über die Zuweisung von Prozessoren des Systems an ausführbare Tasks. Das gesamte taskbasierte Programm wird durch einen Taskgraphen dargestellt. Die Knoten des Taskgraphen entsprechen den Teilprogrammen und die Kanten definieren die Abhängigkeiten zwischen den Tasks. Bei den Abhängigkeiten handelt es sich entweder um Kontroll- oder um Datenabhängigkeiten. Die Abarbeitung eines Taskgraphen auf einem parallelen System sollte möglichst schnell durchgeführt werden. Sehr häufig bestehen Taskgraphen aus Einprozessor-Tasks, d. h. eine Task wird von genau einem Prozessor ausgeführt (*S-Task*). Eine umfassende Einführung in das Problem des Scheduling für parallele Systeme (inkl. Graphentheorie) und eine große Anzahl von Scheduling-Algorithmen für Taskgraphen aus Einprozessor-Tasks sind in [98] zu finden.

Die betrachteten Applikationen in dieser Arbeit unterscheiden sich dahingehend, dass die Tasks eines Taskgraphen mit einer unterschiedlichen Anzahl von Prozessoren ausgeführt werden können. Diese Anwendungen werden als gemischt-parallele Applikationen (*mixed-parallel applications*) bezeichnet, da datenparallele Tasks gleichzeitig abgearbeitet werden können. Die simultane Ausnutzung von Daten- und Taskparallelität kann gegenüber rein datenparallelen Anwendungen zu einer verbesserten Laufzeit führen [25]. Im Bereich des Scheduling von Tasks wurden mehrere Taskbegriffe eingeführt. Anfang der 1990er Jahre wurde der Begriff der *Multiprozessor-Task* noch sehr generell benutzt. Eine Multiprozessor-Task nach Drozdowski ist eine Task, die mehr als einen Prozessor zur Ausführung benötigt [39]. Später wurden differenziertere Modelle entwickelt, welche die Art der Zuweisung genauer charakterisieren. In letzter Zeit haben sich drei unterschiedliche Taskmodelle herauskristallisiert: *rigid*, *modal* und *malleable*. Bei einer *rigiden* Task ist die Anzahl der Prozessoren fest an die Task geknüpft und nicht variabel. Anders ist es bei den Modellen *modal* und *malleable*. Eine *modal* Task („formbar“) kann von einer beliebigen Anzahl von Prozessoren ausgeführt werden [71]. Die Laufzeit hängt dabei von der Anzahl der zugeteilten Prozessoren ab. Eine *malleable* Task geht noch einen Schritt weiter. Bei diesem Modell kann die Größe (Anzahl der Prozessoren) der Task während der Ausführung dynamisch geändert werden [67]. Damit können Tasks während der Ausführung dynamisch wachsen oder schrumpfen. Ein weiteres Modell ist die teilbare Task (*divisible task*) [15], bei dem eine Task in unabhängige Teile beliebiger Granularität zerlegt werden kann. Diese Teile werden dann parallel auf verteilten Systemen berechnet. Das Modell der *divisible task* ist aber eher vergleichbar mit den Taskgraphen der anderen Modelle, da die „divisible“ Task auch Kontroll- und Datenabhängigkeiten kennt.

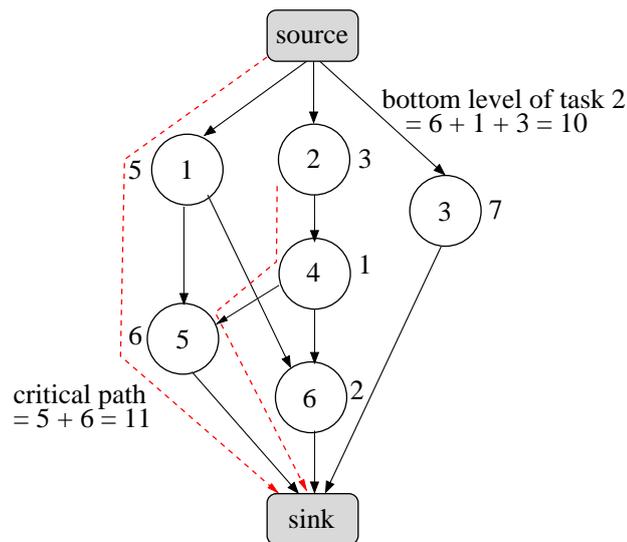
Auch bei der Art der Taskgraphen existieren verschiedene Ansätze z. B. series-parallel, fork-join, harpoon oder bipartite [98]. Viele Applikationen, die durch Taskgraphen definiert werden, können als *series-parallel graphs* (SP-Graphen) beschrieben werden. Diese Graphen besitzen einen definierten Quellknoten (*source*) und einen ausgezeichneten Endknoten (*sink*). Ein größerer Graph entsteht dann durch das Aneinanderhängen (Serialisieren) zweier SP-Graphen oder durch ein Zusammenfügen zweier Graphen durch das Hinzufügen eines neuen Quell- und eines neuen Endknotens. In dieser Arbeit steht die Entwicklung von Scheduling-Algorithmen für TGrid im Vordergrund. TGrid erlaubt auch die Ausführung von Taskgraphen, die nicht „series-parallel“ aufgebaut sein müssen.

TGrid kann Tasks mit einer beliebigen Anzahl an Prozessoren ausführen, aber unterstützt das dynamische Hinzufügen oder Entfernen von Prozessoren einer Task nicht. Man kann also von dem Taskmodell *modalable* sprechen. Allerdings können TGrid-Tasks Ausführungsbedingungen (*constraints*) besitzen, welche die Anzahl der Prozessoren einschränken, z. B. nur Vielfache von zwei. Bei der Entwicklung der Scheduling-Algorithmen in diesem Kapitel werden trotzdem generelle *modalable* Tasks angenommen. Wie schon im Kapitel über TGrid angeklungen, unterstützt TGrid die Abarbeitung von dynamischen Taskgraphen. Diese Graphen entstehen bei taskbasierten Lösungsverfahren, bei denen die Taskerzeugung an bestimmte Bedingungen geknüpft ist und diese Bedingungen iterativ oder rekursiv ausgewertet werden. Ein Beispiel eines solchen mit Bedingungen attribuierten Taskgraphen ist der Graph der Matrixmultiplikation nach Strassen aus Abb. 4.9. Da die Tasks dynamisch sind, können Tasks der TGrid-Anwendungen unterschiedlich viele Kindtasks erzeugen. Das bedeutet aber auch, dass die Graphen vor dem Start des Programms nicht bekannt sind. Die Unterstützung von „dynamischen Taskgraphen“ bedeutet jedoch im Kontext der Arbeit nicht, dass sich Grid-Systeme bei der Abarbeitung dynamisch ändern oder Tasks dynamisch neu platziert werden müssen.

Die meisten in der Literatur vorgeschlagenen Scheduling-Algorithmen für M-Tasks wurden für statische Taskgraphen entworfen. Außerdem konzentriert sich die Mehrheit dieser Algorithmen auf Parallelrechner mit einer homogenen Menge von Prozessoren. Damit gab es bisher kaum Algorithmen, die zum Scheduling von M-Tasks innerhalb von TGrid eingesetzt werden könnten. Aus diesem Grund ist eine weitere Zielstellung dieser Arbeit die Entwicklung von Scheduling-Algorithmen, die dynamische Taskgraphen auf einer heterogenen Menge von Clustern effizient abarbeiten können. Das Vermögen von TGrid, dynamische Taskgraphen abzuarbeiten, geht mit der Fähigkeit zur Abarbeitung von statischen Taskgraphen einher. Deshalb wird auch ein Algorithmus zum Scheduling von statischen M-Taskgraphen in Grid-Umgebungen vorgestellt.

## 5.2 Definition einiger Scheduling-Begriffe

**DAG** Parallele Programme lassen sich als gerichteter azyklischer Graph (DAG)  $G = (V, E)$  darstellen, wobei  $V$  die Menge der Knoten und  $E$  die Menge der Kanten ist. Ein Knoten  $t \in V$  repräsentiert eine Task und besitzt Kosten  $c_t$ , die der Ausführungszeit der Task entsprechen. Eine Kante  $e_{ij}$  von Knoten  $t_i$  zu Knoten  $t_j$  bezeichnet eine Datenabhängigkeit zwischen den Knoten. Diese Kante besitzt ein Gewicht  $d_{ij}$ , das die Kommunikationskosten zwischen den Knoten angibt.



**Abb. 5.1:** Beispiel für die Bestimmung des kritischen Pfades und des Bottom-Levels.

Ein häufig verwendetes Maß zur Priorisierung von Knoten eines Graphen sind die Level. Die Level geben verschiedene Pfadlängen des Graphen an.

**Bottom-Level** Das *Bottom-Level* eines Knotens  $bl(t_i)$  bezeichnet die *Länge des längsten Pfades* vom Knoten  $t_i$  zum Endknoten. Bei DAG-Scheduling-Algorithmen berechnet sich das Bottom-Level einer Task  $t_i$  durch Summierung der Berechnungskosten und Kommunikationskosten auf den Pfaden von Task  $t_i$  zum Endknoten. Der Pfad mit den größten Kosten entspricht dem Bottom-Level der Task  $t_i$ .

**Kritischer Pfad** Der kritische Pfad bezeichnet den längsten Berechnungspfad des Graphen zwischen Start- und Endtask. Dieser Pfad ist somit eine untere Schranke für die Ausführungszeit (Makespan) des gesamten Graphen.

Die Abb. 5.1 zeigt die Berechnung des kritischen Pfades und der Bottom-Level an einem Beispiel. Die Nummern neben den Knoten definieren die Berechnungskosten eines jeden Knotens. Es wurde aus Gründen der Übersichtlichkeit auf Kantengewichte verzichtet. Der kritische Pfad verläuft durch die Knoten 1 und 5, da die Gesamtausführungszeit auf diesem Pfad mit 11 ein Maximum bildet. Das Bottom-Level von Knoten 2 ist der längste Pfad von 2 zur Senke (sink), demnach der Pfad  $2 \rightarrow 4 \rightarrow 5 \rightarrow \text{sink}$  mit Kosten 10.

**Makespan** Der Makespan ist die Zeit, die zum Ausführen des gesamten DAGs benötigt wird. Der Makespan ist mindestens so lang wie der kritische Pfad. Im Normalfall wird die Minimierung des Makespans als Hauptziel beim Scheduling betrachtet.

---

**Algorithmus 5.1** HEFT

---

- 1: set weights of tasks and edges
  - 2: compute rank (bottom-level) of nodes
  - 3: sort list of tasks by decreasing rank
  - 4: **while** list of tasks is not empty **do**
  - 5:   select the first task  $t$  of the list
  - 6:   select the processor  $p$  that has the earliest finish time for the task  $t$
  - 7:   schedule task  $t$  on processor  $p$
  - 8:   remove  $t$  from list
- 

## 5.3 Verwandte Arbeiten

Scheduling-Algorithmen zur Ausführung von Taskgraphen auf parallelen Systemen waren schon in der Vergangenheit Gegenstand intensiver Forschung. Diese Bemühungen lassen sich abhängig von den gemachten Annahmen und Anwendungsfällen unterschiedlich kategorisieren.

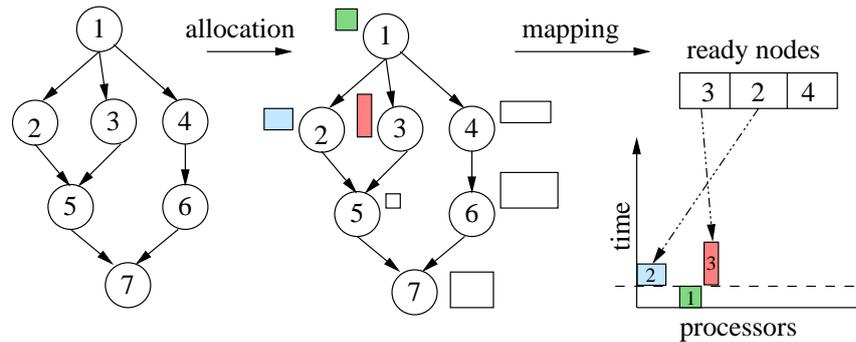
Zahlreiche Algorithmen zur Abarbeitung von Einprozessor-Taskgraphen auf homogenen parallelen Systemen werden in [98] vorgestellt.

Mit Hilfe des HEFT-Algorithmus (Heterogeneous Earliest Finish Time) lassen sich Taskgraphen auf heterogene Systeme abbilden [107]. Dabei handelt es sich um einen List-Scheduling-Algorithmus, der Einprozessor-Tasks voraussetzt. In einem typischen List-Scheduling-Ansatz sind die Tasks und die Kanten der Graphen gewichtet. Damit lassen sich diverse Eigenschaften eines Graphen berechnen, z. B. die Bottom-Level oder der kritische Pfad. Mit Hilfe einer solchen Metrik werden die ausführbaren Tasks sortiert und nacheinander abgearbeitet. Die Funktionsweise von HEFT ist in Alg. 5.1 veranschaulicht. Zuerst werden die Bottom-Level der einzelnen Knoten des Graphen bestimmt und danach anhand dieses Bottom-Levels absteigend sortiert. Somit steht die kritischste Task am Anfang der Liste. Die Tasks werden nacheinander den verfügbaren Prozessoren zugeteilt. Dazu wird immer der freie Prozessor für eine Task gewählt, welcher die Beendigungszeit der Task minimiert (EFT). Eine ausführliche Zusammenfassung verschiedener Algorithmen und einige Ansätze zum Scheduling von Einprozessor-Tasks auf *Heterogeneous Distributed Computing Systems* (HDCCS) sind in [97] zu finden. Andere aktuelle Arbeiten betrachten den Fall, in dem mehrere DAGs zur gleichen Zeit auf einem parallelen System abgearbeitet werden sollen. In [117] wird beispielsweise ein Algorithmus vorgestellt, der HEFT für das Scheduling von mehreren DAGs verwendet. Um HEFT auf mehrere DAGs anzuwenden, werden die einzelnen Taskgraphen zu einem einzigen DAG zusammengefügt.

Auf homogenen parallelen Plattformen fallen Scheduling-Algorithmen von M-Taskgraphen häufig in die Kategorie *one-step*- oder *two-step*-Algorithmus (Ein- oder Zweischrittverfahren). Bei vielen *one-step*-Algorithmen werden die Allokationen der Tasks direkt beim Abarbeiten der Liste der ausführbaren Tasks bestimmt (*list scheduling*). Viele Zweischrittverfahren berechnen im ersten Schritt aus dem statisch gegebenen Taskgraphen und den Gewichten der Knoten und Kanten die Allokation jeder Task. Damit steht die Anzahl der Prozessoren für jede Task vor der zweiten Phase, der *Mapping*-Phase, fest. Im Gegensatz dazu haben Einschrittverfahren deshalb oft eine größere Flexibilität, da

**Algorithmus 5.2** CPA - allocation

- 
- 1: set  $N_p(t) = 1$  for all tasks /\* starting with a one-processor allocation \*/
  - 2: **while**  $T_{CP} > T_A$  **do**
  - 3:    $t \leftarrow$  task from critical path such that  $N_p(t) < p$  and  $\left(\frac{T(t, N_p(t))}{N_p(t)} - \frac{T(t, N_p(t)+1)}{N_p(t)+1}\right)$  is max.
  - 4:    $N_p(t) \leftarrow N_p(t) + 1$  /\* increase allocation of  $t$  by one processor \*/
  - 5:   recompute bottom levels and critical path
- 



**Abb. 5.2:** Beispiel eines Zweischritt-Algorithmus wie CPA. Allokations- und Mapping-Phase.

beim Platzieren (Mapping) einer Task mehr Informationen verfügbar sind, wodurch eine Allokation dynamisch an die aktuellen Systemeigenschaften angepasst werden kann.

Ein bekannter Einschnitt-Algorithmus zum Scheduling von Taskgraphen auf homogenen Systemen ist CPR (Critical Path Reduction) [92]. CPR beginnt mit einer Einprozessor-Allokation für jede Task und ändert iterativ die Allokationen der Tasks. Es wird dabei in jedem Iterationsschritt die M-Task vergrößert (ihre Allokation erweitert), die am meisten durch den zusätzlichen Prozessor gewinnen würde. Dieser Prozess wird solange fortgeführt, bis der Makespan nicht mehr verbessert werden kann. Es handelt sich hierbei um eine Heuristik, die den Schedule zur Compile-Zeit generiert. Der Nachteil dieses Verfahrens ist die hohe Zeitkomplexität. Aus diesem Grund haben Rădulescu und van Gemund (Autoren von CPR) einen weiteren Compile-Zeit-Algorithmus namens CPA (Critical Path and Allocation) entworfen, um die Zeitkomplexität zu verbessern [93]. CPA verwendet dazu ein Zweischrittverfahren. Im ersten Schritt wird die Allokationsgröße für jede Task durch einen Greedy-Algorithmus bestimmt. Die zweite Phase besteht aus einem List-Scheduling, welches die ausführbaren Knoten nach Prioritäten (*earliest starting time* oder *bottom level*) sortiert und danach die Allokationen abbildet. Die Allokationsphase von CPA wird in Alg. 5.2 verdeutlicht. Zu Beginn starten alle Tasks mit einer Einprozessor-Allokation. Danach wird immer der Task des kritischen Pfades ein Prozessor hinzugefügt, die am meisten durch den zusätzlichen Prozessor profitiert. Dieser Prozess wird fortgesetzt, bis ein guter Kompromiss zwischen der Länge des kritischen Pfades  $T_{CP}$  und der durchschnittlichen Prozessornutzung aller Tasks  $T_A$  erreicht ist ( $T_A = 1/p \sum_t (T(t, N_p(t)) \cdot N_p(t))$ ,  $p$  ist die Anzahl der Prozessoren,  $N_p(t)$  ist die Allokation der Task  $t$ ,  $T(t, N_p(t))$  ist die Laufzeit von  $t$  mit der Allokation  $N_p(t)$ ). Die Abb. 5.2 illustriert die beiden Phasen eines Zweischrittverfahrens, wie sie z. B. von CPA genutzt werden.

Ältere Ansätze zum Scheduling von statischen Taskgraphen, welche auch in die Entwicklung von CPA und CPR einfließen, sind TSAS und TwoL. Mit TSAS entwickelten Ramaswamy et al. einen Zweischnitt-Algorithmus, um gemischt-parallele Applikationen auf Parallelrechnern auszuführen [86]. Im ersten Schritt wird mittels konvexer Programmierung die Prozessor-Allokation ermittelt. Danach kommt der übliche List-Scheduling-Algorithmus mit Prioritätsliste zum Einsatz.

Im Rahmen der Definition und Entwicklung des TwoL-Programmiermodells wurde ein Scheduling-Algorithmus für M-Tasks vorgestellt [87], welcher meist einfach als TwoL-Algorithmus bezeichnet wird. Der TwoL-Algorithmus betrachtet als Eingabe einen SP-Graphen (siehe oben) aus M-Tasks, normalisiert zuerst den Taskgraphen und teilt die Tasks des Graphen so in unterschiedliche Schichten (*layers*) ein, dass die Tasks einer Schicht unabhängig voneinander sind. Danach wird für jede Schicht ein Schedule berechnet. Dazu werden die Prozessoren in gleich große Gruppen aufgeteilt. Man startet mit einer Gruppengröße  $g = p$ , wobei  $p$  die Anzahl der Prozessoren angibt. Jede Prozessorgruppe entspricht dann einer möglichen Allokation einer Task. Mit den gegebenen Prozessorgruppen wird der Schedule für die Tasks einer Schicht berechnet. Im Anschluss wird eine neue Gruppengröße  $g = p/i$ ,  $i = 1, \dots, p$ , gewählt und der Schedule für diese Allokationsgröße berechnet. Das Verfahren wird bis zu einer Allokationsgröße von 1 und dementsprechend  $p$  Prozessorgruppen fortgesetzt. Der TwoL-Algorithmus verwendet dann die Gruppengröße und den Schedule, der für die aktuelle Tasksschicht den geringsten Makespan erzielt.

Da CPA nicht auf heterogene Plattformen anwendbar ist und HEFT nicht für Multiprozessor-Tasks entworfen wurde, erweiterten Suter et al. beide Algorithmen zum Scheduling von M-Taskgraphen auf heterogenen Systemen. Die resultierenden Algorithmen – HCPA [75] und MHEFT [23] – sind Compile-Zeit-Algorithmen für statische Taskgraphen. Beide verwenden das gleiche Plattformmodell und berücksichtigen während der Scheduling-Phase die Kommunikationskosten unter Anwendung eines Modells zur Datenumverteilung. Die betrachtete Plattform setzt sich aus einer heterogenen Menge von homogenen Clustern zusammen (Multicluster), wobei die Cluster über ein gemeinsames Backbone verbunden sind. Für beide Algorithmen gilt, dass Tasks nur innerhalb eines homogenen Clusters ausgeführt werden dürfen, also Prozessoren nicht über Cluster Grenzen hinweg zugeordnet werden. Wie auch beim Entwurf von TGrid, wird für diese Annahme davon ausgegangen, dass eine Inter-Cluster-Kommunikation einen stark negativen Einfluss auf die Leistungsfähigkeit einer M-Task haben würde.

Der Algorithmus MHEFT (*Mixed Parallel Heterogenous Earliest Finish Time*) benutzt den List-Scheduling-Algorithmus HEFT als Ausgangspunkt. Das Ziel von MHEFT ist, HEFT so zu erweitern, dass ein Scheduling von datenparallelen Tasks auf Multiclustern möglich ist. Zur Bestimmung der durchschnittlichen Laufzeit einer Task werden wie bei HEFT die Laufzeiten der Task für alle Einprozessor-Allokationen der Cluster betrachtet. Es gibt auch eine zweite mögliche Methode zur Bestimmung der durchschnittlichen Laufzeit. In dieser werden neben den Einprozessor-Allokationen auch die möglichen Mehrprozessor-Allokationen der Cluster in die Berechnung einbezogen. Nachdem die durchschnittlichen Laufzeiten ermittelt sind, lassen sich die Bottom-Level bestimmen. Nach der Priorisierung der Tasks und der anschließenden Sortierung werden die Tasks auf die Pro-

zessoren abgebildet. Dazu wird die Menge von Prozessoren für eine Task gewählt, die die Laufzeit minimiert, wobei die Umverteilungskosten einbezogen werden.

Da auch MHEFT eine relative hohe Zeitkomplexität besitzt und vor allem viele Ressourcen verwendet, wurde mit HCPA ein Algorithmus entwickelt, der wie CPA mit weniger Aufwand eine relativ gute Effizienz erreichen soll. Wie CPA arbeitet HCPA in zwei Phasen: In der ersten Phase werden die Allokationen aller Tasks bestimmt und in der zweiten werden die Tasks auf die Prozessoren abgebildet. Um die Allokationen zu bestimmen, wird für HCPA ein virtueller homogener Referenzcluster geschaffen. Die Prozessoren dieses Clusters entsprechen dem langsamsten Prozessor der Originalplattform. Mit Hilfe des Referenzclusters lässt sich der Allokationsalgorithmus von CPA wiederverwenden. Die Größe einer Allokation wird dann je nach ausgewähltem Zielcluster durch Anwendung eines Performance-Modells (z. B. *Amdahl's law*) angepasst. Auch bei HCPA werden anschließend die Tasks entsprechend ihrer Bottom-Levels geordnet und nacheinander auf die einzelnen Cluster abgebildet. Da in dieser Arbeit HCPA und MHEFT als Vergleichsalgorithmen zu den beschriebenen Verfahren dienen, wird in den folgenden Abschnitten nochmals genauer auf Details beider Algorithmen eingegangen.

Die originalen Varianten von MHEFT und HCPA erzielten nicht in allen Szenarien sehr gute Ergebnisse. Deshalb wurden in [76] diverse Optimierungen beider Algorithmen vorgestellt und verglichen. Für MHEFT wurde z. B. der großzügige Umgang mit Ressourcen beschränkt, in dem die Allokationsgröße der Tasks durch verschiedene Heuristiken limitiert wurde. Eine zu große Ausdehnung der Allokationen war auch bei HCPA ein Problem, wenn der virtuelle Referenzcluster aus sehr vielen Prozessoren bestand. Mit der optimierten Variante von HCPA wird das Anwachsen eingeschränkt. Außerdem wird während der Mapping-Phase versucht, auftretende „Löcher“ im Ablaufplan auszunutzen, d. h. es wird ggf. eine kleinere Allokation von Prozessoren verwendet, wenn diese Allokation zu einem früheren Zeitpunkt verfügbar ist und zu einer kürzeren Fertigstellungszeit der Task führt.

Die List-Scheduling-Algorithmen arbeiten die Tasks der Prioritätsliste nacheinander ab. Zur Bestimmung der Priorität und damit zur Sortierung wird das Bottom-Level der Tasks verwendet. Die serielle Abarbeitung hat den Nachteil, dass Tasks mit ähnlicher Priorität, beim Scheduling nicht gleichzeitig betrachtet werden. Das führt oft zu einer geringeren Ausnutzung der möglichen Taskparallelität, da den zuerst betrachteten Tasks oft sehr große Allokationen zugeordnet werden. Um dieses Problem zu lösen, wird von Suter das Verfahren  $\Delta$ -CTS ( *$\Delta$ -critical task scheduling*) vorgeschlagen [104], welches die Allokationsgröße von Tasks mit ähnlicher Priorität beschränkt, so dass nach Möglichkeit noch andere  $\Delta$ -kritische Tasks im System ausgeführt werden können.

Das Scheduling von DAGs ist auch für die Entwicklung von Grid-Workflows von großer Bedeutung. Das verteilte Abarbeiten eines Workflows im Grid hat ähnliche Anforderungen wie High-Performance-Computing im Grid, denn auch hier wird eine schnelle Abarbeitung eines DAGs (Workflows) gewünscht. Für das Scheduling von Workflows speziell im Grid sind dynamische Parameter einzubeziehen, z. B. die sich dynamisch ändernde Grid-Infrastruktur oder das dynamische Erzeugen von Sub-Workflows. Workflows lassen sich damit eher als dynamische Taskgraphen charakterisieren, denn sie können neben Schleifen auch Rekursionen enthalten. Prodan und Fahringer entwickelten einen hybriden Scheduling-Ansatz, um Workflows in dynamischen Grid-Umgebungen (Prozessoren, Netzwerkverbin-

dungen) auszuführen [83]. Damit die dynamischen Graphen mit einem statischen Scheduling-Algorithmus verarbeitet werden können, müssen die Zyklen in den Graphen nach bestimmten Regeln eliminiert werden (*cycle elimination*). Außerdem können bei diesem Ansatz Tasks bei Bedarf im Grid migriert werden. Prodan entwarf auch einen weiteren Scheduling-Algorithmus zur Ausführung von Workflows im Grid. Dieser Algorithmus verwendet einen HEFT-Ansatz zum Scheduling von Einprozessor-Tasks in der ASKALON-Programmierungsumgebung. Dabei werden hierarchisch-strukturierte Workflows zuerst in einen „flachen“ DAG überführt. Dieser DAG wird danach mittels List-Scheduling auf die Prozessoren abgebildet.

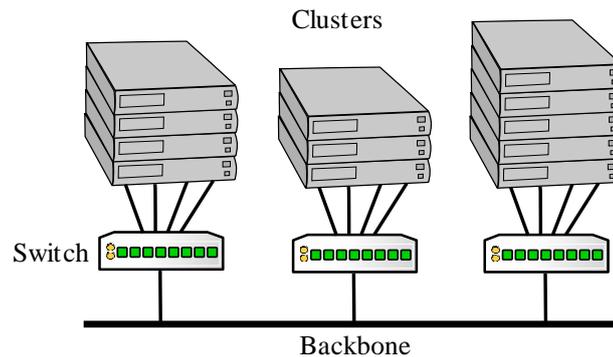
Im Bereich des Grid-Computing sind beim Scheduling neben einem kurzen Makespan viele wirtschaftliche Faktoren von Interesse, z. B. eine gegebene Deadline oder ein Budget-Limit. Buyya et al. entwickelten ein Framework, um beim Ausführen von Grid-Applikationen die wirtschaftlichen Aspekte zu beachten [19]. Innerhalb dieses Kontextes wurden verschiedene Scheduling-Algorithmen vorgestellt, um Workflows auf Grid-Ressourcen abzubilden. Neben dem in [19] dargestellten DBC-Algorithmus (*deadline- and budget constrained*) für Parameter-Sweep-Applikationen wurde auch ein genetischer Ansatz zum Workflow-Scheduling im Grid vorgeschlagen [115].

Ein Überblick über aktuelle Grid-Plattformen und Scheduling-Algorithmen wird von Wicczorek et al. in [113] gegeben. Die Autoren kategorisieren u. a. Taskmodelle (rigid, moldable), die Arten von DAGs oder die Komplexität von Ressourcen (Einprozessorsysteme vs. Cluster). Außerdem werden die Anwendungsfälle der verschiedenen Scheduling-Algorithmen (auch MHEFT und HCPA) verglichen. Nach der Erstellung der Taxonomie wird deutlich, dass ein allgemeiner und alles abdeckender Scheduling-Algorithmus nicht praktikabel und nicht realisierbar scheint.

## 5.4 Scheduling von M-Taskgraphen in Multiclustern

Wie bereits mehrfach in diesem Kapitel angeklungen, konzentriert sich diese Arbeit auf dynamische Taskgraphen im TGrid-Kontext. Als dynamisch wird hierbei die unbekannt finale Struktur des Taskgraphen vor dem Start der Applikation betrachtet und nicht die dynamische Änderung der Grid-Umgebung. Das Problem beim Scheduling von dynamischen Taskgraphen ist die geringe Anzahl an Informationen, die bei der Entscheidungsfindung ausgenutzt werden können. Bei Compile-Zeit-Ansätzen ist der gesamte DAG beim Start des Scheduling-Algorithmus bekannt. Dadurch lassen sich z. B. der kritische Pfad oder diverse Kostenmodelle auf die einzelnen Knoten anwenden. Dies ist bei dynamisch erzeugten Tasks nicht möglich. Als einzige Informationsquelle können nur die bisherigen Scheduling-Entscheidungen der abgearbeiteten Tasks sowie der bisher bekannte Taskgraph herangezogen werden. Ein wichtiges Kriterium bei der Entwicklung von Scheduling-Algorithmen für dynamische Taskgraphen ist die Zeitkomplexität der Verfahren. Da der Scheduler online arbeitet – also immer dann anspringt, wenn Tasks ausführbar sind – muss ein Algorithmus möglichst schnell ein gutes Ergebnis liefern.

Im Folgenden werden die verwendeten Applikations- und Plattformmodelle vorgestellt und formalisiert. Im Anschluss werden mit Hilfe der Modelle zwei Scheduling-Algorithmen (RePA, DMHEFT) für dynamisch erzeugte Taskgraphen entwickelt und mit Compile-Zeit-Verfahren wie MHEFT verglichen.



**Abb. 5.3:** Darstellung des verwendeten Plattformmodells: Heterogener Cluster aus homogenen Clustern, die über ein Backbone verbunden sind.

### 5.4.1 Plattformmodell

Als Plattform zum Ausführen von taskparallelen M-Taskprogrammen werden in dieser Arbeit Cluster aus Clustern (Multicluster) betrachtet. Dabei sind die einzelnen Cluster selbst homogen aufgebaut, d. h. alle Prozessoren eines Clusters haben dieselbe Architektur und Leistung. Die Cluster untereinander können sich aber in diesen Eigenschaften unterscheiden, so dass das Gesamtsystem heterogen ist. Die Betrachtung einer solchen Umgebung ist nicht ungewöhnlich, denn speziell im akademischen Umfeld bilden viele Forschungseinrichtungen Meta-Cluster, die aus den einzelnen verfügbaren Clustern aufgebaut sind. Durch eine virtuelle Bereitstellung eines einzigen Clusters können Wissenschaftler bei der Abarbeitung ihrer Jobs von der gebündelten Leistung profitieren [81].

Ein Beispiel einer solchen Grid-Umgebung, die genau diese Eigenschaften erfüllt ist Grid'5000<sup>1</sup> [16]. Das gleichnamige französische Forschungsprojekt hat das Ziel, eine konfigurierbare, kontrollierbare und überwachbare Grid-Plattform anzubieten. Diese Plattform besteht aus 9 geografisch verteilten Clustern mit einer Gesamtzahl von 5000 Prozessoren (AMD Opteron, Intel Xeon, Intel Itanium 2, PowerPC).

Zusammenfassend wird zum Ausführen von gemischt-parallelen Programmen eine Plattform betrachtet, die aus  $c$  unterschiedlichen Clustern besteht, wobei jeder Cluster  $C_k$ ,  $k = 1, \dots, c$  aus  $p_k$  identischen Prozessoren aufgebaut ist. Die Prozessoren des Clusters  $C_k$  operieren mit einer Geschwindigkeit von  $s_k$  (Operationen pro Sekunde). Jeder Cluster kann unterschiedliche Netzwerktopologien zum Verbinden der Prozessoren einsetzen (z. B. diverse Switches). Die verschiedenen Cluster sind alle an ein leistungsstarkes Backbone angebunden, wobei jeder Cluster durch eine einzelne Netzwerkverbindung mit dem Backbone verbunden ist. Die Charakteristika (Bandbreite, Latenz) der Links von den Clustern zum Backbone können variieren. Kommunikationsoperationen zwischen Clustern werden nicht serialisiert, d. h. sie finden gleichzeitig statt und können Netzwerk-Contention (hohe Netzlast) auf den Links der Cluster oder auf dem Backbone verursachen. Die Abb. 5.3 illustriert das hier beschriebene Plattformmodell.

<sup>1</sup><http://www.grid5000.fr>

## 5.4.2 Applikationsmodell für gemischt-parallele Applikationen

Gemischt-parallele (taskparallele) Applikationen können durch einen gerichteten azyklischen Graphen (DAG, directed acyclic graph)  $\mathcal{G} = (\mathcal{N}, \mathcal{E})$  beschrieben werden, wobei die Menge der Knoten  $\mathcal{N} = \{t_i | i = 1, \dots, N\}$  die datenparallelen Tasks repräsentiert. Die Menge  $\mathcal{E} = \{e_{ij} | (i, j) \in \{1, \dots, N\} \times \{1, \dots, N\}\}$  bezeichnet die Menge der Kanten, welche die Datenabhängigkeiten zwischen Tasks darstellen. Jede Kante  $e_{ij}$  besitzt ein *Gewicht*  $d_{ij}$ , welches der Anzahl der zwischen Task  $t_i$  und  $t_j$  zu übertragenden Daten (in Bytes) entspricht. Die Task  $t_j$  ist somit ein Nachfolger von  $t_i$ . Es wird darauf hingewiesen, dass bei dem verwendeten Modell neben den reinen Kommunikationskosten auch noch Kosten durch eine mögliche Umverteilung von Daten hinzukommen können. Die Umverteilungskosten werden z. B. dann betrachtet, wenn die Tasks  $t_i$  und  $t_j$  mit einer unterschiedlichen Menge von Prozessoren ausgeführt werden. Da jede datenparallele Task (M-Task) auf einer variierenden Anzahl von Prozessoren ausgeführt werden kann (*modalable*), bezeichnet  $T_k(t, p)$  die Ausführungszeit der Task  $t$  mit  $p$  Prozessoren auf Cluster  $C_k$ . Die Zeit  $T_k(t, p)$  setzt sich aus der Berechnungs- und der Kommunikationszeit zusammen, die zur Ausführung der Task  $t$  auf Cluster  $C_k$  benötigt wird. In der Praxis kann die Zeit  $T_k(t, p)$  durch ein Benchmarking auf allen Clustern für verschiedene Werte von  $p$  empirisch bestimmt werden. Die Laufzeit einer Task kann auch durch Anwendung eines mathematischen Performance-Modells abgeschätzt werden. Die gesamte Ausführungszeit (*makespan*) der Applikation wird als die Zeit zwischen dem Beginn der Ausführung der Starttask (*source*) und dem Beenden der Endtask (*sink*) definiert. Es wird außerdem angenommen, dass Tasks, wenn sie gestartet wurden, nicht unterbrochen werden können (*non-preemptive*).

Nach Angabe des Plattform- und des Applikationsmodells kann das Scheduling-Problem für dynamische Taskgraphen wie folgt definiert werden: Für jede ausführbereite Task soll die Startzeit und die Anzahl der Prozessoren bestimmt werden, so dass die Gesamtausführungszeit minimiert wird. Bedingt wird die Abarbeitungsreihenfolge der Tasks durch die Datenabhängigkeiten, d. h. eine Task kann nicht gestartet werden, bevor die Vorgänger beendet und die Daten aller Vorgänger empfangen wurden.

## 5.5 Der ReP-Algorithmus (RePA)

### 5.5.1 Motivation und Ziel

Ein wichtiges Ziel dieser Arbeit ist die Entwicklung und Implementierung eines kostengünstigen Algorithmus, der mit dem zuvor definierten Applikations- und Plattformmodell gute Scheduling-Resultate in TGrid erzielen kann.

Es wird nochmals daran erinnert, dass dem Scheduling-Algorithmus bei der Erstellung des Ablaufplans nicht der gesamte Taskgraph zur Verfügung steht. Der Algorithmus kennt zu einem Zeitpunkt nur die ausführbereiten Knoten und das Mapping der schon beendeten Tasks. Wurde die Ausführung eines Knotens des gemischt-parallelen Programms beendet, können Datenabhängigkeiten aufgelöst werden, so dass neue Tasks gestartet werden können. Mit diesen Informationen gilt es, eine möglichst gute Entscheidung zum Platzieren

einer Task im Grid zu erzielen. Damit lassen sich drei essenzielle Schritte angeben, die der Scheduling-Algorithmus durchführen muss:

1. Sortieren der ausführbaren Tasks. Damit stellt sich die Frage, welche Metrik zur Sortierung herangezogen werden soll.
2. Selektion eines Clusters. Da jede Task nach Definition nur innerhalb eines Clusters ausgeführt werden kann, muss dieser Zielcluster bestimmt werden.
3. Bestimmung von Allokation und Mapping. Auf dem Zielsystem (oder Zielcluster) müssen die Tasks auf eine Menge von Prozessoren abgebildet werden. Die Schritte (2) und (3) könnten auch vertauscht werden. Wie im Fall von CPA ließe sich auch zuerst eine Allokation bestimmen, die anschließend im Grid abgebildet werden muss.

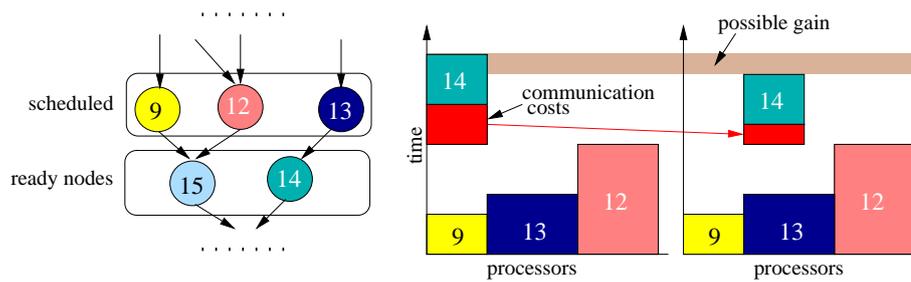
Im folgenden Abschnitt werden der Algorithmus RePA sowie die Implementierung der hier angegebenen Schritte detailliert beschrieben und die jeweiligen Entscheidungen begründet.

### 5.5.2 Beschreibung des Algorithmus

Ein wichtiges Ziel des ReP-Algorithmus (*Reuse Processor Algorithm*) ist die Reduzierung der teuren Kommunikationskosten bei der Datenumverteilung zwischen kooperierenden Tasks. Dabei wird vorausgesetzt, dass bei der Kommunikation innerhalb eines Prozessors (oder Rechners) keine Kosten entstehen. Das bedeutet, dass keine Kommunikationskosten entstehen, wenn zwei aufeinanderfolgende Tasks mit gleicher Datenverteilung der verbindenden Variablen auf dieselbe Menge von Prozessoren abgebildet werden. Daraus folgt auch, dass ein Wiederwenden einer Teilmenge der Prozessoren der Elterntask zu einer kleineren Kommunikationszeit führen kann. Auch wenn das Mapping beider Tasks unterschiedlich ist, sich also die sendende und empfangende Menge an Prozessoren in einem Teil unterscheidet, kann eine Wiederverwendung zu weniger Netzwerk-Contention im Cluster beitragen und damit die Gesamtkommunikationszeit senken.

Die Abb. 5.4 verdeutlicht die ursprüngliche Idee, die die Entwicklung von RePA motiviert hat. Die Abbildung zeigt auf der linken Seite den Teil des gerade betrachteten Teilgraphen. Auf der rechten Seite sind zwei mögliche Schedules abgebildet, die durch eine unterschiedliche Abbildung der Task 14 entstehen können. Im Beispiel wurden die Tasks mit den Nummern 9, 12 und 13 bereits ausgeführt. Damit ist auch die Task 14 ausführbar. Würde Task 14 nun den Prozessoren von Task 9 zugewiesen werden, müssten alle Pakete bei der Datenkommunikation zwischen 13 und 14 über das Clusternetzwerk geschickt werden. Wenn Task 14 aber Prozessoren ihrer Elterntask, z. B. von Task 13, wiederverwenden kann, dann könnte sich der Kommunikationsoverhead reduzieren, da für die Umverteilungen zwischen 13 und 14 nicht alle Pakete über das Netzwerk geschickt werden müssten.

Die Arbeitsweise von RePA wird in Alg. 5.3 beschrieben. Die Zeilen 1–9 werden immer dann ausgeführt, wenn neue Tasks ausführbar sind oder es freie Prozessoren im



**Abb. 5.4:** Idee von RePA: Die Kommunikationszeit kann verbessert werden, wenn Kindtasks den Prozessoren der Elterntasks zugewiesen werden.

System gibt. Damit definieren sie die Grundstruktur des Algorithmus. Zuerst werden die ausführbaren Tasks (*ready nodes*) nach ihren Berechnungskosten sortiert, was in der Literatur speziell bei List-Scheduling-Algorithmen als LPT-Strategie (*Largest Processing Time First*) bekannt ist (z. B. [68], Kapitel 9). Dies ist ein gängiges Verfahren, damit die großen (großflächigen) Tasks zuerst platziert und kleinere Tasks später verwendet werden können, um die Lücken zwischen den großen Tasks zu füllen. Außerdem umgeht man so das Problem, dass Task verhungern könnten (*starvation*), wenn große Allokationen nie abgebildet werden können. Danach werden die Tasks mit einem stabilen Sortieralgorithmus nach steigender Knotentiefe sortiert. Damit wird eine Bottom-Level-Sortierung simuliert, da Knoten, die weit von der Senke entfernt sind (*bottom-up*), auch oft sehr nah an der Wurzel liegen (*top-down*). Das Ziel dieser zweiten Sortierung ist, kritische Knoten bevorzugt auszuführen.

Der Algorithmus versucht danach in drei Hauptschritten, die ausführbaren Tasks auf die  $c$  Cluster abzubilden. Dazu werden die Tasks nacheinander aus der sortierten Liste entnommen und für jede Task der Zielcluster bestimmt (*Schritt 1*). Ist ein Cluster ausgewählt, versucht RePA in *Schritt 2* eine effiziente Größe der Allokation zu ermitteln. Im darauf folgenden *Schritt 3* selektiert RePA eine dieser Allokation entsprechende Anzahl von Prozessoren auf dem gewählten Cluster. Alle drei Schritte basieren auf Heuristiken, die sich in einer Vielzahl von Tests als effektiv gezeigt haben. In Schritt 1 hat sich als effizient erwiesen, den Cluster mit der größten freien Rechenleistung als Zielcluster  $C_j, 1 \leq j \leq c$ , für die aktuelle Task  $t_i, 1 \leq i \leq N$ , zu wählen (Zeile 11). Dies entspricht einer EFT-Strategie (*earliest finishing time*), wobei die Ausführungs- und die Kommunikationszeit der Task nicht berücksichtigt werden. Es wird nur vermutet, dass eine Task auf dem Rechner, der die meiste verfügbare Rechenleistung aufweist, auch am schnellsten ausgeführt wird. Eine brauchbare Abschätzung der Kommunikationszeit ist für eine steigende Anzahl von Clustern relativ teuer, was für einen online arbeitenden Algorithmus stark ins Gewicht fällt. Für eine genaue Abschätzung der Kommunikationszeit muss die gesamte Netzwerkhierarchie zwischen den Prozessoren der Cluster, das Verbindungsprotokoll (z. B. TCP) und die Datenverteilung innerhalb der M-Task berücksichtigt werden. Aus den Datenverteilungen zwischen zwei M-Tasks könnte man dann die Nachrichten bestimmen, die zwischen den Prozessorgruppen gesendet werden müssen. Anschließend muss dann die Zeit für das Senden und Empfangen dieser Nachrichten im hierarchischen Netzwerk bestimmt werden.

**Algorithmus 5.3** RePA

---

```

1: INPUT: queue - ready nodes
2: sort queue by decreasing computation cost
3: sort queue by increasing node depth // 2nd sort, stable sort
4: while queue is not empty and at least one cluster is available do
5:   node = queue.pop()
6:   cluster = find_target_cluster(node)
7:   processor_nb = get_allocation_on_cluster(node, cluster)
8:   processor_list = map_allocation_on_cluster(node, cluster, processor_nb)
9:   add new schedule(node, cluster, processor_list)

function find_target_cluster( node )
11: return cluster with highest computational power available

function get_allocation_on_cluster( node, cluster )
12: if node is root node then
13:   return number of free processors on cluster
14: else
15:   free_processor_nb = number of free processors on cluster
16:   cluster_power_ratio = (free computational power of cluster) / (free computational
    power of all clusters)
17:   node_computation_ratio = (computation cost of node) / (computation cost of unsche-
    duled ready nodes)
18:   return max( (min(1,  $\frac{\text{node\_computation\_ratio}}{\text{cluster\_power\_ratio}}$ ) * free_processor_nb), 1 )
    // at least one processor, at most the whole cluster

function map_allocation_on_cluster(node, cluster, processor_nb)
19: processor_list = empty list
20: append all free processors to processor_list which were assigned to a parent node on
    cluster // try to reuse processors which were assigned to any parent node
21: append all other free processors on cluster to processor_list
22: return first processor_nb processors of processor_list

```

---

Nach der Wahl des Clusters wird in Schritt 2 die Anzahl der Prozessoren für die betrachtete Task so bestimmt (Funktion `get_number_of_processors()`), dass die resultierende Allokation in einem möglichst guten Verhältnis zu den anderen wartenden Tasks steht. Die Allokation sollte nicht zu groß sein, um auch noch andere Tasks ausführen zu können, und nicht zu klein, um die Task entsprechend schnell zu beenden.

Die Allokationsgröße  $\phi_{ij}$  einer Task  $t_i$  auf Cluster  $C_j$  wird wie folgt berechnet: Es sei  $\zeta_k$  die Menge der verfügbaren Prozessoren eines Clusters  $C_k$ :

$$\zeta_k = \{p : p \in C_k \wedge p \text{ is idle}\}. \quad (5.1)$$

Für eine Prozessormenge  $\varphi$  sei die Rechenleistung  $\mathcal{P}(\varphi)$  definiert als

$$\mathcal{P}(\varphi) = \sum_{p \in \varphi} \text{power}(p). \quad (5.2)$$

Die Funktion  $power(p)$  könnte z. B. die Anzahl der FLOPS zurückgeben, die ein Prozessor ausführen kann.

Die Allokation  $\phi_{ij}$  der Task  $t_i$  auf einem Cluster  $C_j$  hat dann eine gute Größe, wenn die verwendete Rechenleistung im selben Verhältnis steht, wie die Berechnungskosten  $c(t_i)$  der Task  $t_i$  zu den Berechnungskosten aller ausführbaren Tasks. Die Rechenleistung der  $\phi_{ij}$  Prozessoren wird deshalb wie folgt gewählt:

$$\mathcal{P}(\phi_{ij}) = \frac{c(t_i)}{\sum_{t_k \text{ is ready}} c(t_k)} \cdot \sum_{1 \leq l \leq c} \mathcal{P}(\zeta_l) \quad (5.3)$$

Die verfügbaren Prozessoren des homogenen Zielclusters  $C_j$  haben jeweils eine Rechenleistung von:

$$\theta_j = \frac{\mathcal{P}(\zeta_j)}{|\zeta_j|}. \quad (5.4)$$

Verwendet man nur die Prozessoren des Cluster  $C_j$ , benötigt man  $\phi_{ij}$  Prozessoren mit Leistung  $\theta_j$ , um die Rechenleistung  $\mathcal{P}(\phi_{ij})$  zu erreichen:

$$\mathcal{P}(\phi_{ij}) = \phi_{ij} \cdot \theta_j. \quad (5.5)$$

Aus den Gleichungen (5.3) und (5.5) ergibt sich die Berechnung der Allokation  $\phi_{ij}$  wie folgt:

$$\begin{aligned} \phi_{ij} \cdot \theta_j &= \frac{c(t_i)}{\sum_{t_k \text{ is ready}} c(t_k)} \cdot \sum_{1 \leq l \leq c} \mathcal{P}(\zeta_l) \\ \phi_{ij} &= \frac{c(t_i)}{\sum_{t_k \text{ is ready}} c(t_k)} \cdot \frac{\sum_{1 \leq l \leq c} \mathcal{P}(\zeta_l)}{\theta_j} \\ &= \underbrace{\frac{c(t_i)}{\sum_{t_k \text{ is ready}} c(t_k)}}_{\text{node\_computation\_ratio}} \cdot \underbrace{\frac{\sum_{1 \leq l \leq c} \mathcal{P}(\zeta_l)}{\mathcal{P}(\zeta_j)}}_{1/\text{cluster\_power\_ratio}} \cdot |\zeta_j|. \end{aligned} \quad (5.6)$$

Aus Gleichung (5.6) folgt damit die Berechnungsvorschrift der Variablen `node_computation_ratio` und `cluster_power_ratio` aus Alg. 5.3 (Zeilen 16–18).

In Schritt 3 wird die Allokation  $\phi_{ij}$  von  $t_i$  auf die Prozessoren von  $C_j$  abgebildet. Erst an dieser Stelle wird versucht, die Prozessoren wiederzuverwenden, die einer Elterntask zugeordnet waren. Es werden solange Prozessoren von Elterntasks zugeteilt, bis die gesamte Allokation abgebildet wurde oder keine Elternprozessoren mehr verfügbar sind. Die Auswahl der Elternprozessoren und ggf. der anderen unbenutzten Prozessoren erfolgt durch die Betrachtung der Prozessornummern (kleinere Nummer zuerst). Es wäre möglich, die Kommunikations- und Umverteilungskosten einzubeziehen, um ein gute Wahl der Menge an Prozessoren zu finden. Diese Methode wird aber im Moment nicht angewendet, da diese Abschätzung teuer ist und die Entwicklung eines Low-Cost-Algorithmus im Vordergrund stand. An dieser Stelle könnten in Zukunft einfache und effiziente Heuristiken betrachtet werden.

Die angewendeten Heuristiken und die Anordnung der Schritte sind vor allem ein Resultat intensiven Testens mit unterschiedlichen Grid-Konfigurationen und verschiedenen

Typen von DAGs. Ein Großteil dieser Arbeit konzentrierte sich dabei auf die Selektion des Zielclusters und die Bestimmung der Allokationsgröße. Es wurde festgestellt, dass die resultierenden Makespans am stärksten variieren, wenn die Strategie zur Bestimmung des Zielclusters geändert wird. Damit kristallisierte sich Schritt 1 als kritischster Punkt für eine schnelle Abarbeitung heraus. Für die Implementierung dieses Schrittes wurden verschiedene Heuristiken getestet, z. B.

- wähle den Cluster mit den meisten freien Prozessoren, die Elterntasks zugeordnet sind, oder
- wähle den Cluster mit den meisten Elterntasks.

Diese Heuristiken zur Bestimmung des Zielclusters sind in irgendeiner Weise an die Elterntask und an den Graph geknüpft. Mit diesen Varianten wurde die Migration von Tasks im Grid stark behindert. Oft wurden Kindtasks auf denselben Cluster abgebildet, obwohl nur sehr wenige Elternprozessoren frei waren, anstatt die Task auf einen anderen wenig benutzten Cluster zu migrieren. Damit konnte zwar eine relative hohe Effizienz erreicht werden (geringe Nutzung von Ressourcen), die aber wiederum auf Kosten des Makespans ging. Da aber ein kleiner Makespan als Hauptziel ausgegeben wurde, musste die Selektion des Clusters von den Prozessoren der Elterntasks entkoppelt werden. Aus diesem Grund wurde nur die verfügbare Rechenleistung der Cluster herangezogen und erst in Schritt 3 versucht, Elternprozessoren wiederzuverwenden.

### 5.5.3 Komplexitätsanalyse

Unter Zuhilfenahme des Pseudocodes von RePA in Alg. 5.3 lässt sich der Aufwand zum Erstellen des Ablaufplans (Schedule) abschätzen.

Es sei  $p_{max}$  die maximale Anzahl an Prozessoren, die ein Cluster der Clustermenge aufweist:

$$p_{max} = \max_{1 \leq k \leq c} \{p_k\}. \quad (5.7)$$

Die Gesamtanzahl der Prozessoren aller Cluster sei als  $P$  bezeichnet und definiert als:

$$P = \sum_k p_k. \quad (5.8)$$

Wie bereits erwähnt, wird der Algorithmus immer dann aufgerufen, wenn neue Tasks ausführbereit sind oder Tasks beendet wurden. Im schlechtesten Fall werden damit die Zeilen 1–9 für jede Task einmal ausgeführt. Nach der Sortierung der Tasks (Aufwand  $O(N \log N)$ ) wird der Zielcluster bestimmt. Es ist für alle Cluster zu überprüfen, über wie viel freie Rechenkapazität sie verfügen (Aufwand  $O(c)$ ). Dazu ist zu Beginn des Algorithmus einmal die Gesamtrechenleistung der Cluster zu bestimmen ( $O(P)$ ). Auch die Gesamtberechnungskosten der ausführbereiten Tasks sind am Start des Algorithmus zu ermitteln ( $O(N)$ ). Das Ermitteln der Allokationsgröße kann dann für jede Task in konstanter Zeit berechnet werden. Danach wird die Task auf die Prozessoren des Zielclusters abgebildet. Dazu müssen die Allokationen der Elterntasks überprüft werden. Die Selektion

der Prozessoren stoppt spätestens nach  $p_{max}$  Prozessoren. Insgesamt ergibt sich folgende Komplexität für den ReP-Algorithmus:

$$\begin{aligned} &O(N(c + p_{max} + N \log N) + N + P) \\ &= O(N(c + p_{max}) + N^2 \log N + P) \end{aligned} \tag{5.9}$$

### 5.5.4 Experimentelle Auswertung

In diesem Abschnitt wird die Scheduling-Leistung von RePA evaluiert. Dabei stellt sich zuerst die Frage nach dem „Wie“, denn RePA ist ein Algorithmus, um dynamische M-Taskgraphen auf Multiclustern abzubilden. In diesem Bereich gibt es keine verwandten Verfahren und man muss deshalb einen guten Vergleichsalgorithmus wählen. Andererseits sind Aussagen über Laufzeit-Experimente in dynamischen Umgebungen wie TGrid schlecht übertragbar. Experimente in realen Grid-Umgebungen haben zu viele Störfaktoren, um die Güte des Algorithmus ernsthaft abzuschätzen. Ein weiteres Problem sind die Kosten, die solche Messungen nach sich ziehen. Es müssten entsprechende Testprogramme geschrieben, Cluster reserviert und Messungen sehr oft wiederholt werden. Darüber hinaus lassen sich die Ergebnisse nur sehr schwer reproduzieren.

Aus den genannten Gründen wurde die Evaluation des Algorithmus mit Hilfe eines Simulators durchgeführt. Eine Simulation hat den Hauptvorteil, dass sich Ergebnisse sehr gut reproduzieren lassen. Natürlich spielt auch die Geschwindigkeit eine Rolle. Wird anstatt eines ausführbaren Programms nur ein mathematisches Modell zur Bestimmung der Laufzeit verwendet, kann man in vergleichsweise kurzer Zeit viele Testreihen aufnehmen.

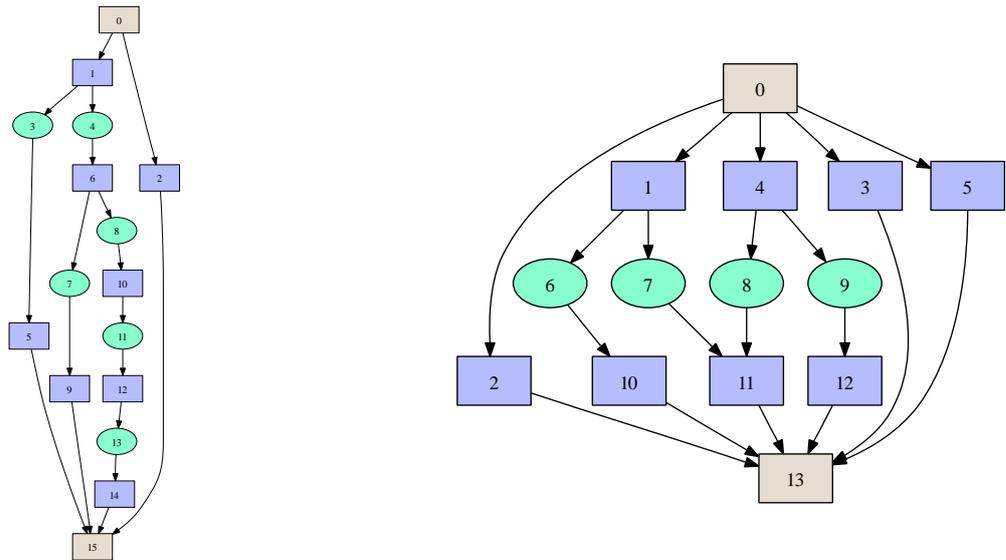
In den folgenden Abschnitten werden die Werkzeuge und Laufzeitmodelle beschrieben, die zur Simulation der Scheduling-Algorithmen eingesetzt werden.

#### Werkzeuge und Abstraktionsmodelle

**SimGrid** Als Simulationsplattform wurde SimGrid gewählt [66], das ursprünglich speziell für die Evaluation von Scheduling-Algorithmen entwickelt wurde. Später wurde es mit der Version 3 zu einem allgemeinen Simulationswerkzeug, um verteilte Anwendungen in heterogenen Umgebungen zu testen [24]. Casanova et al. untersuchen in [24] eine Reihe anderer Grid-Simulatoren und fassen deren Vor- und Nachteile zusammen. Bei Betrachtung dieser Zusammenstellung wird klar, dass kein anderer Simulator besser für die Tests im Rahmen dieser Arbeit geeignet ist. Alle weiteren Informationen, z. B. zur Programmierung von SimGrid mit C und Java, sind auf der zugehörigen Website<sup>2</sup> zu finden.

**Testfall-Generatoren** Damit die künstlichen Testszenarien ein möglichst großes Spektrum an Applikationen und Grid-Umgebungen abdecken, wurden Generatoren verwendet, um viele Systemparameter zufällig zu generieren. Für die Erzeugung einer zufälligen Beschreibung einer Plattform und einer Applikation (DAGs) wurde auf die Programme *daggen* [103] und *gridgen* zurückgegriffen. Die von Suter entwickelten Generator-Programme wurden schon für die Evaluation von MHEFT und HCPA eingesetzt. Die Funktionsweise

<sup>2</sup><http://simgrid.gforge.inria.fr>



**Abb. 5.5:** Beispielgraphen mit unterschiedlichem Grad an Taskparallelität. Links:  $fat = 0.5$ . Rechts:  $fat = 0.8$ .

und die vorhandenen Parameter beider Programme sind für *daggen* in Anhang A.1 und für *gridgen* in Anhang A.2 detailliert beschrieben.

Da in den folgenden Experimenten DAGs mit unterschiedlichen Charakteristika als Eingabe der Scheduler benutzt werden, zeigt die Abb. 5.5 exemplarisch zwei mit *daggen* erzeugte Graphen mit unterschiedlichem Grad an potenzieller Parallelität (*fat*-Wert bei *daggen*). Beide Graphen besitzen 8 ausführbare Tasks (blaue Rechtecke). Durch die unterschiedlichen Datenabhängigkeiten (grüne Ellipsen) zwischen den Tasks wird die Anzahl der gleichzeitig ausführbaren Tasks verändert. Können im linken Graph maximal vier Tasks parallel ausgeführt werden (z. B. 5, 9, 12, 2), sind es im rechten bereits sechs (2, 10, 11, 12, 3, 5).

**Performance-Modell für parallele Programme** Die Entwicklung, Implementierung und speziell das Testen von realen Programmen ist sehr zeitaufwendig. Deshalb wird zur Abschätzung der Laufzeiten der einzelnen Tasks ein Performance-Modell verwendet, mit dem sich die Laufzeit einer parallelen Task berechnen lässt. Für Laufzeitabschätzungen von parallelen Tasks werden im Scheduling-Umfeld oft *Amdahls Gesetz* [3] oder das *Modell von Downey* [38] eingesetzt. Für eine bessere Vergleichbarkeit wird in dieser Arbeit Amdahls Gesetz verwendet, da bei CPA, HCPA und MHEFT das gleiche Modell zu Grunde lag. Amdahls Gesetz besagt, dass sich die Berechnungszeit einer M-Task  $t$  aus der Zeit des parallelisierbaren Teils ( $1 - \alpha$ ) und des nicht-parallelisierbaren Teils ( $\alpha$ ) zusammensetzt. Die sequenzielle Laufzeit beträgt  $\tau$  und nur der parallelisierbare Teil des Programms kann beim Einsatz von  $p$  Prozessoren beschleunigt werden.

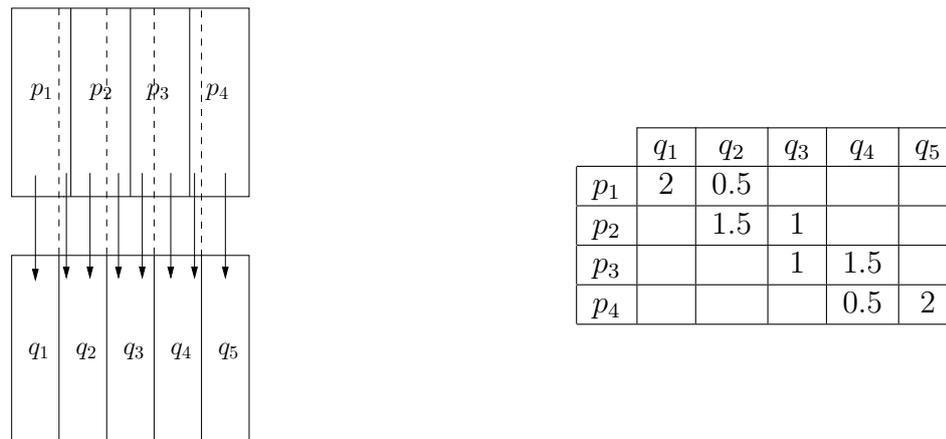
$$T(t, p) = \left( \alpha + \frac{1 - \alpha}{p} \right) \cdot \tau. \quad (5.10)$$

Die DAGs in unserem Modell besitzen deshalb einen Parameter, mit dem der  $\alpha$ -Wert einer Task definiert werden kann. Eine ausführlichere Betrachtung der Laufzeitvorhersage von datenparallelen Tasks mit Amdahls Gesetz ist in [93] zu finden.

Die Laufzeit einer Task wird im Experiment wie folgt bestimmt: Jede Task empfängt eine bestimmte Datengröße, die durch die Gewichte der Kanten zu den Elternknoten bestimmt ist. Auf diesen Daten wird eine Funktion mit einer bestimmten Komplexität angewendet, womit sich die Gesamtanzahl an auszuführenden Operationen (Anzahl von FLOP) ergibt, z. B. benötigt die Addition zweier Vektoren der Länge  $n$  auch  $n$  FLOP. Die Laufzeit für den parallelisierbaren und nicht-parallelisierbaren Teil wird nach Bestimmung der Allokation und mit dem gegebenen Prozessortyp berechnet. Die Geschwindigkeit des Prozessors wird im Modell in FLOPS (FLOP pro Sekunde) angegeben.

**Modell zur Datenkommunikation** Die Kommunikationszeit zwischen zwei Prozessoren hängt von der Anzahl der zu übertragenden Datenpakete und dem Verbindungsnetzwerk ab. Um die Anzahl und Größe der Pakete zwischen den Prozessoren zu bestimmen, bedarf es eines Daten- und eines Kommunikationsmodells. Das Datenmodell gibt an, welche Datenstrukturen mit welcher Verteilung auf den Prozessoren einer Task gespeichert sind. In dieser Arbeit wird das Modell der blockverteilte Matrizen (1D-Layout) verwendet, um Daten zwischen Tasks zu übertragen. Diese Datenverteilung kommt z. B. bei den gemischt-parallelen Implementierungen der Strassen-Matrixmultiplikation in Kapitel 2 oder in ScaLAPACK-Routinen [14] zum Einsatz. Ein Vorteil dieser blockweisen Verteilung ist, dass sie mit einer beliebigen Anzahl von Prozessoren verwendet werden kann. Trotzdem dient diese Modell primär der Abstraktion und Einbeziehung von Umverteilungskosten in die Simulation. Ein weitere Möglichkeit wäre die Verwendung eines blockzyklischen Datenlayouts, was aber relativ hohe Berechnungskosten für die Bestimmung der Kommunikationsmatrix nach sich ziehen würde. Da das grundlegende Prinzip für beide Modelle gilt, dass die Kommunikationskosten geringer sind, wenn zwischen zwei gleichen Prozessorgruppen kommuniziert wird, ist das Modell der blockverteilten Matrizen besser für die Simulationen geeignet.

Die Abb. 5.6 zeigt beispielhaft die Umverteilung einer blockverteilten Matrix von 4 auf 5 Prozessoren. Da die Matrizen als Spaltenblöcke auf die Prozessoren verteilt sind, müssen 4 Spaltenblöcke von der sendenden Prozessorgruppe auf 5 Spaltenblöcke der empfangenden Gruppe umverteilt werden. Besteht die Matrix aus 20 Spalten kann man sie in 10 Blöcken (a 2 Spalten) auf die Prozessoren verteilt betrachten. Die notwendigen Nachrichten zur Datenumverteilung zwischen den Prozessorgruppen kann mit Hilfe der Überdeckung der beiden virtuellen Matrixgitter berechnet werden. Im Beispiel bedeutet das, dass Prozessor  $p_1$  einen Block der Größe 2 an Prozessor  $q_1$  schicken muss und den restlichen Teil, den er speichert, der Größe 0.5, wird an  $q_2$  geschickt. Die so entstehende Kommunikationsmatrix ist auf der rechten Seite dargestellt, wobei die Prozessoren in den Zeilen Daten versenden und die in den Spalten Daten empfangen. Deutlich wird hierbei, dass alle Prozessoren zwar insgesamt gleich viele Daten verschicken, dazu aber unterschiedlich viele Übertragungen notwendig sind. Es wird deshalb auch klar, dass ein mehrmaliger Verbindungsaufbau zwischen Clustern bei hoher Latenzzeit einen erheblichen Einfluss auf die Kommunikationszeit haben kann.



**Abb. 5.6:** Beispiel einer Matrixumverteilung zum Senden von 10 Datenpaketen von 4 Prozessoren an 5 empfangende Prozessoren. Links: schematische Darstellung. Rechts: Zugehörige Kommunikationsmatrix.

### Konfiguration und Vorbereitung der Simulation

Unter Beachtung der zuvor definierten Performance- und Plattformmodelle wurde aufbauend auf SimGrid ein Simulator entwickelt, mit dem die Leistung von RePA in verschiedenen Testszenarien evaluiert werden kann. Da kein weiterer Scheduling-Algorithmus für dynamische M-Taskgraphen existiert, musste ein statischer Ansatz als Vergleichsalgorithmus benutzt werden. Zum Zeitpunkt der Untersuchungen waren MHEFT und HCPA die einzigen Algorithmen, die ein Scheduling von M-Tasks in heterogenen Cluster-Umgebungen unterstützten. Da MHEFT im Vergleich zu HCPA die kürzeren Ablaufpläne (*schedules*) generierte, wurde MHEFT [23] (siehe Abschnitt 5.3) als Vergleichsalgorithmus gewählt.

Als Plattformen wurden zwei Typen von Grid-Umgebungen generiert. Der erste Typ besteht aus einer homogenen Ansammlung von Clustern, wohingegen sich der zweite aus heterogenen Clustern zusammensetzt. Im Fall der homogenen Menge weisen alle Cluster den gleichen Prozessortyp auf. Trotzdem können die Cluster eine unterschiedliche Anzahl dieser Prozessoren besitzen. Der heterogene Multiclustern beschreibt den zuvor mehrfach angesprochenen Fall, dass die Prozessoren und deren Leistung von Cluster zu Cluster variieren können. In beiden Fällen wurde die Clustergröße zufällig bestimmt (im Bereich zwischen 16 und 64 Prozessoren), da viele verfügbare Cluster-Systeme im universitären Bereich diese Installationsgröße aufweisen. Als Verbindungsnetzwerk der Cluster wurde ein 10-GBit-Ethernet-Netzwerk angenommen (Latenz von  $100 \mu s$ ). Die Geschwindigkeit der Prozessoren wurde für jeden Cluster zufällig zwischen 1 und 1.5 GFLOPS festgelegt. Es wurden jeweils 10 zufällige Grid-Umgebungen mit 4 und 8 Clustern im homogenen und im heterogenen Fall generiert. Insgesamt wurden somit 40 verschiedene Konfigurationen erzeugt.

Neben den Plattformbeschreibungen mussten auch DAGs für die Experimente generiert werden. Die Anzahl der Berechnungsknoten für die DAGs sollte 50, 75 oder 100 betragen, da größere DAGs wegen der hohen Zeitkomplexität von MHEFT schlechter evaluiert werden konnten. Für jede Anzahl von Tasks wurden 50 DAGs generiert. Jedem Knoten

**Tab. 5.1:** Vergleich des Makespans von RePA und MHEFT mit 4 und 8 Clustern.

#Tasks		4 Cluster			8 Cluster		
		fat	homogen	heterogen	homogen	heterogen	
50	min RePA makespan [%]	0.6	87.3	89.3	104.0	104.9	
		0.9	74.1	69.8	93.0	86.4	
	max RePA makespan [%]	0.6	130.4	151.4	111.9	151.4	
		0.9	127.6	137.3	135.0	169.0	
	avg RePA makespan [%]	0.6	104.5	112.8	107.0	120.6	
		0.9	99.0	102.0	117.5	124.0	
	RePA wins / #Tests	0.6	145/500	36/500	0/500	0/500	
		0.9	272/500	222/500	10/500	4/500	
	75	min RePA makespan [%]	0.6	85.0	88.7	108.2	110.5
			0.9	64.7	61.7	86.6	81.7
max RePA makespan [%]		0.6	131.1	166.2	127.3	165.8	
		0.9	113.9	131.6	139.9	167.7	
avg RePA makespan [%]		0.6	106.0	117.6	115.5	132.2	
		0.9	91.6	92.1	114.5	122.4	
RePA wins / #Tests		0.6	112/500	22/500	0/500	0/500	
		0.9	422/500	381/500	27/500	12/500	
100		min RePA makespan [%]	0.6	90.3	94.8	110.6	117.5
			0.9	60.7	57.6	85.5	83.3
	max RePA makespan [%]	0.6	132.1	154.0	140.6	169.1	
		0.9	116.4	121.5	140.4	161.2	
	avg RePA makespan [%]	0.6	107.3	118.9	123.3	141.0	
		0.9	89.8	87.1	113.3	121.2	
	RePA wins / #Tests	0.6	84/500	12/500	0/500	0/500	
		0.9	449/500	431/500	45/500	25/500	

(Tasks) wurde ein randomisierter  $\alpha$ -Wert (Amdahl) zwischen 0 und 0.2 zugewiesen. Außerdem kann mit dem DAG-Generator der resultierende Grad an Parallelität durch Angabe des *fat*-Wertes verändert werden (siehe Anhang A.1). Die Angabe eines größeren *fat*-Wertes schafft mehr unabhängige Tasks pro DAG-Ebene. Die DAGs wurden mit *fat*-Werten von 0.6 und 0.9 generiert.

### Simulationsergebnisse

Die gesamten Simulationsergebnisse sind in Tab. 5.1 zusammengestellt. Die Tabelle gliedert sich in die drei Abschnitte mit den Resultaten für DAGs mit 50, 75 und 100 Tasks. Für jede Taskanzahl wurden pro Gridart (homogen oder heterogen) unterschiedliche Leistungsmaße berechnet. Es gibt drei verschiedene Werte, um den erreichten Makespan zu charakterisieren, die im Folgenden definiert werden.

Das Scheduling eines DAGs  $\mathcal{D}_i$  auf einer Grid-Plattform  $\mathcal{G}_j$  führt bei RePA zu einem Makespan  $M_{REPA}(i, j)$ . Analog dazu wird der Makespan für diese Eingabe bei MHEFT mit  $M_{MHEFT}(i, j)$  bezeichnet. Zu einem besseren Vergleich der Algorithmen wird der relative Makespan von RePA herangezogen, der wie folgt definiert ist

$$M_{rel}(i, j) = \frac{M_{REPA}(i, j)}{M_{MHEFT}(i, j)}. \quad (5.11)$$

Der relative Makespan wird für alle Eingabepaare aus Gridplattform und DAG ermittelt. Mit diesen Werten ergibt sich der minimale relative Makespan als

$$M_{rel}^{min} = \min_{\forall D_i \forall G_j} \{M_{rel}(i, j)\}. \quad (5.12)$$

Die Größen für den maximalen und durchschnittlichen relativen Makespan werden analog berechnet. In der Tabelle werden verschiedene Werte des relativen Makespans (min, max, avg) für die unterschiedlichen DAGs pro Grid-Konfiguration in Prozent angegeben. In der letzten Zeile jedes Abschnitts ist der absolute Vergleich der gemessenen Makespans verdeutlicht, d. h. die Anzahl der Fälle, in denen RePA kürzere Schedules als MHEFT produziert. Mit „#Tests“ wird die Gesamtanzahl der Tests pro Konfiguration angegeben (500 = 50 DAGs bei 10 verschiedenen Gridkonfigurationen). Der Wert *fat* bezeichnet den schon mehrfach angesprochenen Parallelitätsgrad (Breite) der DAGs.

Betrachtet man den durchschnittlichen relativen Makespan in Tab. 5.1 bei *fat* = 0.9 ist zu erkennen, dass dieser mit einer steigenden Zahl von Tasks abnimmt. Das bedeutet, dass RePA die vorhandene Parallelität besser ausnutzen kann. Für die aus vier Clustern bestehenden Plattformen erreichte RePA bei fast allen Tests (*fat* = 0.9) einen relativen Makespan von weniger als 100 % und gewinnt bei der absoluten Auszählung in mehr als 50 % der Fälle (272/500 bis 449/500). Für DAGs mit einem *fat*-Wert von 0.6 sieht die Statistik ein wenig anders aus. Bei diesen DAGs kann man erkennen, dass der durchschnittliche relative Makespan mit zunehmender DAG-Größe leicht anwächst (zwischen 104 und 118 %). Das liegt vor allem daran, dass die Tiefe der DAGs bei gleicher Taskanzahl bei kleinerem *fat*-Wert steigt. Damit gibt es zum einen weniger gleichzeitig ausführbare Tasks und andererseits mehr negative Auswirkungen des unbekanntes kritischen Pfades (wie im Fall von RePA). MHEFT hat dieses Wissen und kann somit über die gesamte Tiefe der DAGs optimieren. Trotzdem liegt RePA, mit einem Makespan, der maximal 20 % schlechter ist, im Vergleich zum statischen Ansatz MHEFT gut im Rennen. Der Hauptgrund für die besseren Schedules von RePA bei großen *fat*-Werten ist die bessere Ausnutzung der Taskparallelität. MHEFT tendiert dazu, die Cluster mit einer einzigen Task komplett zu belegen, auch wenn der Zeitgewinn nur minimal ist. Das ist eine durch die Verwendung von Amdahls Gesetz gererbte Problematik, welche durch  $\Delta$ -CTS-Verfahren [104] verbessert wird.

Im Vergleich von heterogenen und homogenen Systemen ist festzuhalten, dass eine größere Heterogenität den Makespan von RePA in Relation zu MHEFT erhöht. Das ist durch eine bessere Analysemöglichkeit von schmaleren und tieferen DAGs und der Betrachtung der Kommunikationszeit von MHEFT begründet. Außerdem hat MHEFT noch einen weiteren Vorteil, welcher in solchen Situation besser zum Tragen kommt: Da der Schedule nach der Compile-Zeit bekannt ist, weiß MHEFT zur Laufzeit auch, wohin jede Task im Grid abgebildet werden soll. Somit kann schon mit der Datenkommunikation einer fertig gestellten Task begonnen werden, auch wenn die Kindtasks noch gar nicht ausführbereit sind. Bei RePA muss erst abgewartet werden, bis die Kindtask abgebildet wurde, damit die Elterntasks ihr Umverteilung beginnen können. Ein weiterer kleiner Faktor für eine schlechtere durchschnittliche Performance von RePA ist der Informationsaustausch zwischen Scheduler und Clustern. Im Modell von RePA arbeitet der Scheduler auf einem

dedizierten Cluster der Gesamtmenge. Das ist auch ein realistisches Modell der Arbeitsweise von Meta-Schedulern. Haben nun Tasks auf einem anderen Cluster ihre Ausführung beendet, muss der Scheduler benachrichtigt werden. Dazu sind Nachrichten zu verschicken, die zum Gesamtkommunikationsoverhead dazugezählt werden müssen. Das bedeutet im Speziellen, dass – auch bei identischem Mapping jeder Task des DAGs – RePA durch den zusätzlichen Nachrichtenoverhead einen größeren Makespan produziert.

Zusammenfassend kann man sagen, dass RePA erstaunlich gut im Vergleich zu einem Compile-Zeit-Algorithmus abschneidet. Die berechneten Schedules sind bei großem Parallelitätsgrad ( $fat = 0.9$ ) sogar oft besser als die von MHEFT. Die Leistung von RePA ist respektabel, so dass RePA als Scheduling-Algorithmus innerhalb des TGrid-Frameworks eingesetzt werden kann.

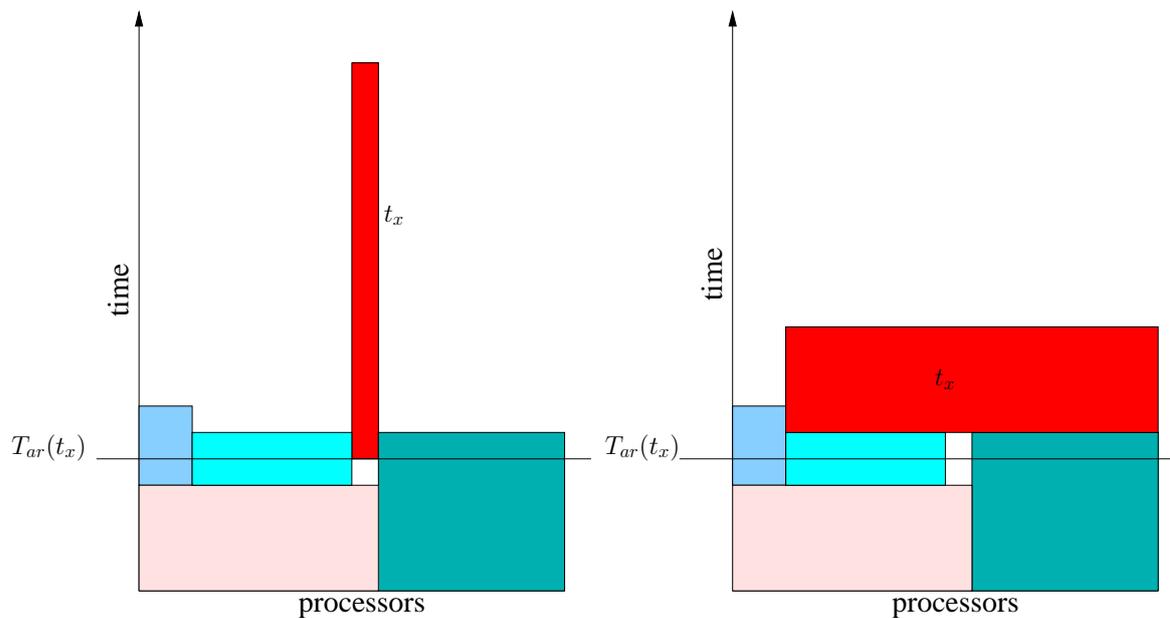
## 5.6 DMHEFT

### 5.6.1 Motivation und Ziel

Im vorherigen Abschnitt wurde der ReP-Algorithmus (*Reuse Processor Algorithm*) beschrieben, der als möglicher Scheduling-Algorithmus innerhalb des TGrid-Frameworks in Frage kommt. Der Algorithmus arbeitet in drei Schritten, um eine brauchbare Abbildung einer Task auf Prozessoren zu finden. Zuerst bestimmt der Algorithmus den Zielcluster, welcher anhand der verfügbaren Rechenkapazität ausgewählt wird (EFT-Strategie). Danach wird die Allokation auf diesem Cluster bestimmt, deren Größe an die Anzahl der auszuführenden Operationen einer Task geknüpft ist, um auch noch andere wartende Tasks ausführen zu können. Im dritten Schritt wird diese Allokation auf reale Prozessoren des Systems abgebildet, wobei Prozessoren von Elterntasks bevorzugt wiederverwendet werden. Mit dieser Technik lässt sich die Kommunikationszeit reduzieren, die bei der Datenumverteilung benötigt wird. Die experimentellen Auswertungen haben gezeigt, dass RePA im Vergleich zu statischen Ansätzen wie MHEFT gute Schedules produziert, d. h. der durchschnittliche Makespan nicht viel größer als der von MHEFT ist.

Gleichwohl wurden die Kombinationen aus DAG und Grid-Plattform genauer untersucht, bei denen RePA deutlich schlechtere Ergebnisse als MHEFT lieferte oder bei denen man eine Verbesserungsmöglichkeit erkennen konnte. Eine weitere wichtige Neuerung war die Einführung des Workstation-Modells `ptask_L07` in SimGrid. Mit diesem Modell konnten auch die Netzwerk-Contention und die Latenzzeiten bei der Simulation der Scheduling-Algorithmen einbezogen werden. Damit hatten die Umverteilungskosten in den Simulationen einen noch höheren Stellenwert.

Bei der genauen Untersuchung der von RePA produzierten Schedules, unter Verwendung des neuen Workstation-Modells, wurde deutlich, dass ein Nichtbeachten der Kommunikationskosten (Umverteilungskosten) bei der Bestimmung des Zielclusters oft zu einem großen Kommunikationsoverhead führt. Dieser Overhead fällt speziell dann ins Gewicht, wenn eine Inter-Cluster-Kommunikation angestoßen werden muss. Ein weiteres Problem besteht darin, dass RePA immer versucht, alle Prozessoren aller Cluster mit Arbeitslast zu füllen. Da in dem betrachteten Modell die Tasks *modal* sind, d. h. sich die Anzahl der zugeordneten



**Abb. 5.7:** Problem von RePA: Das Ausführen einer Task  $t_x$  zur Zeit  $T_{ar}(t_x)$  kann zu einem ineffizienten Schedule führen, wenn die Anzahl der freien Prozessoren sehr klein ist.

Prozessoren während der Ausführung nicht ändern kann, muss mit ungünstigen Allokationen gelebt werden. Das daraus resultierende Problem ist in Abb. 5.7 illustriert. In der linken Grafik wird zur Zeit  $T_{ar}$  die Task  $t_x$  ausführbereit. In diesem Fall versucht RePA die größte Task (LPT-Strategie) auf die verfügbaren Prozessoren abzubilden. Als Konsequenz daraus wird  $t_x$  auf einer kleineren Menge an Prozessoren ausgeführt, wodurch die Laufzeit extrem steigt (Amdahls Gesetz)<sup>3</sup>. Diese Abbildung wird dann zum Geschwindigkeitsproblem, wenn die Anzahl der verfügbaren Tasks in der Zeitspanne, in der  $t_x$  ausgeführt wird, sehr gering ist. Es würde zu „Löchern“ innerhalb des Schedules kommen, was eindeutig auf eine schlechtere parallele Effizienz hinweist. Darüber hinaus könnten von  $t_x$  eine Reihe von anderen Tasks abhängen. Betrachtet man den Graphen, ließe sich sagen, dass dadurch das Bottom-Level vieler abhängiger Tasks unnötig vergrößert wird. Eine mögliche Lösung des letzteren Problems wäre die Anwendung einer Verzögerungsstrategie. Dabei wird die Ausführung der Task  $t_x$  solange hinausgezögert bis mehr Prozessoren verfügbar sind. Diese Variante ist auf der rechten Seite von Abb. 5.7 veranschaulicht.

Aus den genannten Gründen werden die folgenden zwei Ziele definiert, um mit einer Weiterentwicklung von RePA eine Verbesserung des durchschnittlichen Makespan zu erreichen:

1. *Berücksichtigung der Kommunikations- und Umverteilungskosten* bei der Wahl des Zielclusters, um den Kommunikationsoverhead zu senken.
2. *Anwendung einer Verzögerungsstrategie (postponing)* für rechenintensive Tasks, um die Effizienz des Schedules zu erhöhen und den Makespan zu reduzieren.

<sup>3</sup>An dieser Stelle könnten auch *malleable* Tasks Abhilfe schaffen, bei denen sich die Zuordnung zu einem späteren Zeitpunkt dynamisch ändern ließe.

## 5.6.2 Beschreibung des DMHEFT-Algorithmus

Der hier beschriebene Algorithmus ist eine Weiterentwicklung von RePA und versucht, die oben erwähnten Zielsetzungen zu erfüllen. Der Name „DMHEFT“ (Dynamic Mixed-Parallel Heterogeneous Earliest Finish Time) wurde gewählt, weil er dynamisch erzeugte Graphen aus M-Tasks mit Hilfe einer EFT-Strategie auf eine heterogene Menge von Clustern abbildet. Auch wenn der Name so vermuten lässt, basiert DMHEFT nicht direkt auf dem MHEFT-Algorithmus aus [23]. Wie zuvor erwähnt, sind die Grundideen von DMHEFT und RePA gleich, d. h. beide nutzen die gleiche Schrittfolge: (1) Auswahl des Zielclusters, (2) Bestimmung der Allokation und (3) Mapping der Allokation auf den Zielcluster.

Der Alg. 5.4 zeigt den Pseudocode von DMHEFT. Die Zeilen 1–4 werden immer dann ausgeführt, wenn neue Tasks ausführbereit sind. Es wird dann versucht, für jede ausführbare Task eine Abbildung im System zu finden. Gelingt das, wird die Task im Grid ausgeführt. Das Abarbeiten der Liste mit ausführbaren Tasks wird von der Funktion `schedule()` durchgeführt. Wie bei RePA wird die Liste zuerst nach fallenden Berechnungskosten und steigender Knotentiefe sortiert (LPT). Nach der Sortierung wird versucht, für jede Task den optimalen Cluster zu ermitteln (Zeile 10). Optimal heißt dabei, dass die Beendigungszeit (Startzeit + Ausführungszeit) der Task minimiert werden soll. Dazu wird die Funktion `get_best_cluster()` benötigt, die für jeden Cluster eine mögliche Allokation und jeweils ein Mapping vorschlägt, mit denen die Kommunikationskosten zum Auflösen einer Datenabhängigkeit abgeschätzt werden können. Die Größe der Allokation wird mit derselben Heuristik wie bei RePA bestimmt (Zeile 25), d. h. es fließen die verfügbare Rechenleistung der Cluster und die Berechnungskosten der ausführbaren Knoten in die Berechnung ein, siehe Funktion `get_allocation_on_cluster()` in Abschnitt 5.5.2. Auch die Bestimmung des Mappings der Allokation wird von RePA übernommen (`map_allocation_on_cluster()`). Damit werden die Prozessoren der Elterntasks bevorzugt ausgewählt. Die Abschätzung der Kommunikationszeit für das aktuelle Mapping wird durch die Funktion `estimate_time()` (Zeile 27) berechnet. Der Cluster, mit dem die kleinste Beendigungszeit (*completion time*) der Task erreicht werden kann, wird mit dem zugehörigen Mapping zurückgegeben. Wird an dieser Stelle kein Cluster gefunden, ist das System ausgelastet (*saturated*) und das Scheduling wird unterbrochen.

Nachdem der Zielcluster und das Mapping bestimmt sind, wird überprüft, ob diese Wahl für die aktuelle Task ein gutes Resultat liefert oder ob eine Verzögerungsstrategie angewendet werden sollte. Um dies abzuschätzen, wird überprüft, ob es sich um eine schon verzögerte Task handelt (Zeile 13). Ist dies der Fall und ein Schwellenwert  $T_{pp}^{max}$  (*threshold*) wurde bereits überschritten, wird die Task mit der gerade ermittelten Abbildung ausgeführt. Ansonsten wird die Ausführung der Task weiter verzögert. Um die Güte einer möglichen Verzögerung zu bewerten, implementiert die Funktion `postponable` eine entsprechende Heuristik. Schätzt diese Funktion die aktuelle Task als *verzögerbar* ein, wird diese an eine separate Liste von verzögerten Tasks angehängt.

**Algorithmus 5.4 DMHEFT**

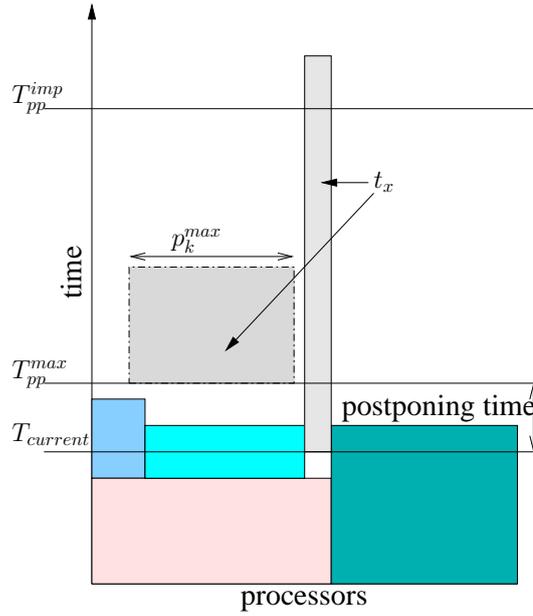

---

```

1: while not done do
2:   node_schedules = schedule( get_ready_nodes() )
3:   for each s in node_schedules do
4:     run_task( s )
function schedule ( list ready_nodes )
5:    $Q = \{ ready\_nodes \} \cup PQ$  // PQ contains postponed tasks
6:   sort Q by increasing depth and decreasing computation amount
7:   node_schedules = {}
8:   while Q is not empty and clusters not saturated do
9:      $t_i = \text{pop}(Q)$ 
10:     $[C_{best}, T_{best}, lp_{best}] = \text{get\_best\_cluster}(t_i)$ 
11:    if  $C_{best}$  is not none then // clusters not saturated
12:      task_schedule =  $(t_i, lp_{best}, T_{current}, T_{best})$ 
13:      if  $t_i$  is postponed then
14:        if  $T_{current} + T_{best} < T_{pp}^{imp}$  or  $T_{current} \geq T_{pp}^{max}$  then
15:          add task_schedule to node_schedules // force scheduling
16:        else
17:          append  $t_i$  to PQ
18:        else if postponable( $t_i, T_{best}$ ) then
19:          append  $t_i$  to PQ
20:        else add task_schedule to node_schedules
21:   return node_schedules
function get_best_cluster( task t, queue Q )
23:    $C_{best} = \text{none}, T_{best} = \text{none}, lp_{best} = \{ \}$ 
24:   for each cluster  $C_j$  do
25:      $p_{C_j} = \text{get\_allocation\_on\_cluster}(t, C_j)$ 
26:      $lp_{avail} = \text{map\_allocation\_on\_cluster}(C_j, p_{C_j})$ 
27:      $T_j^{est} = \text{estimate\_time}(t_i, lp_{avail})$ 
28:     if  $(T_j^{est} < T_{best})$  or  $(C_{best} \text{ is none})$  then
29:        $[C_{best}, T_{best}, lp_{best}] = [C_j, T_j^{est}, lp_{avail}]$ 
30:   return  $[C_{best}, T_{best}, lp_{best}]$ 
function postponable( task t ,  $T_t^{est}$  )
31:    $T_{pp}^{max} = T_{current} + f_{pp} \cdot T_t^{est}$  // max postponing time span
32:    $T_{pp}^{imp} = T_{current} + f_{pi} \cdot T_t^{est}$  // improvement minimum
33:   for each cluster  $C_k$  do
34:      $p_{pp}^{T_{pp}^{max}} = \text{get\_free\_processors\_at\_time}(T_{pp}^{max}, C_k)$ 
35:      $p_k^{max} = f_{cu} \cdot p_{pp}^{T_{pp}^{max}}$  // only a fraction of the processors might be available
36:      $lp_{pp}^{T_{pp}^{max}} = \text{map\_allocation\_on\_cluster}(C_k, p_k^{max})$ 
37:      $T_{i,k}^{pp} = \text{estimate\_time}(t, lp_{pp}^{T_{pp}^{max}})$ 
38:     if  $T_{pp}^{max} + T_{i,k}^{pp} \leq T_{pp}^{imp}$  then
39:       return true // postponing will probably work
40:   return false

```

---



**Abb. 5.8:** Postponing-Strategie von DMHEFT für die M-Task  $t_x$ .

**Heuristik zur Taskverzögerung** Die grundlegende Idee der verwendeten Heuristik zur Taskverzögerung ist in Abb. 5.8 skizziert. Der Scheduler versucht zum Zeitpunkt  $T_{current}$ , die Task  $t_x$  auf den Cluster abzubilden. Es wird dazu probiert, den Start der Task soweit in die Zukunft zu verlegen, dass eventuell mehr Prozessoren für eine Zuweisung zur Verfügung stehen. Aus der aktuellen Zeit und der berechneten Beendigungszeit der Task werden drei Werte bestimmt, die bei der Abschätzung einer möglichen Verzögerung helfen:

- $T_{pp}^{max}$  definiert die Zeitspanne, die eine Task maximal verzögert werden kann. Sie ergibt sich aus der aktuellen Zeit  $T_{current}$  und der geschätzten Ausführungszeit  $T_t^{est}$  von Task  $t_x$  als

$$T_{pp}^{max} = T_{current} + f_{pp} \cdot T_t^{est}, 0 \leq f_{pp} < 1. \quad (5.13)$$

- $T_{pp}^{imp}$  bezeichnet den Zeitpunkt, zu dem die verzögerte Task spätestens die Abarbeitung abgeschlossen haben muss, damit eine Verzögerung gerechtfertigt werden kann.

$$T_{pp}^{imp} = T_{current} + f_{pi} \cdot T_t^{est}, 0 < f_{pi} \leq 1 \wedge T_{pp}^{imp} \geq T_{pp}^{max}. \quad (5.14)$$

- $p_k^{max}$  bestimmt die Anzahl der Prozessoren, die zum Zeitpunkt  $T_{pp}^{max}$  der Task  $t_x$  zugewiesen werden können.

$$p_k^{max} = f_{cu} \cdot p^{T_{pp}^{max}}, 0 < f_{cu} < 1, \quad (5.15)$$

wobei  $p^{T_{pp}^{max}}$  die Anzahl der verfügbaren Prozessoren zum Zeitpunkt  $T_{pp}^{max}$  charakterisiert.

Die Variablen  $f_{pp}$ ,  $f_{pi}$  und  $f_{cu}$  sind Anpassungsparameter, welche später näher betrachtet werden.

Die so definierten Werte für  $T_{pp}^{max}$ ,  $p_k^{max}$  und  $T_{pp}^{imp}$  werden wie folgt verwendet: Der Wert  $T_{pp}^{max}$  bezeichnet die Zeit zu der die verzögerte Task gestartet werden muss. Der Scheduler versucht nun, die Anzahl der verfügbaren Prozessoren zu diesem Zeitpunkt zu bestimmen. Dabei werden die geschätzten Beendigungszeiten der aktuell laufenden Tasks betrachtet (`get_free_processors_at_time()`). Da es eher unwahrscheinlich ist, dass zum Zeitpunkt  $T_{pp}^{max}$  nur Task  $t_x$  ausführbar ist, wird die Allokation von  $t_x$  auf einen Teil der verfügbaren Prozessoren ( $p_k^{max}$ ) limitiert. Mit der Allokation kann der Scheduler die Ausführungszeit der Task  $t_x$  abschätzen. Da die Anzahl der ausführbaren Tasks zum Zeitpunkt  $T_{pp}^{max}$  schwer vorhersagbar ist, wird neben der Limitierung der Allokation auch noch ein signifikanter Leistungsgewinn gefordert. Dieser Schwellenwert wird als Zeit  $T_{pp}^{imp}$  bezeichnet. Das bedeutet, dass die Task  $t_x$  nur verzögert wird, wenn das Verzögern, die Kommunikation und das Ausführen von  $t_x$  spätestens zum Zeitpunkt  $T_{pp}^{imp}$  beendet ist.

Der dargestellte Algorithmus von DMHEFT verwendet eine Verzögerungsstrategie mit einer festen Verzögerungsschranke, d. h. wenn die Task  $t_x$  zum Zeitpunkt  $T_{pp}^{max}$  noch nicht gestartet wurde, wird sie bei der nächsten Gelegenheit mit der dann besten Allokation abgebildet. Es kann aber auch der Fall betrachtet werden, in dem keine Schranke zum Einsatz kommt. In diesem Fall könnte eine Task solange verzögert werden wie die Verzögerungsbedingung gilt. Aus diesem Grund wurde mit zwei verschiedenen Versionen des Algorithmus experimentiert: (1) Verzögerung mit einer festen Schranke und (2) unbeschränkte Verzögerung möglich. Für den zweiten Fall sind die Zeilen 13–18 im Algorithmus zu entfernen. Im Fall (1) wird eine Task spätestens dann ausgeführt, wenn  $T_{pp}^{max}$  erreicht wurde. Im Fall (2) wird  $T_{pp}^{max}$  nur benutzt, um ein mögliches weiteres Verzögern abzuschätzen.

### 5.6.3 Komplexitätsanalyse

Wie in Gleichung (5.7) definiert, bezeichnet  $p_{max}$  die maximale Anzahl von Prozessoren, über die einer der Cluster verfügt. Außerdem sei  $P$  die Gesamtanzahl der Prozessoren des Multiclusters. Der DMHEFT-Algorithmus verwendet eine Laufzeitabschätzung der ausführbaren Tasks, um gute Scheduling-Entscheidungen zu fällen. Die Abschätzung der Beendigungszeit einer parallelen Tasks erfordert jedoch verschiedene Annahmen und kann von Modell zu Modell unterschiedlich sein. Möchte man z. B. die Kommunikations- und Umverteilungskosten zwischen kooperierenden M-Tasks einbeziehen, benötigt man ein Modell des Netzwerks und ein Modell der Datenverteilung. Des Weiteren ist ein Modell zur Abschätzung der parallelen Laufzeit notwendig. Aus diesem Grund wird die Komplexität der Abschätzung der Laufzeit einer Task mit einer gegebenen Allokation als  $\xi$  bezeichnet. Die Abschätzung der Laufzeit einer Task  $t_i$  wird im Algorithmus durch die Funktion `estimate_time` realisiert. Für diese Funktion wird im Folgenden eine Berechnungskomplexität von  $\xi$  angenommen.

Die Komplexität der Funktion `schedule()` setzt sich wie folgt zusammen: Zuerst werden die ausführbaren Tasks sortiert ( $O(N \log N)$ ). Danach wird für jede ausführbare Task der Zielcluster mit der Funktion `get_best_cluster()` bestimmt. Darin wird für jeden Cluster eine mögliche Allokation und ein Mapping ermittelt. Mit diesen kann die Beendigungszeit der Task auf diesem Cluster berechnet werden. Die Komplexität der Funktion `get_best_cluster()` ist damit  $O(c \cdot (p_{max} + \xi))$ . Mit der bestimmten Allokation und dem

Mapping werden die Randbedingungen der Verzögerungsstrategie ermittelt, um danach die Möglichkeit einer Taskverzögerung mit Hilfe der Funktion `postponable()` zu bestimmen. Die Komplexität von `postponable()` kann folgendermaßen abgeschätzt werden: Es werden die freien Prozessoren auf einem Cluster am Ende der Verzögerungszeit berechnet. Verwendet man eine Liste, die für jeden Prozessor die Zeit angibt, zu der er wieder verfügbar ist, kann die Anzahl der freien Prozessoren in  $O(p_{max})$  Schritten durchgeführt werden. Die Komplexität der Funktion `postponable()` ist damit:  $O(c \cdot (2p_{max} + \xi)) = O(c \cdot (p_{max} + \xi))$ . Somit folgt für die Komplexität der `schedule()`-Funktion:

$$O(N \log N + N(c \cdot (p_{max} + \xi))) . \quad (5.16)$$

Es gilt noch zu klären, wie oft die Funktion `schedule()` aufgerufen wird. Jede Task wird höchstens einmal ausführbereit, d. h. im schlechtesten Fall kommt es zu einer serialisierten Ausführung aller Tasks und `schedule()` wird  $N$  mal aufgerufen. Außerdem ist bei Beginn des Scheduling die verfügbare Rechenkapazität der Cluster zu berechnen ( $O(P)$ ). Diese Werte werden bei der späteren Ausführung von Tasks nur noch aktualisiert. Daraus folgt für die Berechnungskomplexität von DMHEFT:

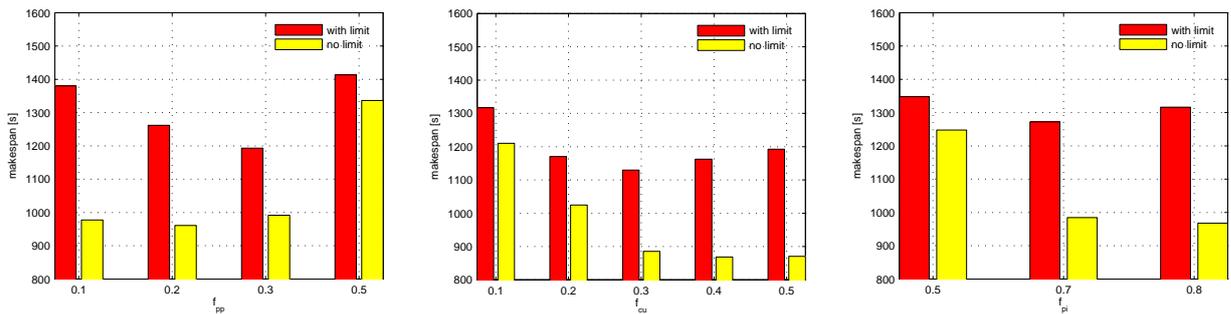
$$\begin{aligned} &O(N(N \log N + N(c(p_{max} + \xi))) + P) \\ &= O(N^2 \log N + N^2 c(p_{max} + \xi)) . \end{aligned} \quad (5.17)$$

#### 5.6.4 Experimentelle Auswertung

Die Qualität der von DMHEFT produzierten Schedules wurde anhand einer Reihe von Simulationsläufen mit verschiedenen DAG- und Grid-Konfigurationen getestet. Wie bei RePA werden randomisierte, synthetische Taskgraphen benutzt, um die Vielfalt der verschiedenen Applikationsklassen abzudecken. Auch hierbei wurden die zuvor beschriebenen Generatoren zur Erzeugung zufälliger Grids und DAGs verwendet (siehe Anhang A). Als Modell zur Datenverteilung wurde wieder auf eine blockverteilte Matrix zurückgegriffen. Insgesamt wurden für die Experimente 144 verschiedene DAGs generiert, die aus 25, 50, 75 oder 100 Tasks bestanden ( $fat \in \{0.5, 0.7, 0.9\}$ ). Außerdem wurden 40 Grid-Plattformen mit jeweils gleichem Anteil aus homogenen und heterogenen Clustern generiert, wobei jeder Multicluster aus 4 oder 8 Clustern bestand. Die Leistung der Prozessoren wurde zufällig im Bereich von 1 GFLOPS bis 1.5 GFLOPS gewählt. Das Verbindungsnetzwerk der Cluster wurde als 10-GBit-Ethernet-Netzwerk mit  $100 \mu s$  Latenz festgelegt.

Alle Experimente wurden mit einem Simulator ausgeführt, der auf dem SimGrid Toolkit [24] (revision 4988) aufsetzt. Als Workstation-Modell wurde das von SimGrid bereitgestellte `ptask_L07`-Modell genutzt, welches Netzwerk-Contention und Latenzzeiten berücksichtigt.

**Empirische Parameterbestimmung der Verzögerungsheuristik** Bevor DMHEFT mit anderen Algorithmen direkt verglichen werden kann, müssen zuerst geeignete Werte für die Parameter der Verzögerungsfunktionen bestimmt werden. Die Parameter wurden in Abschnitt 5.6.2 definiert und sind im Einzelnen:



**Abb. 5.9:** Empirische Bestimmung der Parameter  $f_{pp}$ ,  $f_{cu}$  und  $f_{pi}$  (von links nach rechts): Vergleich des durchschnittlichen Makespans von DMHEFT mit und ohne feste Zeitschranke für verschiedene Werte des zu optimierenden Parameters bei 8 Clustern und 100 Tasks pro DAG.

- $f_{pp}$ : Skalierungsfaktor für die Verzögerungszeit (*postponing factor*).
- $f_{pi}$ : Faktor zur Bestimmung der notwendigen Verbesserungszeit (*postponing improvement*).
- $f_{cu}$ : Faktor zum Anpassen des Anteils an freien Prozessoren auf einem Zielcluster (*cluster usage*).

Außerdem wurden drei Strategien innerhalb von DMHEFT getestet:

- 1) Ausschließliche Betrachtung der Kommunikationskosten bei der Wahl des Zielclusters (keine Verzögerung),
- 2) Verzögerung mit einer festen Zeitschranke und
- 3) Verzögerung ohne feste Schranke.

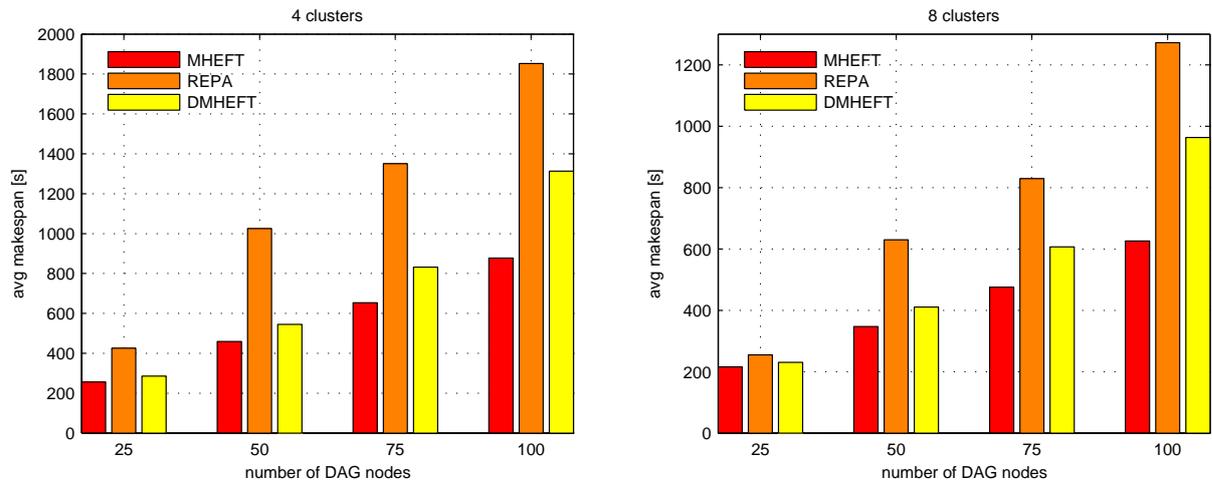
Um eine gute Wahl der Werte für diese Parameter empirisch zu ermitteln, wurde eine große Anzahl Simulationstests (mehr als 400 000) durchgeführt. Schon nach wenigen Tests war ersichtlich, dass Variante 1, die nur die Kommunikationskosten einbezieht, aber kein Postponing verwendet, hinter den anderen Varianten zurückbleibt. Deshalb beschränken sich die weiteren Analysen auf die Varianten 2 und 3. Für beide Strategien, mit und ohne explizite Verzögerungsschranke, wurde eine Serie von Experimenten mit unterschiedlichen Werten der zu bestimmenden Parameter sowie mit verschiedenen Konfigurationen aus DAGs und Plattformen durchgeführt. Die Bestimmung der optimalen Belegung der  $f$ -Parameter ist nicht möglich, da sie von dem jeweiligen DAG und der aktuellen Grid-Plattform abhängen. Es gilt deshalb, eine Belegung zu finden, die in den meisten Fällen gute Ergebnisse liefert. Aus diesem Grund werden die durchschnittlich erzielten Makespans für verschiedene Belegungen eines Parameters verglichen. Für den Vergleich der Makespans werden die Experimente mit 8 Clustern und 100 Tasks pro DAG ausgewählt, da kleinere DAGs sowie eine geringere Anzahl an Clustern weniger potenziell mögliche Task-Mappings aufweisen. Die Abb. 5.9 veranschaulicht die Simulationsergebnisse für verschiedene Werte der Parameter  $f_{pp}$ ,  $f_{cu}$  und  $f_{pi}$ . Alle drei Grafiken lassen erkennen, dass die Verwendung

einer festen Zeitschranke (*with limit*) zu längeren Makespans führt. Das ist ein überraschendes Ergebnis, da davon auszugehen ist, dass ein Verzögern einer kritischen Task – der mit dem größten Berechnungsaufwand – auch durchaus längere Laufzeiten nach sich ziehen kann. Das Anwenden einer Verzögerungsstrategie kann nicht ausschließen, dass die Situation zu einem späteren Zeitpunkt viel besser aussieht. Trotzdem hat sich in den Simulationen gezeigt, dass es sich lohnt, das Mapping einer Task zu verzögern.

Für jeden der drei Parameter wurden zuvor mögliche Werte festgelegt. Für  $f_{pp}$  und  $f_{cu}$  sind möglich:  $0.1, 0.2, \dots, 0.5$ . Im Fall von  $f_{pp}$  heißt das, dass ein Postponing bis zur Hälfte der aktuellen Beendigungszeit einer Task ( $f_{pp} = 0.5$ ) in Betracht gezogen wird. Bei  $f_{cu}$  bedeutet es hingegen, dass angenommen wird, dass bis zu 50 % der Prozessoren eines Clusters nach dem Postponing zur Verfügung stehen. Große Werte für beide Parameter wären möglich, aber nicht intuitiv zu rechtfertigen. Die Simulationsergebnisse zeigen, dass für  $f_{pp} = 0.2$  und  $f_{cu} = 0.4$  die kürzesten Makespans erreicht werden. Dies gilt für das Scheduling ohne feste Zeitschranke. Interessant ist dabei, dass die kürzesten Makespans dann resultieren, wenn  $f_{pp}$  relativ klein ist und keine Postponing-Zeitschranke eingesetzt wird. Im übertragenen Sinn heißt das, dass man beliebig oft verzögern kann, jedoch nicht allzu weit in die Zukunft schauen sollte. Als mögliche Werte für  $f_{pi}$  wurden  $0.5, 0.7$  und  $0.8$  getestet. Die Anwendung von  $f_{pi} = 0.5$  fordert, dass die Beendigungszeit nach dem Postponing mindestens halb so klein sein darf wie die aktuell abgeschätzte Beendigungszeit. Da dies eine sehr harte Forderung ist, zeigen die Simulationsergebnisse, dass  $f_{pi}$  relativ groß gewählt werden sollte. Die kürzesten Makespans wurden bei den Tests mit  $f_{pi} = 0.8$  erreicht. Da der erzielte durchschnittliche Makespan bei  $0.8$  nur etwas kleiner war als bei  $0.7$ , wurde  $0.8$  als Wert für  $f_{pi}$  festgelegt. Man könnte in weiteren Analysen auch noch den Wert von  $f_{pi}$  ermitteln, an dem die durchschnittlichen Makespans wieder anwachsen. Es wird bei größeren Werten von  $f_{pi}$  der Fall eintreten, dass sehr oft verzögert und wenig ausgeführt wird, was eine längere Ausführungszeit des gesamten DAGs nach sich zieht. Da, wie schon erwähnt, die Parameter von einander abhängen und es nicht möglich ist ein globales Optimum zu finden, werden die in den Tests empirisch ermittelten Werte für die weiteren Experimente verwendet. Diese sind:  $f_{pp} = 0.2$ ,  $f_{pi} = 0.8$  und  $f_{cu} = 0.4$ .

**Ergebnisse** Die Scheduling-Leistung von DMHEFT wurde mit denen von RePA und MHEFT verglichen. RePA wurde gewählt, da DMHEFT direkt auf diesem Verfahren aufbaut. MHEFT dient, wie bei der Analyse von RePA, dazu, die Leistung von DMHEFT in Bezug auf einen guten statischen Ansatz zu analysieren. Insgesamt wurden 5760 unterschiedliche Konfigurationen von DAGs und Grids getestet. Die Abb. 5.10 zeigt den durchschnittlichen Makespan, welcher von DMHEFT, RePA und MHEFT für 4 und 8 Cluster erzielt wurde. Die gemessenen Resultate für 4 und 8 Cluster sind sehr ähnlich, d. h. das Leistungsverhältnis der Algorithmen für die unterschiedlichen DAG-Größen ist vergleichbar. Im Mittel produziert MHEFT mit seinem statischen Ansatz die kürzeren Schedules. Interessanter ist aber der direkte Vergleich von RePA und DMHEFT. Dort erzielt DMHEFT eine signifikante Verbesserung gegenüber RePA mit einem um durchschnittlich 34 % besseren Makespan.

Die Tab. 5.2 bietet eine andere Sicht auf die Simulationsergebnisse, da die einzelnen Scheduling-Algorithmen paarweise gegenübergestellt werden. Es wird für jedes Paar die



**Abb. 5.10:** Durchschnittlicher Makespan von MHEFT, RePA und DMHEFT für 4 und 8 Cluster.

Anzahl der Experimente aufgeführt, in denen bessere, gleiche oder schlechtere Ergebnisse erzielt wurden. In der letzten Spalte wird angegeben, in wie viel Prozent der Fälle der jeweilige Algorithmus einen besseren, gleichen oder schlechteren Makespan erreicht hat. Der Algorithmus MHEFT produziert in 79 % aller Fälle einen kürzeren Schedule. Für die dynamischen Algorithmen RePA und DMHEFT bedeutet das aber auch, dass sie in ca. 20 % der Fälle schneller sind. Dies ist ein bemerkenswerter Prozentsatz, der vor allem durch DMHEFT bedingt ist. Vergleicht man nur die dynamischen Ansätze, lässt sich erkennen, dass DMHEFT die Leistung von RePA in vielen Fällen übertrifft (4042/5760). Die relativ hohe Zahl von kürzeren Schedules (1533), die RePA im Vergleich zu DMHEFT erreicht, ist durch die unterschiedliche Abbildungsstrategie der ersten Task auf einen Cluster bedingt. Ist ein Cluster am Anfang leer, versucht RePA alle Prozessoren einer Task zuzuweisen. Demgegenüber nimmt DMHEFT nur einen angemessenen (fairen) Anteil, was bei kleineren DAGs (25 Knoten) häufig zu einem größeren Kommunikationsoverhead führt. Dies könnte in einer hybriden Version verbessert werden, in der speziell für kleine DAGs eine andere Strategie benutzt wird. Im Vergleich zu MHEFT kann DMHEFT in 25 % der Testfälle einen kürzeren Schedule aufweisen. Dies ist besonders beachtenswert, da DMHEFT zukünftige Allokationen nur prognostizieren kann.

Eine weitere interessante Metrik ist die *Abweichung vom Besten* (degradation from best), welche eine qualitative Aussage über die Länge der Schedules zulässt. Dazu werden die relativen Zeitunterschiede zwischen dem Schedule eines Algorithmus und dem Schedule des jeweils besten Algorithmus pro Testfall summiert und gemittelt. Die resultierende Statistik *Abweichung vom Besten* ist für die 5760 Testfälle in Tab. 5.3 angegeben. Man kann in dieser Tabelle erkennen, dass die Schedules von MHEFT immer sehr nah am besten Schedule liegen, denn die durchschnittliche Abweichung beträgt nur ca. 5 %. Da die obere Statistik aber auch den Algorithmus bevorteilt, der öfters gewinnt, werden in der unteren Zeile nur die Fälle betrachtet, in denen der Algorithmus nicht den kürzesten Schedule erreicht. In der zweiten Zeile ist deshalb die Zahl der Testfälle aufgeführt, in denen der Algorithmus

**Tab. 5.2:** Paarweiser Vergleich der Scheduling-Algorithmen für alle 5760 Testfälle.

		MHEFT	REPA	DMHEFT	zusammen
MHEFT	besser		4932	4242	79.64 %
	gleich	xxx	26	54	0.69 %
	schlechter		802	1464	19.67 %
REPA	besser	802		1533	20.27 %
	gleich	26	xxx	185	1.83 %
	schlechter	4932		4042	77.90 %
DMHEFT	besser	1464	4042		47.80 %
	gleich	54	185	xxx	2.07 %
	schlechter	4242	1533		50.13 %

**Tab. 5.3:** Durchschnittliche Abweichung vom besten Algorithmus.

	MHEFT	REPA	DMHEFT
∅ über alle Exp.	5.2 %	98.7 %	36.2 %
# nicht bester	1888	5200	4338
∅ über # nicht bester	16.0 %	109.4 %	48.1 %

nicht der beste war. Daraus kann dann die relative Abweichung in diesen „nicht besten“ Fällen errechnet werden. Egal welche Art der Berechnung angewendet wird, produziert DMHEFT Schedules, die ca. 60 % besser sind als die von RePA. Mit dem Wissen, dass MHEFT oft den kürzesten Schedule erreicht, erlangt der Vergleich zwischen RePA und DMHEFT eine signifikante Bedeutung. Ein Schedule von DMHEFT ist im Mittel ungefähr 50 % langsamer als ein Compile-Zeit-Verfahren wie MHEFT. Im Gegensatz dazu ist der Makespan der Schedules von RePA zweimal so groß wie der beste Makespan. Für den Fall von dynamischen Taskgraphen lässt sich daraus schließen, dass DMHEFT die Qualität der Schedules gegenüber denen von RePA deutlich verbessert.

### 5.6.5 Fazit: Vergleich RePA und DMHEFT

Nach Analyse der experimentellen Ergebnisse wird eindeutig klar, dass DMHEFT die gesteckten Ziele erreicht hat. Durch Beachtung der Kommunikationskosten bei der Wahl des Zielclusters und durch Ausnutzung einer Verzögerungsstrategie (*postponing*) konnte der durchschnittliche Makespan im Vergleich zu RePA erheblich verbessert werden. Die Nachteile liegen dabei natürlich in der höheren Komplexität des Ansatzes (Berechnungszeit), was sich bei einer großen Anzahl von Tasks auswirken könnte. Andererseits beziehen sich die Resultate auf ein spezielles Modell zur Kommunikation und zur Verteilung von Matrizen, die als Kommunikationseinheiten betrachtet werden. Aus diesem Grund ist der Ansatz von RePA flexibler auf unterschiedlichste Szenarien übertragbar, z. B. auch wenn Tasks keine Matrizen oder Vektoren austauschen. Für wissenschaftliche Anwendungen mit dynamisch erzeugten Taskgraphen, die mit solchen Datenformaten arbeiten und schnell ausgeführt werden sollen, ist trotzdem DMHEFT die bessere Wahl.

## 5.7 Zweischritt-Scheduling mit spezieller Beachtung der Datenumverteilungskosten – RATS

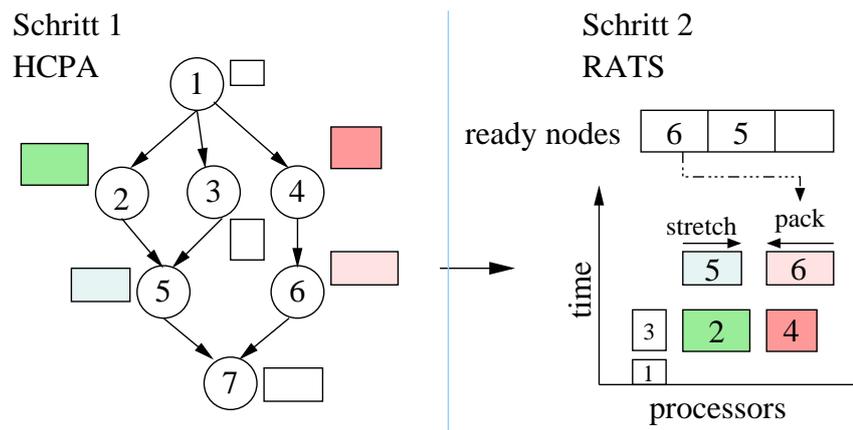
Nach eingehender Analyse der Compile-Zeit-Scheduling-Algorithmen und nach Betrachtung der Zusammenhänge zwischen Kommunikationszeit und Allokation, eröffneten sich Optimierungsmöglichkeiten bei zweistufigen Scheduling-Verfahren, die im Rahmen dieser Arbeit untersucht werden sollen.

In diesem Abschnitt wird ein neues zweistufiges Scheduling-Verfahren für einen homogenen Cluster vorgestellt, bei dem die im ersten Schritt ermittelten Allokationen im zweiten Schritt verändert werden können, um den Kommunikationsoverhead zu verringern. Zunächst werden verwandte Arbeiten betrachtet, bevor die Strategien des verbesserten Zweischritt-Verfahrens RATS (*Redistribution Aware Two step Scheduling algorithm*) erklärt werden. Da der neue Algorithmus sich der Funktion zur Bestimmung einer Allokation von HCPA bedient, werden die produzierten Schedules von RATS mit denen von HCPA in einer Vielzahl von Szenarien verglichen.

### 5.7.1 Motivation

Wie im Abschnitt über verwandte Arbeiten (Abschn. 5.3) beschrieben, arbeiten viele Algorithmen zum Scheduling von gemischt-parallelen Programmen auf homogenen Clustern in zwei Schritten [8, 93, 85, 87].

Ein wichtiger Algorithmus ist CPA (*Critical Path and Area-based scheduling*) [93], welcher versucht, den besten Kompromiss zwischen zwei Maßen zu finden. Das erste Maß ist die Länge des *kritischen Pfades*, also der Pfad im Taskgraph mit der maximalen Summe an Kanten- und Knotengewichten. Der kritische Pfad wird im Folgenden mit  $C_\infty$  bezeichnet. Das zweite Maß ist die Arbeit (Berechnungsfläche), die von einem Prozessor verrichtet wird. Die Gesamtarbeit ist definiert als  $W = \sum_{i=0}^N \omega_i$ , wobei  $\omega_i$  das Produkt aus Allokationsgröße und Ausführungszeit von Task  $t_i$  ist, was der von  $t_i$  belegte Fläche im Schedule entspricht. Um dieses Maß mit der Länge des kritischen Pfades in Beziehung zu setzen, wird die Arbeit (Fläche) pro Prozessor berechnet  $\bar{W} = \frac{1}{p}W$ . Der Algorithmus CPA beginnt dann, jeder Task genau einen Prozessor zuzuordnen. In diesem Fall ist  $C_\infty$  größer als  $\bar{W}$ . Danach erweitert CPA in jeder Iteration die Allokation einer Task des kritischen Pfades um genau einen Prozessor. Es wird die Task gewählt, deren Laufzeit am meisten profitiert. Die Allokationsphase endet, wenn  $C_\infty$  kleiner wird als  $\bar{W}$ . Der Fall  $C_\infty = \bar{W}$  stellt einen optimalen Kompromiss dar, da beide Maße untere Schranken für den Makespan sind. In Abhängigkeit der Charakteristika der Anwendung und der Plattform kann CPA zu sehr großen Allokationen führen, die eine gleichzeitige Ausführung mehrerer unabhängiger Tasks verhindern. Zwei andere Algorithmen versuchen dieses Problem zu lösen. MCPA [8] beschränkt die Allokationen so, dass alle Tasks einer Schicht gleichzeitig ausgeführt werden können. Diese Lösung ist deshalb nur für sehr regulär aufgebaute DAGs anwendbar. Der andere Algorithmus ist HCPA [75], welcher ursprünglich für heterogene Multicluster entworfen wurde. HCPA benutzt eine modifizierte Definition von  $\bar{W}$ , um die Allokationen bei einer großen Anzahl von verfügbaren Prozessoren zu limitieren [76]. Alle



**Abb. 5.11:** Grundlegende Arbeitsweise von RATS: Schritt 1 – Bestimmung der Allokationen aller Tasks, Schritt 2 – Mapping der Allokationen auf Prozessoren.

drei vorgestellten Algorithmen benutzen in der Mapping-Phase einen List-Scheduling-Ansatz, bei dem die Tasks nacheinander auf die Prozessoren abgebildet werden, wobei sie die Kommunikations- und Umverteilungskosten berücksichtigen. Die Reihenfolge der Tasks entspricht einer Sortierung nach abnehmendem Bottom-Level.

Da HCPA kürzere Schedules als CPA erzeugt und auf eine größere Klasse von Anwendungen als MCPA angewendet werden kann, wird HCPA als Ausgangspunkt für den hier vorgestellten Algorithmus (RATS) gewählt.

Das Hauptproblem der Zweischrittverfahren ist die entkoppelte Betrachtung der Bestimmung und des Mappings der Allokationen. Oft kommt es durch die getrennten Phasen bei der Auflösung der Datenabhängigkeiten zu zusätzlichen Umverteilungskosten und mehr Netzwerk-Contention. Im Rahmen dieser Arbeit wird angenommen, dass die *Nacheinanderausführung zweier abhängiger Tasks* auf einer *identischen Prozessormenge keine Datenumverteilung* erfordert. Damit ergibt sich die Hauptidee des neuen zweistufigen Scheduling-Algorithmus: In der Mapping-Phase werden die Allokationen erneut überprüft und ggf. verändert, um die Umverteilungskosten zu reduzieren und einen kleineren Makespan zu erzielen. Die vorgestellte Methode zum Mapping von Tasks arbeitet mit ausführbereiten Tasks, denn für diese Tasks wurden die Vorgängertasks schon im System ausgeführt und damit existiert für diese ein Mapping. Mit Hilfe der Mapping-Informationen der Vorgänger lässt sich die Bearbeitungszeit einer Task für verschiedene Mappings genau bestimmen.

Die Abb. 5.11 veranschaulicht die Idee und die Arbeitsweise des hier beschriebenen Algorithmus an einem Beispiel-DAG. Im ersten Schritt wird mit Hilfe der Allokationsfunktion von HCPA die Anzahl der Prozessoren für jede Task bestimmt. Diese Allokationen sind in der Abbildung als Rechtecke neben den Knoten dargestellt. In einem zweiten Schritt werden diese Allokationen auf die Prozessoren abgebildet. Dazu werden die Tasks nacheinander aus einer Prioritätsliste entnommen. In der Abbildung wurden die Tasks 1 bis 4 bereits ausgeführt. Um die Tasks 5 und 6 abzubilden, betrachtet der RATS-Algorithmus zunächst die den Elterntasks zugeordneten Prozessorgruppen. Im Beispiel werden für Knoten 5 die Prozessorgruppen von 2 und 3, und für Knoten 6 die Prozessorgruppe

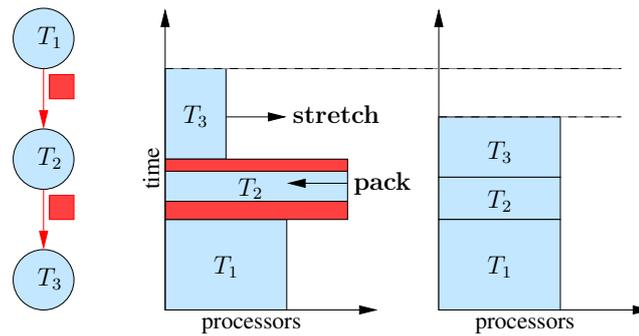


Abb. 5.12: Motivierendes Beispiel für RATS.

von Knoten 4 als mögliche Kandidaten ausgewählt. Der Algorithmus versucht nun, die ursprüngliche Allokationsgröße der Tasks 5 und 6 so anzupassen, dass eine Elternallokation wiederverwendet werden kann. Lässt sich eine Prozessorgruppe wiederverwenden, können Kommunikationskosten eingespart werden. Im Beispiel wird dazu die Allokation von Knoten 5 erweitert (*stretch*) und die von Knoten 6 verkleinert (*pack*).

### 5.7.2 Mapping unter Berücksichtigung der Datenumverteilung

In diesem Abschnitt werden die Strategien zur Veränderung der Allokation beschrieben und außerdem die Bedingungen erörtert, bei denen eine Modifikation in Frage kommt. Die beiden anwendbaren Strategien sind in Abb. 5.12 an einem Beispiel illustriert. Auf der linken Seite des Beispiels ist die Situation bei CPA und HCPA dargestellt. Nach Bestimmung der Allokation werden die Tasks nacheinander auf die Prozessoren abgebildet. In diesem Fall entstehen relativ hohe Umverteilungskosten, welche als Rechtecke zwischen den Tasks  $T_i$  gekennzeichnet sind. Ändert man die Allokationen so, dass die Allokation von  $T_2$  verkleinert (*pack*) und die von  $T_3$  vergrößert (*stretch*) wird, können die Umverteilungskosten (wie im rechten Bild zu sehen) merklich reduziert werden. Diese nahe liegenden Strategien erfordern aber eine genaue Untersuchung der Anwendbarkeit, welche in den folgenden Abschnitten gegeben wird.

#### Allokationen erweitern (*stretch*)

Das Erweitern von Allokationen kann zu einem doppelten Gewinn führen, da es Datenumverteilungskosten vermeiden und die Ausführung der Tasks beschleunigen kann. Der Preis für diese Operation ist eine höhere Ressourcenbelegung. Andererseits könnte eine Erweiterung auch eine gleichzeitige Ausführung anderer ausführbereiter Tasks verhindern. Letztendlich wird den Tasks von CPA oder HCPA, welche nicht auf dem kritischen Pfad liegen, aus gutem Grund eine kleine Allokation zugewiesen. Aus diesem Grund darf bei einer Änderung der Allokation die Größe nicht beliebig wachsen und muss deshalb beschränkt werden. Es werden zwei Strategien betrachtet:

***delta*** versucht, durch einer Erweiterung der Allokation Umverteilungen zu vermeiden.

***time-cost*** betrachtet die zusätzliche Arbeit, die durch das Erweitern verrichtet wird.

Es sei  $N_p(t)$  die Größe der Allokation einer Task  $t$ , die im Allokationsschritt des Algorithmus bestimmt wurde. Außerdem bezeichnet  $pred_i(t)$  die  $i$ -te Elterntask von Task  $t$ .

Die *delta*-Strategie realisiert die Idee, eine Task  $t$  mit Allokation  $N_p(t)$  auf die Elternallokation  $N_p(pred_i(t))$  abzubilden, die die geringste Modifikation der Allokationsgröße erfordert. Dazu wird ein  $\delta^+$  definiert, das die minimale Differenz zwischen der Allokationsgröße der Task  $t$  und aller Allokationsgrößen seiner Vorgänger bezeichnet. Die Variable  $\delta^+$  wird wie folgt berechnet:

$$\delta^+ = \min_i (N_p(pred_i(t)) - N_p(t)), \quad (5.18)$$

wobei nur die Vorgänger der Task  $t$  betrachtet werden, für die  $N_p(pred_i(t)) - N_p(t) \geq 0$  gilt. Die Allokationen anderer Elterntasks kommen für eine Vergrößerung der Allokation von  $t$  nicht in Betracht. Um das zu starke Anwachsen von Allokationen zu begrenzen, wird die Variable  $\delta_{max}$  eingeführt, die der maximalen Anzahl an Prozessoren entspricht, um die eine Allokation vergrößert werden kann. Da in die Berechnung von  $\delta_{max}$  die Größe der Ausgangsallokation eingehen soll, definiert der Parameter  $\mathbf{maxdelta} \in \mathbb{R}^+$  den Anteil, um den eine Allokation erweitert werden kann.

$$\delta_{max} = \lfloor \mathbf{maxdelta} \cdot N_p(t) \rfloor. \quad (5.19)$$

**Beispiel:** Seien  $N_p(t) = 6$  und  $\mathbf{maxdelta} = 0.5$ . Das bedeutet, dass die Allokation um 50 % vergrößert werden darf, also  $\delta_{max} = 0.5 \times 6 = 3$ . Deshalb kann die erweiterte Allokation auf maximal 9 Prozessoren abgebildet werden ( $6 + 3$ ).

Gemäß dieser Definition kann die Mapping-Prozedur der *delta*-Strategie für eine ausführbereite Task  $T$  angegeben werden:

1. Überprüfe, ob  $\delta^+ \leq \delta_{max}$ . Falls die Bedingung nicht erfüllt ist, gibt es keine Elterntask, die eine Vergrößerung der Allokation mit den Randbedingungen erlauben würde. Dann wird die Ursprungsallokation beibehalten.
2. Finde den ersten Vorgänger von  $t$ , für den  $\delta^+$  erfüllt ist.
3. Bilde  $t$  auf die Prozessoren des Vorgängers ab. Damit wird die Allokation auf die Größe des Vorgängers erweitert.

In der *time-cost*-Strategie wird das Verhältnis der Arbeit aus der ursprünglichen und einer neuen erweiterten Allokation (eines Vorgängers) gebildet:

$$\rho_i = \frac{T(t, N_p(t)) \times N_p(t)}{T(t, N_p(pred_i(t))) \times N_p(pred_i(t))}. \quad (5.20)$$

Dieser Wert soll dazu genutzt werden, die Allokation eines Vorgängers nur dann zu betrachten, wenn die verrichtete Arbeit mit beiden Allokationen annähernd gleich ist. Es soll verhindert werden, dass man sehr viele Prozessoren einsetzt, ohne dass ein nennenswerter Gewinn beim Makespan resultiert. Deshalb muss zusätzlich zu  $\rho_i$  eine Schranke  $\rho_{min}$  an die Mapping-Funktion übergeben werden. Der Parameter  $\rho_{min}$  liegt im Intervall  $]0 \dots 1]$  und bestimmt, welche Vorgänger für eine Allokationserweiterung in Frage kommen. Je mehr

sich  $\rho$  der Zahl 1 nähert, desto ausgeglichener ist das Verhältnis zwischen der Reduktion der Ausführungszeit und der Steigerung der zu verrichtenden Arbeit. Wird deshalb der Parameter  $\rho_{min}$  auf einen Wert nahe 1 gesetzt, heißt das, dass man für zusätzliche Ressourcen auch eine entsprechend hohe Laufzeitersparnis erwartet.

Mit dieser Definition kann auch für die *time-cost*-Strategie die Mapping-Funktion angegeben werden:

1. Finde die Vorgänger von  $t$ , welche ein maximales  $\rho_i$  aufweisen.
2. Überprüfe, ob  $\rho_i \geq \rho_{min}$ . Behalte die Ursprungsallokation, wenn die Bedingung nicht erfüllt ist.
3. Bilde  $t$  auf die Prozessoren eines Vorgängers mit minimalem  $\rho$  ab.

### Allokationen komprimieren (pack)

Das Komprimieren der Allokation einer Task erhöht deren Ausführungszeit, da die Zeit im Performance-Modell mit steigender Prozessoranzahl monoton fällt. Dabei kann es sein, dass der Nachteil einer solchen Komprimierung durch zwei Nebeneffekte so ausgeglichen wird, dass der Makespan der Anwendung trotzdem reduziert wird. Durch das Komprimieren einer Allokation kann es passieren, dass die Task eher gestartet werden kann, da auf weniger verfügbare Prozessoren gewartet werden muss. Andererseits kann eine kleinere Allokation mehr Raum für andere Tasks lassen, womit u. U. die verfügbare Taskparallelität besser ausgenutzt werden kann. Wie beim Erweitern von Allokationen werden die Strategien *delta* und *time-cost* benutzt, wobei kleine Änderungen vorgenommen werden.

Mit der *delta*-Strategie sind die Allokationen der Vorgänger nun größer als die der Task  $t$ . Deshalb wird jetzt

$$\delta^- = \max_i (N_p(pred_i(t)) - N_p(t)) \quad (5.21)$$

definiert. Außerdem wird eine untere Schranke  $\delta_{min}$  benötigt, die den minimal erlaubten Wert von  $\delta^-$  pro Task kennzeichnet. Der Wert von  $\delta_{min}$  lässt sich mit Hilfe des Faktors **mindelta**  $\in \mathbb{R}^-$  wie folgt bestimmen:

$$\delta_{min} = \lceil \mathbf{mindelta} \cdot N_p(t) \rceil. \quad (5.22)$$

**Beispiel:** Seien  $N_p(t) = 6$  und **mindelta** =  $-0.5$ , dann kann die Allokation höchstens auf 3 Prozessoren komprimiert werden ( $6 - 0.5 \times 6$ ), denn  $\delta_{min} = -3$ .

Die Mapping-Funktion für das Komprimieren mit der *delta*-Strategie ist dann wie folgt aufgebaut:

1. Überprüfe, ob  $\delta^- \geq \delta_{min}$  und wähle den ersten Vorgänger, der  $\delta^-$  erfüllt. Die Originalallokation bleibt erhalten, falls kein Vorgänger gefunden wird.
2. Mapping der Task  $t$  auf die Prozessoren des gewählten Vorgängers.

Bei der *time-cost*-Strategie wird eine Allokation nur dann komprimiert, wenn eine Task mit komprimierter Allokation eher beendet werden kann als mit der aktuellen Allokation. Dazu wird die Laufzeit der Task auf jeder Allokationen der Vorgänger abgeschätzt und mit der Laufzeit der Task mit der ursprünglichen Allokation verglichen. Es wird die Vorgängerallokation ausgesucht, die die Beendigungszeit am meisten reduziert.

### Strategien zum Sortieren der ausführbaren Tasks

Ein weiteres zu klärendes Problem ist die Frage nach der Reihenfolge, in der die ausführbaren Tasks in der Mapping-Phase betrachtet werden. Wenn eine Task ihre Ausführung beendet, können ein oder mehrere Kinder ausführbar sein. Es ist dann zu entscheiden, welche von diesen Tasks zuerst betrachtet wird. Da die unterschiedlichen Kandidaten mindestens einen gemeinsamen Vorgänger haben, kann die Modifikation einer Allokation einen negativen Einfluss auf die anderen Tasks haben. Es ist z. B. möglich, dass die Erweiterung einer Allokation zur Verzögerung von potenziell gleichzeitig ausführbaren Tasks führt, da nicht mehr genügend Ressourcen bereitstehen.

Wie in der motivierenden Betrachtung (Abschnitt 5.7.1) verdeutlicht, nutzen CPA und HCPA eine Sortierung der Tasks nach fallendem Bottom-Level. Diese Reihenfolge wird gewählt, da eine baldige Ausführung einer Task umso kritischer ist, je weiter die Task vom Endknoten der Applikation entfernt liegt. Der RATS-Algorithmus behält diese Ordnung der Tasks bei. Trotzdem kann eine sekundäre Sortierung angewendet werden, um Tasks mit gleichem Bottom-Level zu differenzieren. Diese zweite Sortierung muss mit einem stabilen Sortierverfahren implementiert werden, damit die primäre Reihenfolge nicht verändert wird. Für die Sortierung der ausführbaren Tasks werden zwei Methoden vorgeschlagen, die sich an der *delta*- und der *time-cost*-Strategie orientieren.

Die erste Methode benutzt die Parameter  $\delta^+$  und  $\delta^-$  aus dem vorherigen Abschnitt. Da  $\delta^+$  positive und  $\delta^-$  negative Werte annimmt, wird mit  $\delta(t)$  das betragsmäßige Minimum der beiden für Task  $t$  definiert:

$$\delta(t) = \min(\delta^+, -\delta^-). \quad (5.23)$$

Es wird hierbei versucht, den Tasks mit geringerem Modifikationsbedarf (bei der Allokation) eine höhere Priorität zuzuweisen. Diese Methode implementiert eine sekundäre Sortierung der Taskliste nach steigenden  $\delta(t)$ -Werten.

Die zweite Methode versucht eine Sortierung mit einem Kompromiss zwischen Zeit und Kosten (Ressourcen) ähnlich der *time-cost*-Strategie. Dazu wird für jede Task der *maximal* mögliche *Gewinn* in Bezug auf Beendigungszeit berechnet, der mit der Wiederverwendung einer Elternallokation erreichbar wäre. Dieser Gewinn (*gain*) wird dann wie folgt definiert:

$$gain(t) = \max_i (T(t, N_p(t)) - T(t, N_p(pred_i(t)))) . \quad (5.24)$$

Damit implementiert auch diese *time-cost*-Methode eine sekundäre Strategie zur Sortierung der ausführbaren Tasks nach steigendem *gain*( $t$ )-Wert.

### Beschreibung von RATS

Nach Definition und Erläuterung der verschiedenen Mapping- und Sortierstrategien kann die algorithmische Darstellung angegeben werden. Der Pseudocode des *Zweischritt-Scheduling-Algorithmus unter Beachtung der Datenumverteilung* (RATS) ist in Alg. 5.5 veranschaulicht. Der erste Schritt des Algorithmus ist die von HCPA entlehnte Methode zur Bestimmung der Allokationen der Tasks. Im zweiten Schritt, der Mapping-Phase, werden

**Algorithmus 5.5** RATS

---

```

1: compute allocation /* from HCPA */
2: while not all nodes scheduled do
3:   for each ready node do
4:     compute_strategy_parameters( ready node )
5:   sort ready nodes
6:   for each ready node  $t_r$  do
7:     if a parent allocation  $a_p$  matches (delta xor time-cost) condition then
8:       map node  $t_r$  onto parent allocation  $a_p$ 
9:       for each ready node  $t_o$  do
10:        if strategy parameter of  $t_o$  depends on  $a_p$  then
11:          compute_strategy_parameters(  $t_o$  )
12:        resort ready nodes if any  $t_o$  has been updated
13:      else
14:        map using HCPA
function compute_strategy_parameters( node  $t$  )
1: for each parent of  $t$  do
2:   if delta strategy then
3:     compute delta
4:   else
5:     estimate execution time on parent's allocation

```

---

die Tasks und deren Allokation auf die Prozessoren abgebildet. Dazu werden die  $\delta$ - oder  $\rho$ -Werte der ausführbaren Tasks je nach Strategie (*delta* oder *time-cost*) ermittelt. Diese Werte werden neben dem Bottom-Level verwendet, um die Liste nach Prioritäten zu sortieren. Der Algorithmus RATS entnimmt der Liste die ausführbare Task mit der höchsten Priorität und versucht eine bessere Allokation zu finden. Dazu werden nacheinander die Allokationen der beendeten Elterntasks getestet und mit den entsprechenden Kriterien verglichen (z. B.  $\rho_{min}$ ,  $\delta_{max}$ ). Wird eine bessere Allokation gefunden, kann diese der Task zugewiesen und auf die Prozessormenge dieses Vorgängers abgebildet werden. Da die Strategie-Parameter, z. B. die  $\delta$ -Werte, anderer ausführbarer Tasks von der gerade verwendeten Elternallokation abhängen, müssen die Parameter  $\delta$  und  $\rho$  dieser Tasks ggf. neu berechnet werden. Wird keine bessere Elternallokation gefunden, verwendet RATS die Mapping-Routine von HCPA, um die Allokation der Task auf den Cluster abzubilden. Dazu werden die ersten  $p_a$  freien Prozessoren im Cluster bestimmt, wobei  $p_a$  der Allokationsgröße der betrachteten Task entspricht.

### 5.7.3 Komplexitätsanalyse

Sei  $P$  die Anzahl der Prozessoren des betrachteten Clusters. Außerdem bezeichnet  $\xi$  die Komplexität, die Beendigungszeit einer Task abzuschätzen. Die Berechnungskomplexität des RATS-Algorithmus setzt sich aus der Komplexität des Allokations- und des Mapping-Schrittes zusammen. Die Allokationsfunktion von HCPA ist nur eine leicht modifizierte Variante von CPA, die das Anwachsen einer Allokation beschränkt. Um die Allokatio-

nen zu bestimmen, werden den Tasks nacheinander Prozessoren zugewiesen. Dazu muss der kritische Pfad berechnet werden (Aufwand  $O(N + E)$ ). Da jeder Task höchsten  $P$  Prozessoren zugewiesen werden können, muss der kritische Pfad für jede Task maximal  $P$ -mal aktualisiert werden. Da es  $N$  Tasks gibt, wird der kritische Pfad im schlechtesten Fall  $(P \cdot N)$ -mal berechnet. Die Komplexitätsanalyse der Allokationsfunktion ist detailliert in [93] beschrieben. Für die Komplexität der Allokationsphase von RATS ergibt sich damit:

$$O(N(N + E)P). \quad (5.25)$$

In der zweiten Phase wird für jede Allokation ein Mapping auf die Plattform erstellt. Der gesamte Code wird höchstens einmal für jede Task durchlaufen, da eine Task auch abgebildet wird, sobald sie ausführbar ist. Zur Abbildung einer ausführbaren Task werden die Parameter einer der beiden möglichen Strategien berechnet, siehe Alg. 5.5. Um die Parameter zu bestimmen, müssen im schlechtesten Fall alle Vorgänger der ausführbaren Knoten betrachtet und entweder die  $\delta$ -Werte ( $O(E)$ ) oder die Laufzeit ( $O(E\xi)$ ) berechnet werden. Danach werden die Tasks sortiert ( $O(N \log N)$ ), eine Task selektiert und abgebildet ( $O(P)$ ). Es kann passieren, dass die Parameter der anderen ausführbaren Tasks neu berechnet werden müssen. Für das Mapping mit HCPA müssen maximal  $P$  Prozessoren ausgewählt werden ( $O(P)$ ). Daraus folgt für die Komplexität der Mapping-Phase mit der *time-cost*-Strategie:

$$\begin{aligned} &O(N(2N \log N + 2(E\xi + P))) \\ &= O(N^2 \log N + NE\xi + NP), \end{aligned} \quad (5.26)$$

und mit der *delta*-Strategie:

$$\begin{aligned} &O(N(2N \log N + E + P)) \\ &= O(N^2 \log N + NE + NP). \end{aligned} \quad (5.27)$$

Die Gesamtkomplexität von RATS mit der *time-cost*-Strategie ist damit:

$$O(N(N + E)P + N^2 \log N + N^2 E\xi). \quad (5.28)$$

#### 5.7.4 Experimentelle Auswertung

Die Evaluation und der Vergleich von RATS zu anderen Heuristiken basiert wie bei RePA und DMHEFT auf Simulationsergebnissen. Wie zuvor mehrfach erwähnt, erlaubt es die Simulation, eine statistisch signifikante Anzahl von Experimenten für ein großes Spektrum von Applikation in angemessener Zeit durchzuführen. Der hierfür genutzte Simulator basiert wieder auf dem SimGrid Toolkit [24]. SimGrid bietet eine entsprechend gute Abstraktionsschicht, um mittels diskreter Simulation von Ereignissen parallele Applikationen in verteilten Umgebungen zu analysieren. Für die im Folgenden beschriebenen Experimente wurde SimGrid in der Version 3.3-r5344 eingesetzt.

**Tab. 5.4:** RATS: Eckdaten der betrachteten Cluster.

Cluster	Chti	Grelon	Grillon
# Prozessoren	20	120	47
GFLOPS	4.311	3.185	3.379

### Konfiguration der Testumgebung

Im Gegensatz zu den synthetisch erzeugten Plattformen bei DMHEFT werden für die experimentellen Tests drei Cluster der Grid'5000-Umgebung nachgebildet. Zwei dieser Cluster mit den Namen **Grillon** und **Grelon** gehören zum Gridpartner aus Nancy. Der dritte Cluster **Chti** steht in Lille. Alle Cluster haben eine ähnliche Gesamtleistung in FLOPS aber unterschiedliche Charakteristika, speziell in der Anzahl und Leistung der Prozessoren. Alle drei Cluster sind intern über ein Gigabit-Ethernet-Netzwerk verbunden (100  $\mu$ s Latenzzeit). Der **Grelon**-Cluster ist überdies auf fünf unterschiedliche Gehäuse verteilt, wobei sich in jedem Gehäuse 24 Knoten befinden. Demzufolge besitzt der Cluster ein hierarchisch aufgebautes Netzwerk. Die Tab. 5.4 gibt einen Überblick über die Prozessoren pro Cluster und die Gesamtleistung jedes Clusters in GFLOPS. Diese Werte wurden zuvor mit Hilfe des High-Performance Benchmarks *Linpack* unter Verwendung der AMD Core Math Library (ACML) ermittelt. Mehr Informationen über die Hardware der Cluster sind in Anhang B zu finden.

Die hier beschriebenen Strategien zur Verbesserung der Mapping-Phase versuchen, die Kosten der Datenumverteilung zu reduzieren. Deshalb ist es wichtig, die Details des Netzwerkmodells von SimGrid in die Betrachtungen einzubeziehen. SimGrid basiert auf einem beschränkten Multi-Port-Modell, d. h. ein Knoten kann gleichzeitig Daten von anderen Knoten senden und empfangen, wobei die Bandbreite des privaten Links auf die einzelnen Datenströme aufgeteilt wird. Jeder Netzwerklink wird im Modell mit einer Latenzzeit  $\lambda$  und einer Bandbreite  $\beta$  versehen. Um ein Gigabit-Netzwerk mit TCP genauer zu simulieren, verwendet SimGrid ein empirisches Modell zur Beschränkung der Bandbreite  $\beta' = \min(\beta, \frac{W_{max}}{RTT})$ , wobei  $W_{max}$  die maximale Fenstergröße von TCP (*window size*) und  $RTT$  die Round-Trip-Zeit zwischen zwei Knoten des Netzwerks bezeichnet. Im Fall von Multi-Hop-Verbindungen<sup>4</sup> ist  $RTT$  demnach doppelt so groß wie die Summe der Latenzzeiten der einzelnen Links. Das von SimGrid eingesetzte Modell zur gemeinsamen Nutzung von Netzwerk-Ressourcen basiert auf einer Max-Min-Fairness [17]. Bei einer Max-Min-Fairness wird zunächst die Gesamtbandbreite unter den verschiedenen Datenflüsse gleichmäßig verteilt. Benötigt ein Datenfluss weniger Bandbreite, als der zugewiesene Anteil, wird dieser Rest unter den anderen Datenflüssen gleichmäßig aufgeteilt. Diese Methode wird fortgeführt, bis die Bandbreite aller Datenflüsse maximiert wurde, wobei jeder Datenfluss nur seinen minimal benötigten Anteil beansprucht.

Ein quantitativer Vergleich zwischen den Kommunikationsmodellen von SimGrid und Möglichkeiten der Simulation auf Paketebene wird in [46] gegeben.

<sup>4</sup>Es sind mehrere „Sprünge“ zwischen den Computern notwendig.

**Tab. 5.5:** Parameter der randomisierten Generation der DAGs.

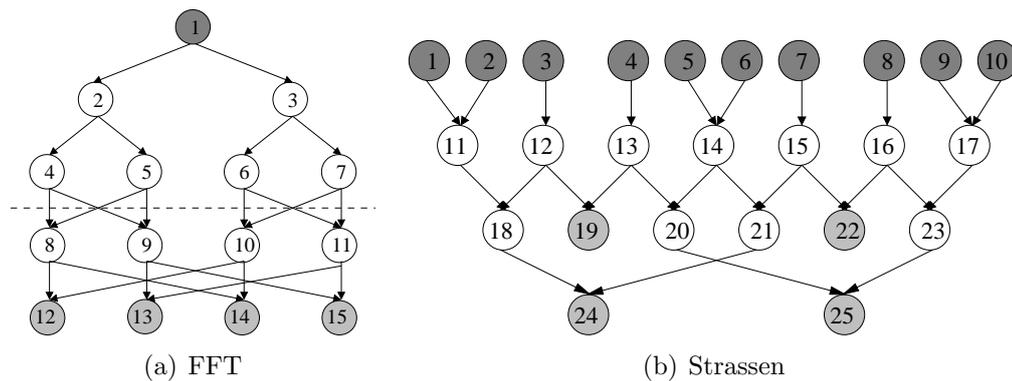
	geschichtet	irregulär
# Tasks	25, 50, 100	
nicht-parallelisierbarer Anteil ( $\alpha$ )	[0.0, 0.25]	
Breite ( <i>fat</i> )	0.2, 0.5, 0.8	0.2, 0.5, 0.8
Dichte	0.2, 0.8	0.2, 0.8
Regularität	0.2, 0.8	0.2, 0.8
Sprunglänge	-	1, 2, 4
#Muster	3	3
<b>Gesamt</b>	108	324

Es werden im Folgenden vier verschiedene Applikationsklassen betrachtet, um den Nutzen der Anpassung von Allokationen während der Mapping-Phase zu quantifizieren. Die Applikationen basieren auf dem in Abschnitt 5.4.2 definierten Applikationsmodell mit parallelen Tasks. Es werden zwei Arten von DAGs randomisiert erzeugt: geschichtete (*layered*) und irreguläre DAGs. In den hierzu generierten geschichteten DAGs besitzen alle Tasks einer Ebene dieselben Kosten. Demzufolge sind auch die Datentransferkosten (Kommunikationskosten) zwischen Knoten zweier Ebenen identisch. In einem irregulär aufgebauten DAG können Tasks derselben Ebene unterschiedliche Kosten aufweisen. Damit lassen sich die heterogenen und nicht vorhersagbaren Aspekte von wissenschaftlichen Workflows ausdrücken.

Für beide Arten von synthetischen Applikationen werden DAGs mit 25, 50 oder 100 M-Tasks (datenparallel) generiert. Durch die Änderung von drei Parametern lässt sich die Form der DAGs beim DAG-Generator variieren: Breite, Regularität und Dichte. Die Breite bezeichnet den Grad an Taskparallelität im DAG, was der Anzahl der Tasks in der größten Schicht entspricht. Ein kleiner Wert führt zu einer Kette von Tasks, während große Werte „fork-join“-Graphen erzeugen. Die Regularität definiert die Gleichmäßigkeit, mit denen die Tasks in den Ebenen verteilt werden, und mit der Dichte lässt sich die Verteilung der Kanten im DAG steuern. Alle drei Parameter können Werte zwischen 0 und 1 annehmen (siehe Anhang A.1). Die Tab. 5.5 enthält eine Übersicht über die verschiedenen Parameter und deren Werte, die zur Generierung der zufälligen Taskgraphen benutzt wurden. Da auch andere Werte bei der DAG-Generierung zufällig gewählt werden (z. B. der nicht-parallelisierbare Anteil  $\alpha$ , Datengröße), werden jeweils drei Muster pro Fall erzeugt.

Zusätzlich zu den zufällig generierten Taskgraphen wurden auch Graphen zweier High-Performance-Berechnungskernels betrachtet: (1) die Fast Fourier Transformation (FFT) und (2) die Matrixmultiplikation nach Strassen. Bei beiden Applikationen ist die Form der Taskgraphen fest definiert. Jedoch lassen sich die entsprechenden Gewichte (Kosten) der Knoten und Kanten verändern, wobei für die Tests die Gewichte zufällig gewählt wurden.

Die Abarbeitung des FFT-Taskgraphen kann in zwei Schritten betrachtet werden, die rekursiven Aufrufe und die Butterfly-Operationen [29]. Bei  $k$  Datenpunkten entstehen  $(2k - 1)$  Tasks für rekursive Aufrufe und  $k \cdot \log_2 k$  Tasks für die Butterfly-Operationen. Es



**Abb. 5.13:** Beispiel-DAGs für die Experimente. FFT-DAG mit  $k = 4$  und DAG für den Algorithmus von Strassen.

ist das Hauptmerkmal dieser FFT-DAGs, dass alle Pfade vom Startknoten zum Endknoten kritische Pfade sind, d. h. alle Knoten und Kanten haben pro Ebene die gleichen Kosten. Für die FFT-Tests wurde die Anzahl der Datenpunkte  $k$  auf der Menge  $\{2, 4, 8, 16\}$  so gewählt, dass FFT-Graphen mit 5, 15, 39 und 95 Tasks resultierten. Ein Beispiel eines FFT-DAGs für 4 Datenpunkte ist in Abb. 5.13(a) dargestellt. Die oberen beiden Ebenen repräsentieren die Tasks für die  $(2 \cdot 4 - 1) = 7$  rekursiven Aufrufe und die restlichen Knoten definieren die  $4 \cdot \log_2 4 = 8$  Butterfly-Tasks.

Als gemischt-parallele Variante der Matrixmultiplikation nach Strassen wird der in [35] vorgestellte DAG verwendet, welcher in Abb. 5.13(b) veranschaulicht ist. Der Strassen-DAG besteht aus insgesamt 25 Tasks. Die Knoten 11 bis 17 repräsentieren die sieben Matrixmultiplikationen der Untermatrizen und die restlichen Knoten repräsentieren die notwendigen Additionsoperationen, siehe Abschnitt 2.4.2. Beim Strassen-DAG sind die Tasks der ersten Ebene ähnlich kritisch, weisen also ein ähnliches Bottom-Level auf, da die Kosten der Pfade von der Matrixmultiplikation auf der zweiten Stufe dominiert wird und alle Knoten und Kanten einer Ebene dieselben Kosten aufweisen.

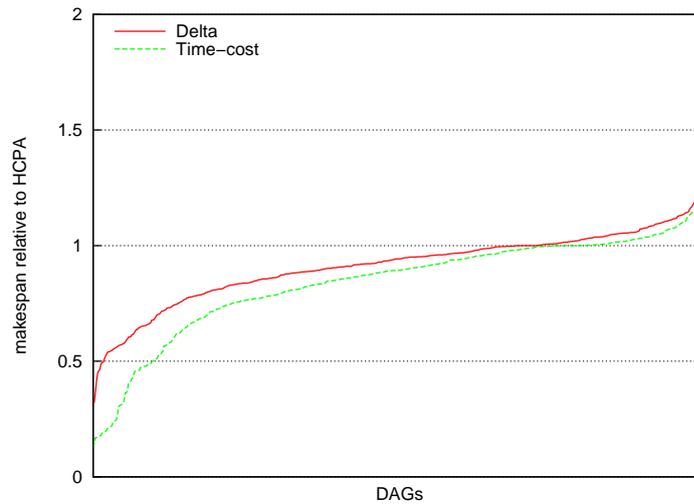
Für jede Kombination der Parametern wurden 25 Muster-DAGs generiert, die sich jeweils in den Kosten, aber nicht in der Form unterscheiden. Insgesamt stehen somit 100 FFT-DAGs und 25 Strassen-DAGs für die Experimente zur Verfügung.

### **Einfluss der Berücksichtigung der Umverteilungskosten auf die Scheduling-Leistung**

Die Leistung von RATS und damit der Einfluss der Berücksichtigung der Umverteilungskosten in der Mapping-Phase wird wie folgt untersucht: Für alle 557 Konfigurationen von Anwendungen wird der Schedule mit RATS und mit HCPA [76] bestimmt. Aus dem erzeugten Schedule werden der Makespan (kleinere Werte bedeutet bessere Leistung) und Gesamtarbeit (kleinere Werte bedeuten geringeren Ressourcenverbrauch) errechnet.

Zwei Varianten von RATS werden verglichen:

- Die *erste Variante* verwendet die *delta*-Strategie, welche versucht Umverteilungen zu vermeiden, indem sie Allokationen einer Task höchstens um den Betrag  $\delta$  ändert. Diese Version benutzt eine sekundäre Sortierung der Tasks nach aufsteigenden  $\delta$ -Werten.



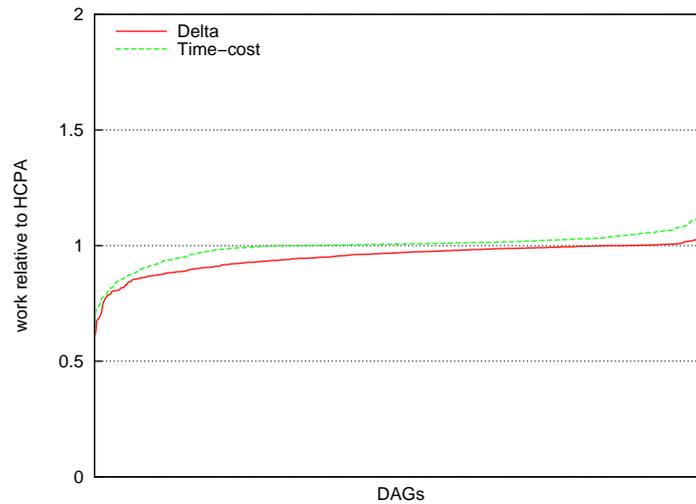
**Abb. 5.14:** Relativer Makespan von RATS mit den Strategien *delta* ( $\text{mindelta} = \text{maxdelta} = 0.5$ ) und *time-cost* (Komprimierung erlaubt und  $\text{minrho} = 0.5$ ) im Vergleich zu HCPA auf dem Grillon-Cluster.

- Die *zweite Variante* setzt auf der *time-cost*-Strategie auf. Dabei werden Allokationen nur vergrößert, wenn ein gutes Verhältnis zwischen Zeit und Arbeit erreicht werden kann. Die Komprimierung von Allokationen setzt voraus, dass die Task durch das Vermeiden von Kommunikationskosten eher beendet wird.

Für einen ersten Vergleich werden Standardwerte für jeden Parameter verwendet. Bei der *delta*-Strategie werden  $\text{mindelta}$  und  $\text{maxdelta}$  auf 0.5 gesetzt, was bedeutet, dass eine Task maximal 50% wachsen oder schrumpfen kann. Für die Untersuchung mit dem *time-cost*-Modell wurde  $\rho_{\min}$  ( $\text{minrho}$  in Abbildungen) ebenso 0.5 angenommen. Damit kann der Effizienzverlust beim Erweitern der Allokation höchstens 50% betragen.

Die Abb. 5.14 zeigt den auf Grillon gemessenen Makespan der beiden Strategien von RATS im relativen Vergleich zu dem Makespan von HCPA. Auf der x-Achse sind die im Experiment verwendeten DAGs aufgetragen. Dazu wurden die gemessenen Werte der relativen Makespans für jede Strategie aufsteigend sortiert. Im Diagramm ist demnach DAG  $d_i, 1 \leq i \leq 557$  der  $i$ -te Graph der sortierten Menge der relativen Makespans jeder Strategie. Es lässt sich erkennen, dass die Mehrheit der produzierten Makespans für beide Strategien von RATS deutlich kürzer ist als die von HCPA. Die *delta*-Strategie erzeugt dabei in 72% der Fälle kürzere Schedules als HCPA. Im Mittel über alle Szenarien liegen die Schedules von *delta* 9% unter denen von HCPA. Man kann auch erkennen, dass die *time-cost*-Strategie im Durchschnitt die besten Resultate erzielt, da sie ungefähr 16% kürzere Makespans produziert und insgesamt in 80% der Szenarien besser abschneidet. Bei der Auswertung konnten keine besonderen Tendenzen in Bezug auf die Applikationsklasse konstatiert werden. Darüber hinaus lieferten die Messungen mit den Clustern Chti und Grelon sehr ähnliche Ergebnisse.

Die Abb. 5.15 zeigt die Gesamtarbeit der Schedules von RATS im Verhältnis zu den Schedules, die durch HCPA auf Grillon produziert wurden. Die Grafiken lassen erkennen,



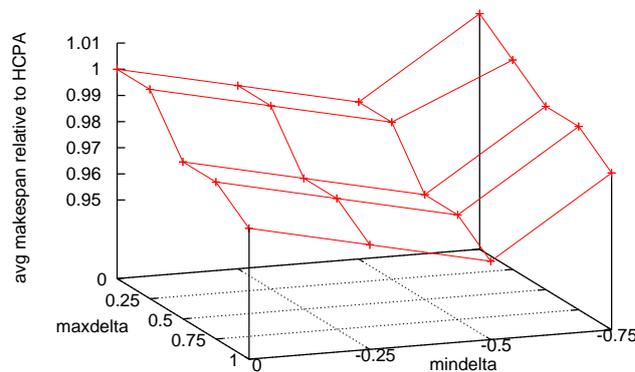
**Abb. 5.15:** Relative Arbeit von RATS mit den Strategien *delta* (`mindelta` = `maxdelta` = 0.5) und *time-cost* (Komprimierung erlaubt und `minrho` = 0.5) im Vergleich zu HCPA auf dem Grillon-Cluster.

dass beide Versionen von RATS nicht mehr Ressourcen verbrauchen als HCPA, meistens sogar weniger. Dabei wird auch deutlich, dass die *delta*-Strategie weniger Ressourcen benötigt als die *time-cost*-Variante. Das deckt sich mit den gemessenen Makespans aus der vorherigen Abbildung, denn dort war *time-cost* die bessere Strategie. Auch hier sind für die Cluster Chti und Grelon ähnliche Ergebnisse erzielt worden.

### Optimierung der Parameter $\delta$ und $\rho$

In den vorangegangenen Experimenten wurden nur Standardwerte zur Parametrisierung der beiden RATS-Varianten verwendet. In dem folgenden Abschnitt wird deshalb beschrieben, wie das Verhalten der beiden Versionen von RATS so geändert werden kann, dass sich Ressourcenbeanspruchung gegen Leistung eintauschen lässt. Das Hauptziel ist dabei, einen geringeren durchschnittlichen Makespan für die betrachteten Klassen gemischt-paralleler Programme zu erzielen.

Bei der *delta*-Strategie haben die beiden Parameter `mindelta` und `maxdelta` einen Einfluss auf die Modifizierbarkeit von Allokationen. Im letzten Abschnitt wurden die Werte auf 0.5 gesetzt. In den folgenden Betrachtungen wird versucht, ein Wertepaar zu ermitteln, das einen möglichst kleinen Makespan in Relation zu HCPA ermöglicht. Außerdem soll für jede Klasse von Applikationen und für jede Clusterkonfiguration ein gutes Wertepaar ausgesucht werden. Die Wertepaare wurden für RATS empirisch ermittelt. Die Abb. 5.16 zeigt die gemessenen relativen Makespans für die Klasse der FFT-DAGs auf dem Grillon-Cluster. Für beide Parameter wurden die folgenden Werte getestet (betragsmäßig): 0, 0.25, 0.5 und 0.75. Damit ist es erlaubt, dass Allokationen um 1/4, 1/2 oder 3/4 wachsen oder schrumpfen können. Für `maxdelta` wird auch noch der Wert 1 betrachtet, bei dem eine Allokation verdoppelt werden kann. Für `mindelta` kann  $-1$  nicht in Frage kommen, da einer Allokation nicht alle Prozessoren weggenommen werden sollen. Danach wird für jede Kombination von `mindelta` und `maxdelta` der zu HCPA relative Makespan ermittelt und aufgetragen.



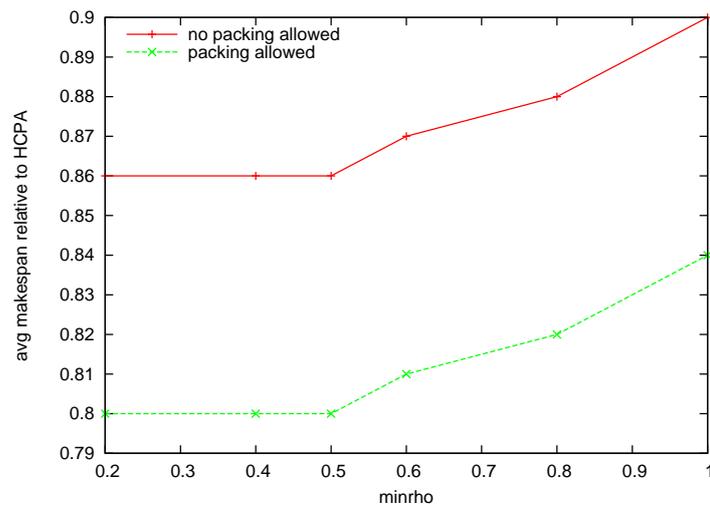
**Abb. 5.16:** Relativer Makespan von RATS mit der *delta*-Strategie für FFT-DAGs im Vergleich zu HCPA auf dem Grillon-Cluster für unterschiedliche Werte von *mindelta* und *maxdelta*.

**Tab. 5.6:** Ermittelte Parameterwerte für RATS (*mindelta*, *maxdelta*, *minrho*) in Abhängigkeit von der Applikationsklasse und der Plattform.

	FFT	Strassen	Layered	Random
Chti	(-.5, 1, .2)	(-.25, .5, .5)	(-.5, 1, .2)	(-.75, 1, .5)
Grillon	(-.5, 1, .2)	(0, 1, .4)	(-.25, 1, .2)	(-.75, 1, .5)
Grelon	(-.25, .75, .4)	(-.25, 1, .5)	(-.5, 1, .2)	(-.75, 1, .4)

In dem speziellen Beispiel in Abb. 5.16 kann man erkennen, dass das Erlauben von größeren Allokationen (durch größere Werte von *maxdelta*) auch zu kürzeren Makespans führt. Diese Erkenntnis kann leicht damit erklärt werden, dass eine Task bei Benutzung von mehr Ressourcen auch schneller ausgeführt werden kann. Interessanterweise wird der Schedule auch verbessert, wenn man den Wert von *mindelta* verkleinert. Dies liegt hauptsächlich daran, dass nun mehr Tasks parallel ausgeführt werden können und Umverteilungskosten wegfallen. Die optimalen Wertepaare für jeden Cluster sind nach Applikationsklasse in Tab. 5.6 aufgeführt. Dabei wird deutlich, dass der ermittelte Wert von *mindelta* zwischen den unterschiedlichen DAG-Klassen sehr stark schwankt.

Auch bei der *time-cost*-Strategie gibt es zwei zu optimierende Parameter. Die erste Parameter ist die boolesche Variable *packing*, die angibt, ob die Komprimierung von Allokationen erlaubt ist. Nach Analyse der Testergebnisse wurde festgestellt, dass die Möglichkeit zur Allokationskomprimierung immer bessere Resultate nach sich zieht. An dieser Stelle sollte nochmals daran erinnert werden, dass bei der *time-cost*-Strategie eine Allokation nur komprimiert wird, wenn sich die Beendigungszeit der entsprechenden Task reduziert. Wird der Mapping-Funktion die Möglichkeit gegeben Allokationen zu komprimieren, wird das kaum einen negativen Einfluss auf die Schedule-Länge haben.



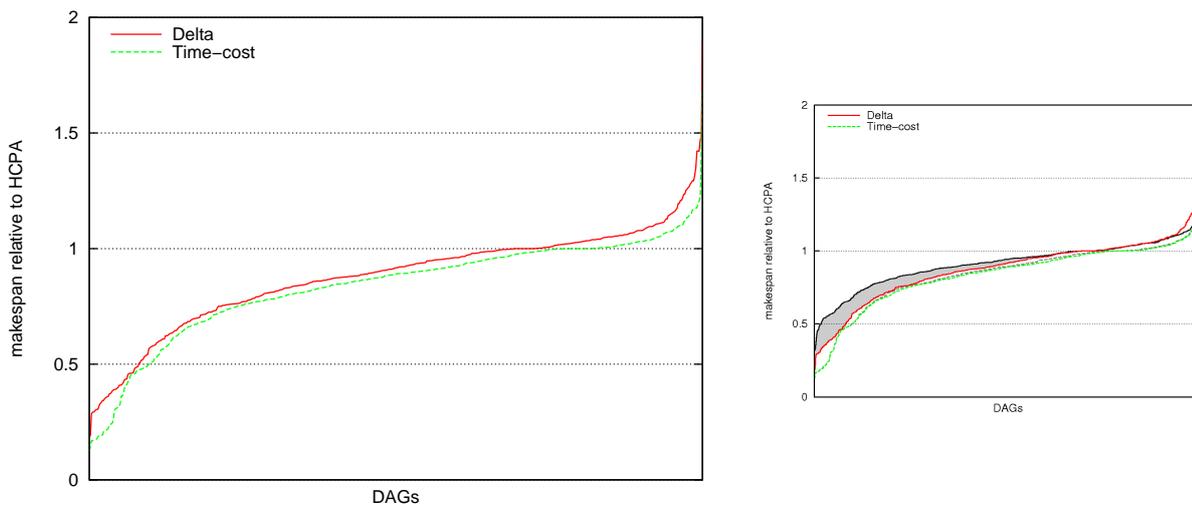
**Abb. 5.17:** Relativer Makespan von RATS für verschiedene Werte von `minrho` der *time-cost*-Strategie mit irregulären DAGs im Vergleich zu HCPA auf dem Grillon-Cluster.

Der zweite Parameter ist `minrho` und begrenzt die Möglichkeit zur Erweiterung von Allokationen. Ein kleiner Wert von `minrho`,  $0 \leq \text{minrho} \leq 1$ , bedeutet, dass man eine Allokation relativ stark erweitern kann, dafür aber nicht besonders viel an Laufzeit gewinnt. Wie bei der *delta*-Strategie wurde der Wert von `minrho` aus einer Liste von zuvor definierten Werten empirisch ermittelt (0.2, 0.4, 0.5, 0.6, 0.8, 1). Die Abb. 5.17 zeigt die für irreguläre DAGs gemessenen Makespans für die Liste der `minrho` auf dem Grillon-Cluster. Hier wird nochmals deutlich, dass die Möglichkeit zur Komprimierung immer zu besseren Schedules führt. Bei Betrachtung der Makespans für unterschiedliche Werte von `minrho` ist zu sehen, dass `minrho = 0.5` einen Grenzwert darstellt. Wird `minrho` weiter verkleinert, lässt sich kein Vorteil in der Laufzeit mehr feststellen.

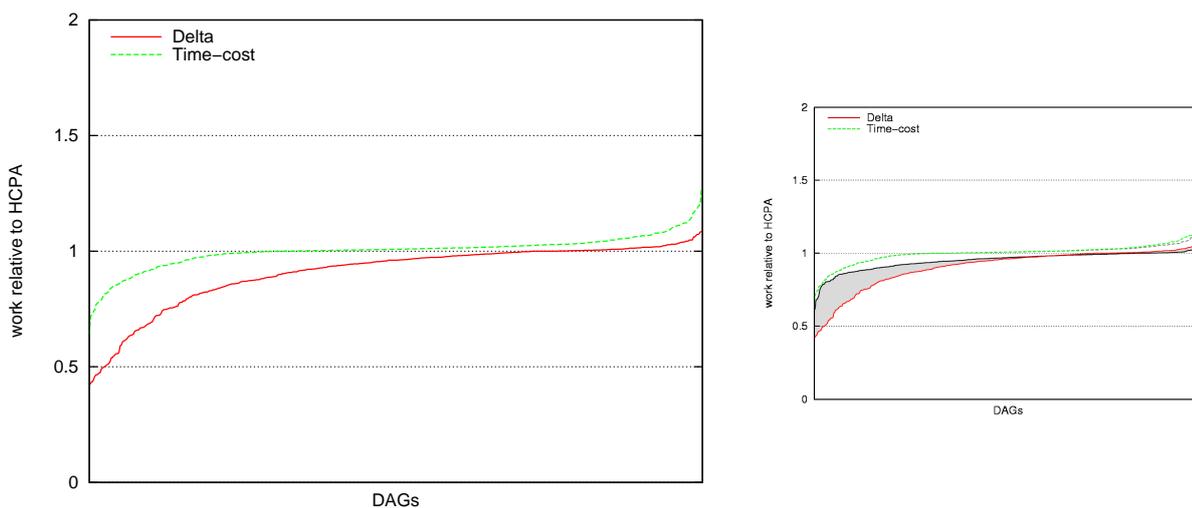
Auch die empirisch ermittelten Werte von `minrho` sind für jede Plattform und alle Applikationsklassen in Tab. 5.6 aufgelistet. Die darin dargestellten Werte der Parameter `mindelta`, `maxdelta` und `minrho` werden im Folgenden für die Untersuchungen und Vergleiche von RATS und HCPA verwendet.

### Vergleich der optimierten Version von RATS mit HCPA

Dieser Abschnitt vervollständigt die Studie, ob sich die Betrachtung der Datenverteilung und eine mögliche Änderung der Allokationen auf die Schedules verbessernd auswirkt. Als Erstes werden, wie in Abschnitt 5.7.4, die Schedules nach Makespan und Arbeit mit HCPA verglichen. Die Abb. 5.18 und 5.19 zeigen die mit den beiden Versionen von RATS erhaltenen Werte von Makespan und Arbeit für alle DAGs auf Grillon. Wieder wurden beide Datensätze unabhängig voneinander sortiert, d. h. es existieren auch Fälle, in denen die *delta*-Strategie einen kürzeren Schedule als die *time-cost*-Variante produziert. Auf der rechten Seite beider Abbildungen wurden die Messergebnisse mit den alten (schwarz) und den optimierten Parametern (farbig) aufgetragen. Die Grafiken zeigen, dass RATS mit der *time-cost*-Strategie nur wenig durch die Optimierung im Gegensatz zu vorher gewonnen hat.



**Abb. 5.18:** Relativer Makespan von RATS mit optimierten Parametern für die *delta*- und *time-cost*-Strategie im Vergleich zu HCPA auf dem Grillon-Cluster.



**Abb. 5.19:** Relative Arbeit von RATS mit optimierten Parametern für die *delta*- und *time-cost*-Strategie im Vergleich zu HCPA auf dem Grillon-Cluster.

Dies ist nachzuvollziehen, da die Komprimierung zuvor erlaubt war und mit `minrho = 0.5` schon ein Wert gewählt wurde, der ein sehr gutes Ergebnis in den meisten Szenarios lieferte. Der Einfluss der Optimierung der Parameter ist stärker bei der *delta*-Variante von RATS zu spüren. Der Unterschied bei der *delta*-Strategie vor und nach dem Tuning ist in den rechten Grafiken grau markiert. Beim Grillon-Cluster sind die durchschnittlichen Schedules nun 13 % kürzer als HCPA. Vorher waren es nur 9 %. Für die anderen Cluster (Chti und Grelon) konnte nach dem Tuning ein mittlerer Makespan erreicht werden, der 11 % (vorher 8 %) kürzer ist als der von HCPA.

Betrachtet man Abb. 5.19 genau, wird klar, dass die Verbesserung der Leistung nicht ausschließlich in einer höheren Ressourcenausnutzung begründet ist. Auch wenn Allokation

**Tab. 5.7:** Paarweiser Vergleich der Scheduling-Algorithmen. Jede Zelle gibt die Werte für die Cluster Chti / Grillon / Grelon an.

		HCPA	<i>delta</i>	<i>time-cost</i>	zusammen (in %)
HCPA	besser		154 / 133 / 161	103 / 88 / 82	23.1 / 19.8 / 21.8
	gleich	XXX	17 / 45 / 27	21 / 50 / 22	3.4 / 8.5 / 4.4
	schlechter		386 / 379 / 369	433 / 419 / 453	73.5 / 71.6 / 73.8
<i>delta</i>	besser	386 / 379 / 369		188 / 199 / 128	51.5 / 51.9 / 44.6
	gleich	17 / 45 / 27	XXX	49 / 90 / 40	5.9 / 12.1 / 6.0
	schlechter	154 / 133 / 161		320 / 268 / 389	42.5 / 36.0 / 49.4
<i>time-cost</i>	besser	433 / 419 / 453	320 / 268 / 389		67.6 / 61.7 / 75.6
	gleich	21 / 50 / 22	49 / 90 / 40	XXX	6.3 / 12.6 / 5.6
	schlechter	103 / 88 / 82	188 / 199 / 128		26.1 / 25.8 / 18.9

tionen stark erweitert werden können (da `maxdelta` oft sehr groß ist), verwendet die *delta*-Strategie erkennbar weniger Ressourcen als HCPA in der Mehrzahl der Fälle.

Zusätzlich zu den Betrachtungen der relativen Werte von Makespan und Arbeit, wurde auch für jede Konfiguration von DAG und Plattform das Verfahren bestimmt, das den besten Schedule liefert. Die Tab. 5.7 enthält für jeden Algorithmus (HCPA, *delta*, *time-cost*) die Anzahl der 557 Fälle, in denen er bessere, gleiche oder schlechtere Schedules als die Konkurrenten produzierte. Jede Zelle dieser Tabelle gibt dabei Auskunft über die Scheduling-Leistung auf den Clustern Chti, Grillon und Grelon (in dieser Reihenfolge, gleichbedeutend mit steigender Anzahl von Prozessoren). Die Zeile „zusammen“ verdeutlicht den Anteil der Szenarien, in welchen der jeweilige Algorithmus bessere (gleiche/schlechtere) Resultate erzielte als die anderen. Anhand der Anzahl der besseren Ergebnisse lässt sich eine Reihenfolge der Scheduling-Algorithmen aufstellen: *time-cost*, *delta*, HCPA. Diese Aussage wird durch die in Abb. 5.18 gezeigten Werte des relativen Makespans untermauert. Es ist überdies zu erkennen, dass die *time-cost*-Strategie auf größeren Clustern im Verhältnis besser abschneidet und dass die *delta*-Variante auf kleinen und mittleren Cluster ihre besten Ergebnisse erzielt. Dies kann dadurch erklärt werden, dass bei der Bestimmung der Umverteilungszeit mögliche Netzwerk-Contention nicht beachtet wird. Für eine Anwendung kann es deshalb auf einem kleineren Cluster eher zu Netzwerk-Contention kommen. Demzufolge basieren die Entscheidungen von RATS bei größeren Clustern auf genaueren Berechnungen. Eine weitere mögliche Erklärung steht in Zusammenhang mit den möglichen Zielallokationen für eine Task. Da die Entscheidungen der *delta*-Variante nicht auf Abschätzungen der Kommunikations- und Laufzeit beruhen, fällt der Fehler bei vielen verfügbaren Prozessoren auf einem großen Cluster höher aus. Die *time-cost*-Variante kann in diesem Fall aus mehreren Optionen die zeitgünstigste auswählen.

Um mehr Informationen über die Qualität der erzeugten Schedules zu erlangen, wird die prozentuale Abweichung vom besten Algorithmus bestimmt. Die Tab. 5.8 zeigt die Ergebnisse für zwei Arten der Berechnung dieser Abweichung vom besten Algorithmus. In der ersten Zeile wird die durchschnittliche Abweichung über alle Fälle gebildet (insgesamt 557). Die Abweichung ergibt sich für jedes Experiment aus der Differenz des Makespans des jeweils besten Algorithmus und des Makespans des aktuell betrachteten. Man kann erkennen, dass die Schedules der *time-cost*-Variante von RATS im Mittel 6% länger sind

**Tab. 5.8:** Durchschnittliche Abweichung vom besten Algorithmus.

		HCPA	<i>delta</i>	<i>time-cost</i>
Chti	∅ über alle Exp.	26.19 %	6.60 %	5.76 %
	# nicht bester	453	299	239
	∅ über # nicht bester	61.03 %	15.39 %	13.42 %
Grillon	∅ über alle Exp.	45.97 %	13.87 %	5.16 %
	# nicht bester	465	361	229
	∅ über # nicht bester	111.81 %	33.74 %	12.54 %
Grelon	∅ über alle Exp.	51.71 %	19.31 %	2.74 %
	# nicht bester	478	412	165
	∅ über # nicht bester	174.57 %	65.18 %	9.24 %

als die der besten Algorithmen. Dieser Prozentsatz verkleinert sich sogar noch mit steigender Clustergröße. Im Gegensatz dazu werden die Schedules mit der *delta*-Version länger (schlechter), wenn die Anzahl der Prozessoren pro Cluster wächst. Da diese Bestimmungsmethode den Algorithmus bevorteilt, der die meisten Fälle gewinnt (da die Abweichung oft 0 ist), wird noch die prozentuale Abweichung der Schedules für die nicht gewonnenen Fälle berechnet. In der zweiten Zeile der Tabelle ist für jeden Algorithmus die Anzahl der Fälle angegeben, die nicht gewonnen wurden. Die dritte Zeile enthält die damit berechneten Abweichungen. Die Abweichung bleibt für die *time-cost*-Strategie sehr klein (kleiner als 15 %), während sie bei HCPA sehr große Werte annimmt. Auf Grillon und Grelon sind die Schedules von HCPA im Mittel mehr als zweimal so lang wie die der Varianten von RATS.

### 5.7.5 Fazit RATS

In diesem Abschnitt wurde der Zweischritt-Scheduling-Algorithmus RATS (*Redistribution Aware Two step Scheduling algorithm*) vorgestellt, welcher versucht die Kosten für die Datenumverteilung in der Mapping-Phase einzubeziehen. Das Ziel ist die Minimierung der Kommunikationskosten, welche durch die Datenumverteilung zwischen Multiprozessor-Tasks bedingt sind. Aus diesem Grund können Allokationen in der Mapping-Phase bei gewissen Bedingungen so angepasst werden, dass die Umverteilungskosten sinken. Zwei parametrisierte Varianten von RATS wurden diskutiert und deren Anwendbarkeit durch Hilfe von Simulationen über einem großen Bereich von Applikationen und Plattformen getestet. Mit der ersten Strategie (*delta*) wird versucht, eine Datenumverteilung pro Task einzusparen, in dem eine Prozessorgruppe eines Vorgängers benutzt wird, die höchstens  $\delta$  Prozessoren mehr oder weniger als die vorgesehene Allokation aufweist. Bei der zweiten Strategie (*time-cost*) wird die Allokation einer Task dann verändert, wenn ein bestimmtes Zeit-Kosten-Verhältnis erreicht wird (natürlich soll primär die Beendigungszeit verkleinert werden). Die experimentellen Auswertungen haben ergeben, dass RATS mit beiden Strategien in den allermeisten Szenarien zu einem kürzeren Schedule führt als der Vergleichsalgorithmus HCPA. Als beste Strategie hat sich die *time-cost*-Variante herauskristallisiert, die auch dann gute Ergebnisse liefert, wenn sie nicht den besten Schedule produziert. Darüber hinaus konnte speziell für die *delta*-Strategie gezeigt werden, dass eine Anpassung der Parameter an die Charakteristika der Plattformen und Applikationen auch kürzere Schedules impliziert.



# Kapitel 6

## Zusammenfassung

*The hardest thing is to go to sleep at night, when there are so many urgent things needing to be done. A huge gap exists between what we know is possible with today's machines and what we have so far been able to finish.*

---

DONALD KNUTH

Das Hauptziel dieser Arbeit war, das Leistungspotenzial von *gemischt-parallelen Algorithmen* in *homogenen* und *heterogenen Umgebungen* zu testen und zu bewerten. Gemischt-parallele Programme bestehen aus taskparallelen sowie aus datenparallelen Anteilen. In dieser Arbeit stehen vor allem Programme im Vordergrund, die aus *Multiprozessor-Tasks* (M-Tasks) bestehen. Eine Multiprozessor-Task ist ein Programm oder Programmteil, welches mit einer variablen Anzahl von Prozessoren ausgeführt werden kann. Die M-Task kann dabei selbst hierarchisch aus M-Tasks aufgebaut sein oder eine datenparallele Implementierung aufweisen. Ein *gemischt-paralleles Programm* aus M-Tasks lässt sich mit Hilfe eines gerichteten azyklischen Graphen (*DAG*) beschreiben. Die Knoten des Graphen repräsentieren die M-Tasks und die Kanten bezeichnen die Kontroll- oder Datenabhängigkeiten zwischen den Tasks. Für eine *effiziente* und schnelle *Ausführung* eines M-Taskgraphen ist das *Scheduling* (Zuordnung von Ressourcen zu einer Task) von entscheidender Bedeutung.

Zuerst wurden exemplarisch gemischt-parallele Realisierungen der Matrixmultiplikation auf homogenen parallelen Systemen betrachtet. Für die Implementierung dieser Algorithmen konnte auf die C-Bibliothek `TLib` zurückgegriffen werden. `TLib` basiert auf MPI und bietet eine API, um gemischt-parallele Programme einfacher formulieren zu können. Mit Hilfe der `TLib` wurde die taskparallele Matrixmultiplikation (`tpMM`) entwickelt. Der `tpMM`-Algorithmus ist dabei rekursiv und hierarchisch durch M-Tasks formuliert. Die Eingabematrizen  $A$  und  $B$  sind blockweise (1D-Block-Verteilung) auf die Prozessoren verteilt. Der resultierende Algorithmus entspricht einer Sequenz von  $k$ -Panel-Updates ( $k$  ist die verkettete Dimension). Durch seine rekursive und hierarchische Struktur entsteht ein Berechnungsmuster, welches einem binären Baum gleicht. Dieses Muster lässt sich besonders gut auf SMP-, Multicore-Systemen oder Clustern aus Multiprozessoren einsetzen, da es die Datenlokalität ausnutzt. In Kombination mit Cache-optimierten Funktionskernen (DGEMM) für die lokale Matrixmultiplikation führt `tpMM` auf homogenen parallelen

SMP-Systemen zu einer kürzeren Laufzeit als andere vergleichbare Algorithmen, wie z. B. PDGEMM aus ScaLAPACK.

Außerdem wurde eine gemischt-parallele Variante der Matrixmultiplikation nach Strassen untersucht. Dieser Algorithmus bietet sich durch seine rekursive Formulierung sehr gut dazu an, einzelne Berechnungen herauszulösen und sie als Task zu formulieren. Im Gegensatz zu anderen taskparallelen Varianten von Strassens Algorithmus, bei denen die Tasks anhand der Matrixoperationen definiert wurden, kam eine ergebnisorientierte Aufteilung zum Einsatz. Jeder Berechnung der vier Teilmatrizen  $C_{ij}$ ,  $1 \leq i, j \leq 2$ , von Ergebnismatrix  $C$  wird eine Task zugewiesen. Damit werden pro Rekursionsstufe von Strassens Algorithmus vier Tasks erzeugt. Jede Task erhält bei der Ausführung einen Teil der zur Verfügung stehende Prozessoren des homogenen Systems. Für die Struktur der M-Tasks wurden zwei Schemata ohne und mit redundanten Berechnungen untersucht: Schema-Q7 und Schema-Q8. Diese Schemata unterscheiden sich in der Zuordnung der Berechnungen der Teilmatrizen  $Q_i$  und der dadurch notwendigen Zuteilung an Prozessoren. Die Rekursionstiefe der gemischt-parallelen Implementierung von Strassens Algorithmus ist an die Anzahl der Prozessoren gekoppelt. Zum Beispiel sind mindestens vier Prozessoren zur Anwendung einer Rekursion nötig. Nach Beendigung der Rekursion kommt ein weiterer Algorithmus zur Lösung des Teilproblems zur Anwendung. Je nachdem, wie viele Prozessoren bereitstehen und welches Datenformat gewählt wurde, stehen verschiedene Kandidaten zur Verfügung. Bei mehr als einem Prozessor können PDGEMM, Ring oder tpMM verwendet werden. Die lokalen Berechnungen auf einem Prozessor werden durch Cache-optimierte BLAS-Kernel wie z. B. ATLAS oder ESSL realisiert. Durch die hierarchische Anordnung von Algorithmen wurden, durch die Wahl der Verfahren auf jeder Stufe, so genannte Poly-Algorithmen erzeugt. Es wurden verschiedene algorithmische Kombinationen mit verschiedenen Zuordnungsschemata implementiert und getestet: *Strassen+Ring*, *Strassen+tpMM* und *Strassen+PDGEMM*. Die einzelnen Poly-Algorithmen haben dabei jeweils andere Anforderungen an das Datenlayout (blockverteilt / 1D-Block oder 2D-blockzyklisch). Je nach Datenlayout mussten andere Umverteilungsroutinen eingesetzt werden. Die Leistung der entstandenen Poly-Algorithmen wurde im Vergleich zu PDGEMM aus ScaLAPACK auf einer Vielzahl von verteilten Systemen evaluiert. Es hat sich dabei gezeigt, dass die Kombination aus *Strassen+Ring* im Schema-Q7 am besten abschneidet und in vielen Fällen zu einer kürzeren Laufzeit als PDGEMM führt.

Zu klären bleibt in diesem Zusammenhang, wie sich die Poly-Algorithmen adaptiv gestalten lassen. Dabei soll in Abhängigkeit von der Zielarchitektur, der Eingabegröße (Matrixdimension) und der gewählten Prozessoranzahl eine gute Wahl der Algorithmen in jeder Hierarchiestufe getroffen werden. Diese Entscheidungen müssen auf genauen Laufzeitabschätzungen basieren. Dazu können die in dieser Arbeit vorgestellten Formeln eingesetzt werden. Außerdem sind für die Abschätzungen Systemparameter wie Latenz oder Prozessorgeschwindigkeit notwendig. Die Bestimmung dieser Werte könnte durch den Einsatz von Hardware-Benchmarks erfolgen.

Bei der genauen Analyse der Laufzeittests von PDGEMM wurde deutlich, dass für einen exakten Leistungsvergleich mit den Poly-Algorithmen die Routine PDGEMM noch optimiert werden muss. Nach eingehender Parameteranalyse (Blockgrößen der verteilten Matrizen, Matrixdimensionen, Prozessorgitter, logische Blockgröße von ScaLAPACK) kris-

tallisierte sich die logische Blockgröße als entscheidender Faktor für eine hohe Leistung heraus. Die logische Blockgröße gibt in ScaLAPACK die Puffergröße vor, die zur parallelen Berechnung verwendet wird. Ist diese logische Blockgröße zu klein, können die Cachezeilen der Prozessoren nicht gefüllt werden, was einem Performance-Verlust im sequenziellen Fall (z. B. beim lokalen Matrixupdate) entspricht. Wählt man die logische Blockgröße zu groß, kann sich das Verhältnis von Berechnungs- und Kommunikationszeit so verschieben, dass eine Überlappung beider Phasen nicht mehr möglich ist. Demzufolge gibt es auch in diesem Fall Leistungseinbußen. In dieser Arbeit wird ein Verfahren angegeben, mit dem sich die Blockgröße automatisch auf einer Zielpattform optimieren lässt. Experimente haben gezeigt, dass die optimierte Variante von PDGEMM signifikant bessere Laufzeiten als die Standardroutine erzielt.

Die experimentellen Auswertungen der Laufzeiten von tpMM haben deutlich gemacht, dass die verfügbaren freien MPI-Bibliotheken oft nicht gut mit SMP- oder Multicore-Maschinen arbeiten. Die einzelnen MPI-Prozesse werden durch Betriebssystem-Prozesse implementiert. Dadurch erzwingt man teure Inter-Prozess-Kommunikation oder die Nutzung eines Netzwerk-Loopback-Treibers. In beiden Fällen kann die Leistung nicht ausgereizt werden. Aus diesem Grund wurde in dieser Arbeit die C++-Programmierungsumgebung vShark entwickelt, die auf MPI und POSIX-Threads aufsetzt und die Intra-Knotenkommunikation von verteilten Programmen optimieren soll. vShark bildet bewusst eine Schicht oberhalb von MPI, um die Bibliothek mit verschiedenen MPI-Implementierungen zu nutzen. Die virtuellen Prozessoren (ehemals MPI-Prozesse) werden durch leichtgewichtige Threads implementiert. Für die Intra-Knotenkommunikation zwischen zwei virtuellen Prozessoren werden dabei Daten direkt zwischen Threads ausgetauscht. Die Inter-Knotenkommunikation wird weiterhin mit MPI realisiert. Da die MPI-Bibliotheken oft nicht Thread-sicher (*thread safe*) sind, muss für die Inter-Knotenkommunikation mit vShark eine verminderte Leistung in Kauf genommen werden. Dieser Overhead wird durch einen Master-Thread bedingt, der für Thread-Sicherheit von vShark sorgt, aber zum Flaschenhals werden kann. vShark wurde speziell entwickelt, um Algorithmen mit hoher Datenlokalität einen Vorteil zu verschaffen. Deshalb schnitt die vShark-Variante von tpMM auf SMP-Clustern besser ab als die reine MPI-Implementierung.

Der erfolgreiche Einsatz von gemischt-parallelen Applikationen auf homogenen verteilten Systemen motivierte auch, die Ausführung dieser Applikationen auf heterogenen verteilten Systemen zu ermöglichen. Als heterogene Systeme werden hierfür Cluster betrachtet, die aus mehreren Clustern aufgebaut und geografisch verteilt sind. Von „traditionellem“ Grid-Computing (P2P-Netze) unterscheidet sich dieser Ansatz dadurch, dass die Co-Allokation von Ressourcen für ein gemischt-paralleles Programm erlaubt werden soll. Damit können mehrere geografisch verteilte Rechner Tasks eines einzigen Programms ausführen. Außerdem soll, im Gegensatz zu anderen Frameworks wie Condor + DAGMan, die Laufzeitumgebung die Ausführung von M-Tasks und die automatische Datenumverteilung zwischen den M-Tasks unterstützen. Da eine effiziente Ausführung von M-Tasks innerhalb eines homogenen Clustersystems sichergestellt werden kann, zielt diese Arbeit auf Multicluster (Cluster aus Clustern) ab. Diese Multicluster bestehen aus einer heterogenen Ansammlung homogen aufgebauter Cluster, wie z. B. Grid'5000. Die entwickelte Laufzeitumgebung TGrid implementiert diese Anforderungen. TGrid bietet zum einen eine Programmierungsumgebung zur

Beschreibung von gemischt-parallelen Algorithmen, zum anderen dient es auch als Grid-Middleware. Die Grid-Middleware muss dafür sorgen, dass geografisch verteilte Cluster miteinander verbunden werden können. Dazu unterstützt TGrid verschiedene Netzwerk-Protokolle (tcp, ssh), um Cluster einzubeziehen, die durch Firewalls geschützt und mit privaten IP-Adressräumen versehen sind. TGrid bietet eine plattform- und architekturunabhängige Programmierschnittstelle in Java, da Tasks in einer heterogenen Menge von Clustern abgearbeitet werden sollen. Ein Kernmodul beim Design von TGrid ist die Komponente zur automatischen Datenumverteilung. Durch Bereitstellung dieser Funktionalität wird die Programmierung von gemischt-parallelen Applikationen für den Entwickler deutlich erleichtert. Der Programmierer muss nur die Quell- und Zieltasks und die umzuverteilenden Daten angeben. Die Erstellung der Nachrichten und der Nachrichtentransfer wird dann von der Laufzeitumgebung erledigt und überwacht. Die Laufzeittests von gemischt-parallelen Programmen (Strassens Matrixmultiplikation, Berechnung eines Mandelbrot-Fraktals) in kleineren Testumgebungen haben das mögliche Leistungspotenzial bestätigt. Auch die Datenumverteilung innerhalb eines homogenen Clusters hat sehr gute Ergebnisse erzielt. Allerdings kann das System an einigen Stellen noch optimiert werden. Ein wichtiger Arbeitspunkt für zukünftige Versionen wäre die Unterstützung von paketbasierter Kommunikation. Da in TGrid eine Nachricht bei der Inter-Cluster-Kommunikation oft über mehrere Rechner geroutet werden muss, würde das die Kommunikationsleistung (Bandbreite und Latenz) signifikant verbessern. In der aktuellen Implementierung wird eine Nachricht erst weitergeschickt, wenn sie komplett von einem routendem Rechner empfangen wurde. Ein weiteres Implementierungsziel ist die Bereitstellung einer Komponente für den verteilten Datenzugriff (*data access*) ähnlich GridFTP. Eine weitere offene Frage bezieht sich auf die Realisierung der Ausfallsicherheit (*resilience*) des Laufzeitsystems und der ausgeführten Programme. Darüber hinaus gilt es auch, die Anwendbarkeit von TGrid mit wissenschaftlichen Applikationen in einer großen Grid-Konfiguration mit hunderten oder tausenden von Prozessoren zu testen.

Die effiziente Ausführung von gemischt-parallelen Applikationen in Multiclustern bedarf einer guten Scheduling-Strategie der ausführbereiten Tasks. Aus diesem Grund wurden in dieser Arbeit zwei Algorithmen zum Scheduling von dynamisch erzeugten M-Taskgraphen in Multiclustern entwickelt und evaluiert. Es handelt dabei um eine neue Klasse von Scheduling-Problemen, für die jedoch gängige Strategien wiederverwendet werden können. Die beiden vorgestellten Algorithmen, RePA und DMHEFT, treffen Scheduling-Entscheidungen unter Berücksichtigung der EFT (*earliest finish time*) und nutzen außerdem eine LPT-Strategie (*largest processing time*) zur Sortierung der Tasks. Im Vergleich beider ist der Algorithmus RePA das kostengünstigere Verfahren. Da er die Kommunikationszeit in der Mapping-Phase nicht einbezieht, benötigt er auch kein Modell zur Datenkommunikation. Deshalb stellt er wenige Bedingungen an das reale Grid-System und kann leichter implementiert werden. Die Hauptidee von RePA ist die Wiederverwendung von Prozessoren der Elterntasks, um die Kommunikationszeit zwischen kooperierenden Tasks zu reduzieren. Die experimentelle Analyse der Scheduling-Leistung wurde mit Hilfe eines Simulators durchgeführt, der auf dem SimGrid-Toolkit aufsetzt. Die Simulationen erlauben eine Rekonstruktion der Ergebnisse und unterliegen keinen großen Schwankungen wie in realen Umgebungen. Das Verfahren RePA wurde mit MHEFT verglichen, wobei es sich bei

MHEFT um einen Compile-Zeit-Algorithmus handelt. Der dynamische Aufbau der Taskgraphen wurde für RePA mit Hilfe der statischen DAGs simuliert. Im direkten Vergleich konnte RePA erstaunlich gute Makespans produzieren. Trotzdem gab es in einigen Szenarien Fälle, in denen das Fehlen einer Abschätzung der Umverteilungskosten beim Mapping einer Task sehr negativ ins Gewicht fiel. Um diese Nachteile zu beheben, wurde DMHEFT entwickelt. Dieser Algorithmus verwendet ein ähnliches Grundkonzept wie RePA, erweitert diesen aber um die Abschätzung von Kommunikationskosten und um eine Verzögerungsstrategie für große (berechnungsintensive) Tasks. Im anschließenden Vergleich beider Verfahren konnte DMHEFT die Scheduling-Leistung signifikant verbessern. Allerdings verlangt das Verfahren auch einen höheren Berechnungsaufwand und setzt ein gutes Modell zur Datenumverteilung voraus.

Da das Scheduling von M-Tasks in Multiclustern ein relativ neues Themengebiet ist, gibt es auch noch eine Menge offener Fragen. Genaue Aussagen über die Leistungsfähigkeit der Algorithmen können nur in realen Umgebungen gemacht werden. Dazu müssen die Verfahren in Umgebungen wie TGrid implementiert und getestet werden. Die vorgestellten Algorithmen nutzen viele Systemparameter zur Entscheidungsfindung, z. B. Prozessorgeschwindigkeit, Latenzzeiten und Bandbreiten. Dies sind noch relativ einfach zu ermittelnde Werte. Problematischer ist die Abschätzung der Laufzeit einer M-Task. Dazu können z. B. Micro-Benchmarks oder Performance-Modelle genutzt werden. Das Zusammentragen all dieser Informationen ist für die reale Einsetzbarkeit von großer Bedeutung. Außerdem ist es notwendig, verschiedene Modelle zur Bestimmung der Laufzeit einer M-Task (Amdahl, Downey) oder der Kosten einer Datenumverteilung mit Hilfe von Simulationen gegeneinander zu evaluieren.

Zur Entwicklung der Scheduling-Algorithmen für dynamisch generierte Taskgraphen wurden bestehende Verfahren für statische Taskgraphen analysiert. Eine Klasse dieser Scheduling-Algorithmen sind Zweischrittverfahren, bei denen im ersten Schritt die Allokation einer Task bestimmt und im zweiten diese Allokation auf die Prozessoren abgebildet wird. Ein Problem dabei ist die entkoppelte Betrachtung beider Schritte. Eine ungünstige Allokation kann zu schlechteren Kommunikations- und Laufzeiten führen. Um diesen Sachverhalt zu verbessern, wird in dieser Arbeit der Algorithmus RATS (*Redistribution Aware Two step Scheduling algorithm*) vorgestellt. Der Algorithmus RATS betrachtet zunächst homogene Cluster als Zielplattform und versucht durch Anpassung der Allokation im Mapping-Schritt, die Zeit für die Datenumverteilung zu reduzieren. Dazu werden die beiden Heuristiken *delta* und *time-cost* entwickelt, die eine Erweiterung (*stretch*) oder Komprimierung (*pack*) von Allokationen unter definierten Randbedingungen durchführen. Auch hier wurde durch Simulation mit SimGrid die Leistungsfähigkeit der RATS-Strategien im Vergleich zum Algorithmus HCPA in verschiedenen Experimenten verglichen. Die gemessenen Makespans waren im Mittel wesentlich kürzer als die von HCPA, obwohl auch die Gesamtarbeit im Durchschnitt nicht stieg. Ein nächster Schritt zur Weiterentwicklung von RATS wäre die Betrachtung von Multiclustern als Zielplattform. Außerdem könnten, wie bei den dynamischen Verfahren, andere Modelle zur Abschätzung der Laufzeit und der Umverteilungskosten getestet werden. Da RATS anwendungs- und plattformabhängige Parameter verwendet, ist es ein wichtiges Ziel, die Bestimmung automatisch an die Charakteristika der Maschinen und Applikationen anpassen zu können.



# Anhang A

## Werkzeuge zur Untersuchung von Scheduling- Algorithmen

### A.1 DAG-Generator

Gemischt-parallele Applikationen können durch gerichtete azyklische Graphen (DAGs) beschrieben werden. In den Graphen bezeichnen die Knoten die Multiprozessor-Tasks, und die Kanten definieren die Datenabhängigkeiten zwischen den Tasks.

Der DAG-Generator *daggen* (entwickelt von F. Suter [103]) erzeugt synthetische Taskgraphen, die zur Evaluation von Scheduling-Algorithmen im Simulator verwendet werden. Es ist dabei sehr wichtig, dass die vom Generator erstellten Taskgraphen möglichst viele Applikationsklassen abbilden. Der DAG-Generator besitzt diverse Parameter, damit die Struktur und die Form der zufällig erzeugten Taskgraphen beeinflusst werden kann. Diese Parameter sind:

- *Anzahl der Knoten* (Tasks) des Graphen.
- Die minimale und maximale *Datengröße*. Es wird davon ausgegangen, dass jede Task mit einer spezifischen Größe von Eingabedaten rechnet (z. B. ein Vektor oder eine Matrix). Auf diese Daten wird dann eine Funktion mit einer Berechnungskomplexität angewendet. Mit beiden ergibt sich dann die Anzahl der Operationen, die von der Task auszuführen sind.
- Der Parameter  $\alpha$  ( $0 \leq \alpha \leq 1$ ) aus *Amdahls* Gesetz.
- Der *Grad der Parallelität* ( $0 \leq fat \leq 1$ ) des Graphen. Damit wird die Anzahl der maximal parallel zu einander ausführbaren Tasks variiert (Tasks pro Ebene).
- Die *Dichte* der Abhängigkeiten ( $0 \leq d \leq 1$ ). Mit  $d = 0$  wird ein Graph mit minimaler Anzahl an Abhängigkeiten erzeugt, mit  $d = 1$  ein Graph, bei dem die Knoten einer Ebene mit allen Knoten der darunterliegenden Ebene verbunden sind.
- Die *Regularität* ( $0 \leq reg \leq 1$ ) des Aufbaus des Graphen. Mit diesem Parameter lässt sich die Anzahl der Tasks pro Ebene steuern. Die Regularität beim Aufbau schwankt zwischen irregulär ( $reg = 0$ ) und einer ausgewogenen Taskverteilung pro Ebene ( $reg = 1$ ).

- *Komplexität* einer Task. Damit lässt sich das Verhältnis von Kommunikation und Berechnung einstellen, denn die Kommunikation ist an das Datenvolumen (-größe) geknüpft. Der Berechnungsaufwand ist abhängig von der Funktion, die auf die Daten angewendet wird. Folgende Werte können für die Komplexität von Tasks gewählt werden:
  - $a \cdot n$  ( $a$  ist eine Konstante, die zufällig zwischen  $2^6$  und  $2^9$  gewählt wird), modelliert eine Bildverarbeitung eines Bildes der Größe  $\sqrt{n} \times \sqrt{n}$ ,
  - $a \cdot n \log n$ , z. B. für das Sortieren eines Feldes,
  - $n^{3/2}$ , repräsentiert z. B. eine Matrixmultiplikation,
  - zufällige Auswahl aus den verfügbaren Funktionen.
- Anzahl der *Ebenen* (*jump*), die *überspannt* werden dürfen. Durch die Angabe von *jump* = 1 verlaufen Kanten immer von Ebene  $l_i$  zu  $l_{i+1}$ . Ein Wert von *jump* > 1 erzeugt Sprünge innerhalb des DAGs, bei denen *jump* Ebenen übersprungen werden. Dadurch können Ausführungspfade mit unterschiedlicher Länge generiert werden.

## A.2 Grid-Generator

Mit dem Grid-Generator (ebenfalls von F. Suter) lassen sich Testplattformen zur Evaluation von Scheduling-Algorithmen erzeugen. Die mit Hilfe des Generators erstellten Plattform-Dateien dienen als Eingabe für SimGrid. Die generierten Grid-Umgebungen folgen dem in dieser Arbeit verwendeten Plattformmodell der heterogenen Multicluster, wobei jeder Multicluster aus homogenen Clustern aufgebaut ist.

Mit folgenden Parametern des Generator können die Charakteristika der Testplattformen variiert werden:

- Anzahl der Cluster.
- Anzahl der Hosts pro Cluster (minimal und maximal).
- *Geschwindigkeit* (*min\_speed*) des langsamsten Prozessors in FLOPS.
- Der *Heterogenitätsgrad* ( $h \geq 1$ ) ist ein Faktor, der zu *min\_speed* multipliziert wird und welcher damit die maximale Prozessorgeschwindigkeit innerhalb der Cluster definiert. Die Geschwindigkeiten der einzelnen Cluster wird zufällig aus dem Intervall von *min\_speed* bis  $h \cdot \text{min\_speed}$  gewählt.

## A.3 Jedule

Das Programm *Jedule* wurde im Rahmen dieser Arbeit als Hilfsmittel zur Untersuchung und Optimierung von Scheduling-Algorithmen entwickelt. Als Eingabe dient eine Schedule-Beschreibung, die man mittels Adapter an die eigenen Ausgabedateien anpassen kann. Die aktuelle Beschreibung liest die Simulator-Ausgabedatei und erzeugt daraus eine grafische

Darstellung des Schedules. *Jedule* kann insbesondere für die Entwicklung von Scheduling-Heuristiken für Multicluster eingesetzt werden, da es eine entsprechende Unterstützung für mehrere parallel arbeitende Cluster mitbringt. Damit lässt sich *Jedule* auch für andere Ablaufpläne aus dem Gridbereich anwenden.

In der Schedule-Ansicht (siehe Abb. A.1) kann für jede Maschine (Cluster) ein separater Schedule grafisch dargestellt werden. Die Typen der einzelnen Tasks (Berechnung, Kommunikation) lassen sich mit Farben attribuieren, so dass die gezeichneten Rechtecke schnell zugeordnet werden können. Alle Rechtecke besitzen eine Identifikationsnummer, die mit der Knotennummer im Taskgraph korrespondiert. Außerdem können Informationen wie Start- und Endzeit einer Task abgefragt werden.

*Jedule* unterstützt das Zooming im Schedule. Damit können kleine Abschnitte eines sehr langen Schedules detailliert angezeigt werden. Überdies können aktuelle Ansichten der Schedules abgespeichert werden. *Jedule* verfügt auch über eine Kommandozeilen-Schnittstelle, womit sich auch ohne die Benutzung der grafischen Oberfläche Bilder von Schedules generieren lassen (siehe Abb. A.2). Dies ist sehr nützlich, wenn man *Jedule* innerhalb einer Pipe zur Auswertung von Messungen benutzt. Der Entwickler hat damit die Möglichkeit, eine Vorschau aller Messungen zu erzeugen, um sich dann spezielle Experimente genau anzusehen.

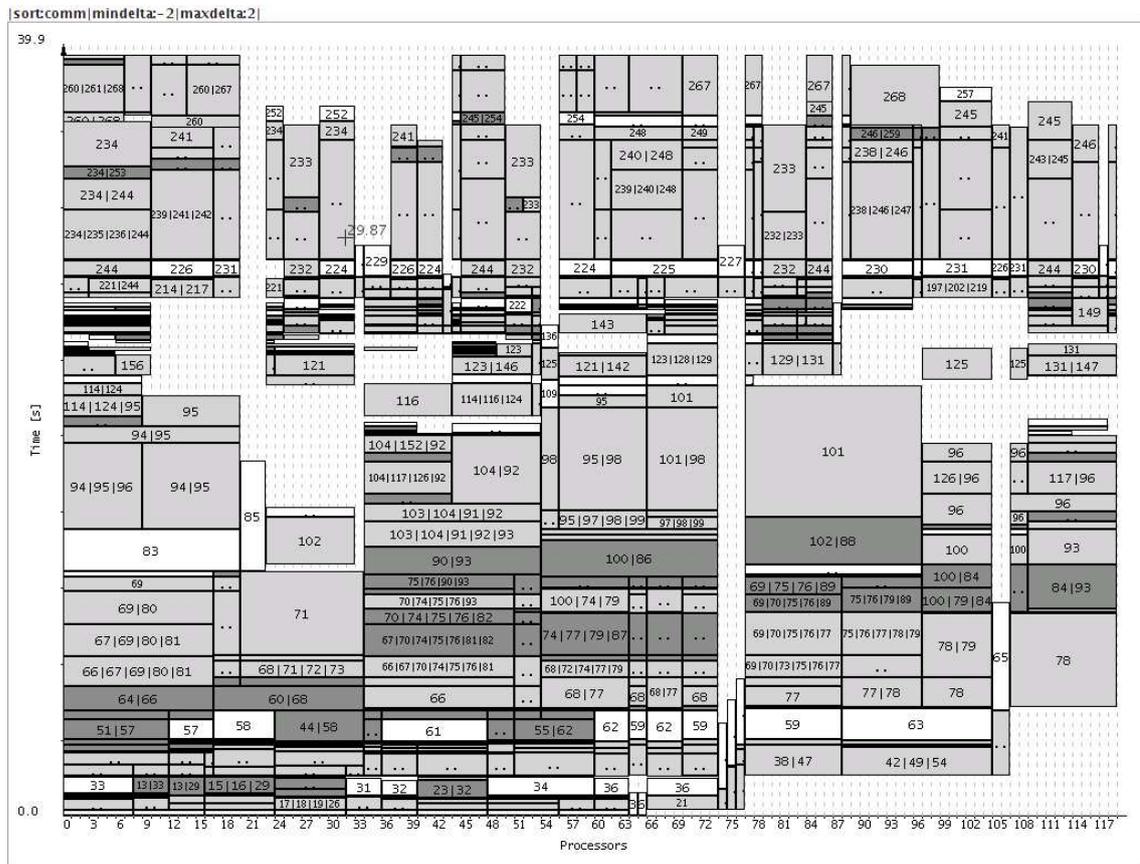
## A.4 mpiJava

Die aktuelle Implementierung von TGrid verwendet mpiJava<sup>1</sup>, um Multiprozessor-Tasks auf Clustern auszuführen. Die Software wurde bis zur Version 1.2.5 an der *Indiana University* entwickelt, wurde aber seit 2003 nicht mehr aktualisiert. Die Versionsreihe 1.2.x bietet die vollständige Umsetzung des MPI-1.1-Standards in Java. Da es keine reine Neuimplementierung ist, nutzt es das JNI (Java Native Interface), um auf native MPI-Bibliotheken wie MPICH oder LAM MPI (OpenMPI) zugreifen zu können.

Die Software wurde im Rahmen dieser Arbeit speziell für die x86\_64-Architektur angepasst, so dass auch auf diesen Maschinen mpiJava lauffähig ist. Außerdem wurde der gesamte Installationsprozess auf die autotools (automake, autoconf) umgestellt. Dadurch konnte eine bessere Portabilität und schnellere Installation ermöglicht werden. Zum Umfang dieser Arbeit gehört damit auch eine neue Version von mpiJava (Version 1.2.6-sahu).

---

<sup>1</sup><http://www.hpjava.org/mpiJava.html>



**Abb. A.1:** Beispielausgabe von *Jedule* für das Scheduling eines DAGs mit RATS (weiß = Rechentasks; hellgrau = Kommunikationszeit; dunkelgrau = Überlappung von Rechentasks mit Kommunikationszeit). Die Nummern in den Rechtecken bezeichnen Task- oder Kantennummern.

```
java -cp jedule-0.2.0.jar net.sf.jedule.JeduleStarter \
-p simgrid -f my_schedule_run.xml \
-d 800x300 -o exp1.png
```

**Abb. A.2:** Bilderzeugung mit *Jedule* aus der Kommandozeile.

## Anhang B

### Übersicht über die verwendeten Plattformen

#### JUMP

---

Bezeichner	IBM Regatta p690
Provider	NIC Jülich
Modell	IBM p690 (Regatta)
Prozessor	Power4+, 1.7 GHz
Cache (L1/L2/L3)	32 kB, / 1.5 MB / 32 MB
Prozessoren pro Knoten	32
Anzahl Knoten	41
Gesamtanzahl an Prozessoren	1312
Hauptspeicher	128 GB pro Knoten
Netzwerk	High Performance Switch Bandbreite $\geq 1.2$ GB/s pro Link Latenz $\leq 5\mu s$

---

#### LRZ SGI Altix

---

Bezeichner	SGI Altix 4700
Provider	LRZ München
Modell	SGI Altix 4700
Prozessor	Intel Itanium2, Dual Core, 2 GHz
Cache (L1/L2/L3)	16 kB / 256 kB / 9 MB
Prozessoren pro Knoten	512
Anzahl Knoten	19
Gesamtanzahl an Prozessoren	9728 (Cores)
Hauptspeicher	4 GB pro Core
Netzwerk	NUMAlink 4 Bandbreite 6.4 GB/s Latenz (MPI) 1-5 $\mu s$

---

## Opteron-Cluster

---

Bezeichner	Opteron-Cluster
Provider	Uni Bayreuth, AI2
Prozessor	Opteron 246, 2 GHz
Cache (L1/L2/L3)	64 kB / 1 MB / n/a
Prozessoren pro Knoten	2
Anzahl Knoten	32
Gesamtanzahl an Prozessoren	64
Hauptspeicher	4 GB pro Knoten
Netzwerk	Infiniband (10 GBit/s) Gigabit Ethernet (1 GBit/s)

---

## Xeon-Cluster

---

Bezeichner	XEON-SCI-Cluster
Provider	TU Chemnitz
Prozessor	Xeon 2.0 GHz, HyperThreading
Cache (L1/L2/L3)	8 kB / 512 kB / n/a
Prozessoren pro Knoten	2
Anzahl Knoten	16
Gesamtanzahl an Prozessoren	32
Hauptspeicher	1 GB pro Knoten
Netzwerk	SCI-Netzwerk, 2D-Torus (Dolphin SCI-Adapter D33x) Gigabit Ethernet (1 GBit/s)

---

## Chti

Quelle: <http://www.grid5000.fr>

---

Bezeichner	Chti
Provider	Lille
Modell	IBM e326m Monocore
Prozessor	AMD Opteron 252 2.6 GHz / 1 MB / 800 MHz
Prozessoren pro Knoten	2
Anzahl Knoten	20
Gesamtanzahl an Prozessoren	40
Hauptspeicher	4 GB
Netzwerk	Gigabit Ethernet

---

---

## Grelon

Quelle: <http://www.grid5000.fr>

---

Bezeichner	Grelon
Provider	Nancy
Modell	HP ProLiant DL140G3
Prozessor	Intel Xeon 5110 1.6 GHz / 4 MB L2 cache / 1333 MHz
Prozessoren pro Knoten	2
Anzahl Knoten	120
Gesamtanzahl an Prozessoren	240
Hauptspeicher	2 GB PC2-5300 DDR2-667
Netzwerk	Gigabit Ethernet (Broadcom BCM5721)

---

## Grillon

Quelle: <http://www.grid5000.fr>

---

Bezeichner	Grillon
Provider	Nancy
Modell	HP ProLiant DL145G2
Prozessor	AMD Opteron 246 2.0 GHz / 1 MB L2 cache / 400 MHz
Prozessoren pro Knoten	2
Anzahl Knoten	47
Gesamtanzahl an Prozessoren	94
Hauptspeicher	2 GB PC3200 DDR SDRAM
Netzwerk	Gigabit Ethernet (Broadcom BCM5721)

---



# Anhang C

## Zugehörige Publikationen

- ▷ R. Hoffmann, S. Hunold, M. Korch, and T. Rauber. Towards Scalable Parallel Numerical Algorithms and Dynamic Load Balancing Strategies. In *Proceedings of the Third Joint HLRB and KONWIHR Result and Reviewing Workshop 2007*, 2008.
- ▷ S. Hunold and T. Rauber. Automatic Tuning of PDGEMM Towards Optimal Performance. In *Proceedings of the Euro-Par Conference 2005*. Springer, Sept. 2005.
- ▷ S. Hunold and T. Rauber. Reducing the Overhead of Intra-Node Communication in Clusters of SMPs. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Processing and Applications (ISPA 2005)*. Springer, Nov. 2005.
- ▷ S. Hunold, T. Rauber, and G. Rünger. Hierarchical Matrix-Matrix Multiplication based on Multiprocessor Tasks. In *Proceedings of the International Conference on Computational Science ICCS 2004, Part II*, LNCS 3037, pages 1–8. Springer, 2004.
- ▷ S. Hunold, T. Rauber, and G. Rünger. Multilevel Hierarchical Matrix Multiplication on Clusters. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing, ICS'04*, pages 136–145, June 2004.
- ▷ S. Hunold, T. Rauber, and G. Rünger. Design and Evaluation of a Parallel Data Redistribution Component for TGrid. In *Proceedings of the International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, 2006.
- ▷ S. Hunold, T. Rauber, and G. Rünger. TGrid – Grid Runtime Support for Hierarchically Structured Task-parallel Programs. In *Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (Heteropar'06)*. IEEE Computer Society Press, 2006.
- ▷ S. Hunold, T. Rauber, and G. Rünger. Dynamic Scheduling of Multi-Processor Tasks on Clusters of Clusters. In *Proceedings of the Sixth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (Heteropar'07)*. IEEE Computer Society Press, 2007.
- ▷ S. Hunold, T. Rauber, and G. Rünger. Combining Building Blocks for Parallel Multi-level Matrix Multiplication. *Parallel Computing*, 34(6-8):411–426, 2008.
- ▷ S. Hunold, T. Rauber, and F. Suter. Redistribution Aware Two-Step Scheduling for Mixed-Parallel Applications. In *Proceedings of the 10th IEEE International Conference on Cluster Computing (Cluster 2008)*, 2008.

- ▷ S. Hunold, T. Rauber, and F. Suter. Scheduling Dynamic Workflows onto Clusters of Clusters using Postponing. In *Proceedings of the 3rd International Workshop on Workflow Systems in e-Science (WSES 08)*. IEEE Computer Society Press, 2008.

# Anhang D

## Danksagung

Ich möchte mich bei Herrn Prof. Dr. Thomas Rauber dafür bedanken, dass er mir die Möglichkeit eröffnet hat, eine Promotion abzulegen. Außerdem bin ich ihm sehr dankbar, dass er mir die Freiheit gab, meinen eigenen wissenschaftlichen Stil zu entwickeln.

Mein Dank gilt auch Frau Prof. Dr. Gudula Rüniger für die hilfreichen Anregungen, mit denen sie im Entstehungsprozess von Publikationen beigetragen hat.

Ich möchte mich ganz herzlich bei Dr. Matthias Korch, Björn Krellner und Thomas Reichel bedanken. Eure Genauigkeit und Euer Engagement beim Korrekturlesen dieser Arbeit waren toll!

Ich danke allen Mitarbeitern des Lehrstuhls für Angewandte Informatik II der Universität Bayreuth für die angenehme Zusammenarbeit in den letzten Jahren.

Der Forschungsaufenthalt am LORIA in Nancy war für mich ein sehr wichtiger Schritt auf dem Weg zur Promotion. Ich danke Dr. Emmanuel Jeannot und Prof. Dr. Jens Gustedt, dass sie mir diese Möglichkeit geboten haben.

Ein besonderer Dank geht an Dr. Frédéric Suter. Ich danke ihm für die unzähligen Diskussionen über M-Task-Scheduling und die Bereitstellung des Quellcodes seiner Scheduling-Algorithmen.

Ich möchte mich auch bei Dr. Fernando Tignetti für die Einladung und die Gastfreundschaft bedanken, die er mir in der Zeit an der Universität La Plata entgegenbrachte.

Einen großen Anteil am Gelingen dieser Arbeit haben meine Freunde und meine Familie. Ich danke Euch, Thorsten, Swenja, Georg und Stephan, für Eure Freundschaft, die mir immer wieder Kraft gab, weiter nach vorn zu gehen. Dir Andju, danke ich für Dein Verständnis und Deine Toleranz, die Du mir entgegengebracht hast und die mir in der letzten Phase meiner Arbeit sehr geholfen haben. Außerdem ist der SV Bayreuth zu meinem zweiten Zuhause geworden. Danke Stefan, Heike und Christof, dass Ihr mir immer wieder klargemacht habt, dass es auch ein Leben neben den Algorithmen gibt.

Vor allem möchte ich mich bei den wichtigsten Menschen in meinem Leben bedanken, die immer bedingungslos an mich geglaubt und mich jede Sekunde unterstützt haben: Danke Herbert, Ursel, Jürgen und Petra!



# Literaturverzeichnis

- [1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.
- [2] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.
- [3] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS 1967 Spring Joint Computer Conference*, volume 30, pages 483–485, Apr. 1967.
- [4] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 61–86. Springer, 2003.
- [5] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.
- [6] G. S. Baker, J. A. Gunnels, G. Morrow, B. Riviere, and R. A. van de Geijn. PLAPACK: High Performance through High-Level Abstraction. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP '98)*, page 414, Washington, DC, USA, 1998. IEEE Computer Society.
- [7] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.
- [8] S. Bansal, P. Kumar, and K. Singh. An Improved Two-Step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines. *Parallel Computing*, 32(10):759–774, 2006.
- [9] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix Multiplication on Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.
- [10] P. H. Beckman, P. K. Fasel, W. F. Humphrey, and S. M. Mniszewski. Efficient Coupling of Parallel Applications Using PAWS. In *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC '98)*, page 215, Washington, DC, USA, 1998. IEEE Computer Society.

- 
- [11] F. Bertrand and R. Bramley. DCA: A Distributed CCA Framework Based on MPI. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pages 90–97, Washington, DC, USA, Apr. 2004. IEEE Computer Society.
- [12] F. Bertrand, R. Bramley, K. B. Damevski, J. A. Kohl, D. E. Bernholdt, J. W. Larson, and A. Sussman. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, page 40b, 2005.
- [13] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the 11th International Conference on Supercomputing (ICS'97)*, pages 340–347, New York, NY, USA, 1997. ACM Press.
- [14] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users’ Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.
- [15] J. Blazewicz, M. Drozdowski, and M. Markiewicz. Divisible task scheduling - Concept and verification. *Parallel Computing*, 25(1):87–98, 1999.
- [16] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Iréa. Grid’5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Nov. 2006.
- [17] J.-Y. L. Boudec. Rate adaptation, Congestion Control and Fairness: A Tutorial. [http://ica1www.epfl.ch/PS\\_files/LEB3132.pdf](http://ica1www.epfl.ch/PS_files/LEB3132.pdf), Mar. 2008.
- [18] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC’00)*, CD-ROM, page 42, Washington, DC, USA, 2000. IEEE Computer Society.
- [19] R. Buyya, M. M. Murshed, D. Abramson, and S. Venugopal. Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost-time optimization algorithm. *Software - Practice And Experience*, 35(5):491–512, 2005.
- [20] E. Caron and F. Desprez. Diet: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.
- [21] E. Caron, F. Desprez, and F. Suter. Parallel Extension of a Dynamic Performance Forecasting Tool. *Scalable Computing: Practice and Experience*, 6(1):57–69, 2005.
- [22] E. Caron and G. Utard. On the performance of parallel factorization of out-of-core matrices. *Parallel Computing*, 30(3):357–375, 2004.

- [23] H. Casanova, F. Desprez, and F. Suter. From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Proceedings of the 10th International Euro-Par Conference (Euro-Par'04)*, volume 3149 of *LNCS*, pages 230–237. Springer, Sept. 2004.
- [24] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experimentations. In *Proceedings of Tenth International Conference on Computer Modeling and Simulation (UKSim)*, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society.
- [25] S. Chakrabarti, K. Yelick, and J. Demmel. Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs. *Journal of Parallel and Distributed Computing*, 47(2):168–184, 1997.
- [26] J. Choi. A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. *Concurrency: Practice and Experience*, 10(8):655–670, 1998.
- [27] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.
- [28] C.-C. Chou, Y.-F. Deng, G. Li, and Y. Wang. Parallelizing Strassen's Method for Matrix Multiplication on Distributed-Memory MIMD Architectures. *Computers and Mathematics with Applications*, 30(2):49–69, 1995.
- [29] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press/McGraw Hill, 1990.
- [30] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, Washington, DC, USA, 1999. IEEE Computer Society.
- [31] P. D'Alberto and A. Nicolau. Adaptive Strassen and ATLAS's DGEMM: A Fast Square-Matrix Multiply for Modern High-Performance Systems. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, pages 45–52, Washington, DC, USA, 2005. IEEE Computer Society.
- [32] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. C. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming*, 13(3):219–237, 2005.
- [33] E. Demaine. A Threads-Only MPI Implementation for the Development of Parallel Programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97)*, pages 153–163, 1997.
- [34] F. Desprez and E. Jeannot. Improving the GridRPC Model with Data Persistence and Redistribution. In *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 193–200, Washington, DC, USA, July 2004. IEEE Computer Society.

- [35] F. Desprez and F. Suter. Impact of Mixed-Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. *Concurrency & Computation, Practice & Experience*, 16(8):771–797, July 2004.
- [36] J. Dongarra, V. Eijkhout, and P. Luszczek. Recursive approach in sparse matrix LU factorization. *Scientific Programming*, 9(1):51–60, 2001.
- [37] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, Mar. 1990.
- [38] A. B. Downey. A Model For Speedup of Parallel Programs. Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, Jan. 1997.
- [39] M. Drozdowski. Scheduling multiprocessor tasks – An overview. *European Journal of Operational Research*, 94(2):215–230, Oct. 1996.
- [40] B. Dumitrescu, J.-L. Roch, and D. Trystram. Fast Matrix Multiplication Algorithms on MIMD Architectures. *Parallel Algorithms and Applications*, 4(2):53–70, 1994.
- [41] T. Eickermann, W. Frings, O. Wäldrich, P. Wieder, and W. Ziegler. Co-allocation of MPI Jobs with the VIOLA Grid MetaScheduling Framework. In *Proceedings of the German e-Science Conference 2007*. Max Planck Digital Library, May 2007.
- [42] A. Ferrari and V. S. Sunderam. Multiparadigm Distributed Computing with TPVM. *Concurrency: Practice and Experience*, 10(3):199–228, 1998.
- [43] I. Foster. The Grid: A New Infrastructure for 21st Century Science. <http://www.physicstoday.org>, 2002.
- [44] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
- [45] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3):200–222, 2001.
- [46] K. Fujiwara and H. Casanova. Speed and Accuracy of Network Simulation in the SimGrid Framework. In *First International Workshop on Network Simulation Tools (NSTools)*, Oct. 2007.
- [47] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996. Third Edition.
- [48] K. Goto and R. A. van de Geijn. Anatomy of a High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(3):1–25, 2008.
- [49] A. Grama, G. Karypis, A. Gupta, and V. Kumar. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley, 2003.

- 
- [50] B. Grayson, A. Shah, and R. van de Geijn. A High Performance Parallel Strassen Implementation. Technical Report CS-TR-95-24, Department of Computer Sciences, The University of Texas, 1, 1995.
- [51] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. A Flexible Class of Parallel Matrix Multiplication Algorithms. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium (IPPS '98)*, pages 110–116, Washington, DC, USA, 1998. IEEE Computer Society.
- [52] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings of the 1994 Conference on Supercomputing (SC'94)*, pages 350–359, Washington, DC, USA, 1994. IEEE Computer Society Press.
- [53] J. Hippold and G. Runger. A Communication API for Implementing Irregular Algorithms on SMP Clusters. In *Proceedings of the 10th EuroPVM/MPI 2003*, volume 2840 of *LNCS*, pages 455–463. Springer, 2003.
- [54] C. Huang, O. Lawlor, and L. V. Kale. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, volume 2958 of *LNCS*, pages 306–322. Springer, Oct. 2003.
- [55] F. Huet, D. Caromel, and H. E. Bal. A High Performance Java Middleware with a Real Application. In *Proceedings of the 2004 Conference on Supercomputing (SC'04)*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.
- [56] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna: A tool for building and running workflows of services. *Nucleic Acids Research*, 34 (Web Server Issue):W729–W732, 2006.
- [57] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson. Implementation of Strassen’s Algorithm for Matrix Multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC'96)*, CD-ROM, page 32, Washington, DC, USA, 1996. IEEE Computer Society.
- [58] E. Jeannot and F. Wagner. Messages Scheduling for Data Redistribution between Heterogeneous Clusters. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005)*, pages 119–124, 2005.
- [59] T. Johnson, T. A. Davis, and S. M. Hadfield. A Concurrent Dynamic Task Graph. *Parallel Computing*, 22(2):327–333, 1996.
- [60] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: a Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.
- [61] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Nov. 1998.

- [62] M. Krishnan and J. Nieplocha. SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, CD-ROM, page 70b, Washington, DC, USA, 2004. IEEE Computer Society.
- [63] S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, CD-ROM, pages 90–97, Washington, DC, USA, 2004. IEEE Computer Society.
- [64] J. Larson, R. Jacob, and E. Ong. The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. *International Journal of High Performance Computing Applications*, 19(3):277–292, 2005.
- [65] J.-Y. Lee and A. Sussman. High Performance Communication between Parallel Programs. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, page 177b, Washington, DC, USA, 2005. IEEE Computer Society.
- [66] A. Legrand, L. Marchal, and H. Casanova. Scheduling Distributed Applications: the Sim-Grid Simulation Framework. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGRID'03)*, pages 138–145, Washington, DC, USA, 2003. IEEE Computer Society.
- [67] R. Lepère, G. Mounié, D. Trystram, and B. Robić. Malleable tasks: An efficient model for solving actual parallel applications. In *Proceeding of the International Conference Parco'99*, pages 598–605, 1999.
- [68] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.
- [69] J. Li, A. Skjellum, and R. D. Falgout. A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies. *Concurrency: Practice and Experience*, 9(5):345–389, 1997.
- [70] Q. Luo and J. B. Drake. A Scalable Parallel Strassen's Matrix Multiplication Algorithm for Distributed-Memory Computers. In *Proceedings of the 1995 ACM symposium on Applied computing*, pages 221–226. ACM Press, 1995.
- [71] L. Masko, G. Mounie, D. Trystram, and M. Tudruj. Moldable Task Scheduling in Dynamic SMP Clusters with Communication on the Fly. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering (PARELEC '04)*, pages 59–64, Washington, DC, USA, 2004. IEEE Computer Society.
- [72] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, Sept. 2004.
- [73] J. McComb and S. Schmidt. Engineering and Scientific Subroutine Library for the IBM 3090 Vector Facility. *IBM Systems Journal*, 27(4):404–415, Nov. 1988.

- [74] W. Nasri and D. Trystram. A Poly-Algorithmic Approach Applied for Fast Matrix Multiplication on Clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, CD-ROM, page 234a, Washington, DC, USA, 2004. IEEE Computer Society.
- [75] T. N'Takpe and F. Suter. 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 (ICPADS 2006)*, pages 3–10, Washington, DC, USA, 2006. IEEE Computer Society.
- [76] T. N'Takpé, F. Suter, and H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *Proceedings of the 6th International Symposium on Parallel and Distributed Computing, Hagenberg, Austria*, page 35, Washington, DC, USA, 2007. IEEE Computer Society.
- [77] Y. Ohtaki, D. Takahashi, T. Boku, and M. Sato. Parallel Implementation of Strassen's Matrix Multiplication Algorithm for Heterogeneous Clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 112, 2004.
- [78] S. Pakin and A. Pant. VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management. In *Proceedings of the Workshop on Novel Uses of System Area Networks (SAN-1)*, Feb. 2002.
- [79] PARKBENCH Committee/Assembled by R. Hockney (Chairman) and M. Berry (Secretary). PARKBENCH report: Public international benchmarks for parallel computers. *Scientific Programming*, 3(2):101–146, Summer 1994.
- [80] A. P. Petitet. *Algorithmic Redistribution Methods for Block Cyclic Decompositions*. PhD thesis, University of Tennessee, Knoxville, 1996. Major Professor-Jack Dongarra.
- [81] P. Plaszczak and R. Wellner Jr. *Grid Computing: The Savvy Manager's Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [82] J. Porras, P. Huttunen, and J. Ikonen. The Effect of the 2nd Generation Clusters: Changes in the Parallel Programming Paradigms. In *Proceedings of the International Conference on Computational Science ICCS 2004, Part III*, volume 3037 of LNCS, pages 10–17. Springer, 2004.
- [83] R. Prodan and T. Fahringer. Dynamic Scheduling of Scientific Workflow Applications on the Grid: A Case Study. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05)*, pages 687–694, New York, NY, USA, 2005. ACM Press.
- [84] B. V. Protopopov and A. Skjellum. A Multithreaded Message Passing Interface (MPI) Architecture: Performance and Program Issues. *Journal of Parallel and Distributed Computing*, 61(4):449–466, 2001.
- [85] S. Ramaswamy. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Computations*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1996. Adviser-Prithviraj Banerjee.

- [86] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, 1997.
- [87] T. Rauber and G. Runger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.
- [88] T. Rauber and G. Runger. M-Task-Programming for Heterogeneous Systems and Grid Environments. In *Proceedings of the IPDPS Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models*, page 178b, Washington, DC, USA, 2005. IEEE Computer Society.
- [89] T. Rauber and G. Runger. Tlib - A Library to Support Programming with Hierarchical Multi-Processor Tasks. *Journal of Parallel and Distributed Computing*, 65(3):347–360, 2005.
- [90] T. Rauber and G. Runger. Anticipated Distributed Task Scheduling for Grid Environments. In *Proceedings of the IPDPS Workshop on High-Performance Grid Computing (HPGC)*, page 399, Washington, DC, USA, 2006. IEEE Computer Society.
- [91] T. Rauber and G. Runger. *Parallele Programmierung, 2. Auflage*. Springer Verlag, eXamen.press, 2007.
- [92] A. Radulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker. CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 39, Washington, DC, USA, 2001. IEEE Computer Society.
- [93] A. Radulescu and A. J. C. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *Proceedings of the 2001 International Conference on Parallel Processing (ICPP '02)*, pages 69–76, Washington, DC, USA, 2001. IEEE Computer Society.
- [94] J. Savant and S. Seidel. MuPC: A Run Time System for Unified Parallel C. Technical report, Department of Computer Science, Michigan Technological University, Sept. 2002.
- [95] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. A. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Proceedings of the Third International Workshop on Grid Computing (GRID '02)*, volume 2536 of *LNCS*, pages 274–278. Springer, 2002.
- [96] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. NetSolve: Grid Enabling Scientific Computing Environments. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*, pages 33–52. Elsevier, 2005.
- [97] Z. Shi. *Scheduling tasks with precedence constraints on heterogeneous distributed computing systems*. PhD thesis, University of Tennessee, Knoxville, 2006.
- [98] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley, 2007.

- 
- [99] F. Song, J. Dongarra, and S. Moore. Experiments with Strassen's Algorithm: from Sequential to Parallel. In *Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS06)*, 2006.
- [100] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
- [101] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–22, 1993.
- [102] Sun Microsystems Computer Company. Sun MPI 4.1 Programming and Reference Guide, Mar. 2000.
- [103] F. Suter. Dag generation program. <http://www.loria.fr/~suter/dags.html>.
- [104] F. Suter. Scheduling  $\Delta$ -Critical Tasks in Mixed-Parallel Applications on a National Grid. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing (GRID 2007)*, pages 2–9, Washington, DC, USA, 2007. IEEE Computer Society.
- [105] H. Tang and T. Yang. Optimizing Threaded MPI Execution on SMP Clusters. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*, pages 381–392. ACM Press, 2001.
- [106] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., Dec. 2002.
- [107] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.
- [108] R. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.
- [109] C. F. Van Loan. *Introduction to scientific computing: a matrix-vector approach using MATLAB*. Prentice-Hall, 1999.
- [110] R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Satin: Simple and Efficient Java-based Grid Programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, Sept. 2005.
- [111] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005.
- [112] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, 1997.
- [113] M. Wiczorek, A. Hoheisel, and R. Prodan. Taxonomy of the multi-criteria grid workflow scheduling problem. In *CoreGrid Workshop*, 2007.

- [114] J. Worringen and T. Bemberl. MPICH for SCI-connected clusters. In *Proceedings of SCI Europe '99 Conference*, pages 3–11, 1999.
- [115] J. Yu and R. Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming*, 14(3-4):217–230, 2006.
- [116] L. Zhang and M. Parashar. Enabling Efficient and Flexible Coupling of Parallel Scientific Applications. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 10, Washington, DC, USA, Apr. 2006. IEEE Computer Society Press.
- [117] H. Zhao and R. Sakellariou. Scheduling Multiple DAGs onto Heterogeneous Systems. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 14, Washington, DC, USA, 2006. IEEE Computer Society.