TECHNISCHE UNIVERSITÄT
CHEMNITZ

Entwicklung effizienter gemischt paralleler Anwendungen

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

Vorgelegt von
Diplom-Informatiker Jörg Dümmler,
geboren am 17. September 1976 in Erfurt.

Tag der Einreichung: 8. April 2010
Tag der Verteidigung: 7. Juli 2010

Gutachter:
Prof. Dr. G. Rünger
Fakultät für Informatik
TU Chemnitz

Prof. Dr. A. Goerdt
Fakultät für Informatik
TU Chemnitz

Inhaltsverzeichnis

1. Einleitung **1**

2. Das CM-task Programmiermodell **5**
2.1. CM-task Programme 6
2.2. Werkzeugunterstützung für CM-task Programme 8
2.3. Programmiermodelle für gemischt parallele Anwendungen 12
2.3.1. Spracherweiterungen 12
2.3.2. Skeletonbasierte Ansätze 15
2.3.3. Bibliotheksbasierte Ansätze 17
2.3.4. Koordinationsbasierte Ansätze 19

3. Scheduling von CM-task Programmen **23**
3.1. Das CM-task Schedulingproblem 23
3.2. CM-task Schedulingalgorithmus 25
3.2.1. Transformation des CM-task Graphen 26
3.2.2. Lastbalancierung für Supertasks 27
3.2.3. Kosten des Supertask Graphen 28
3.2.4. Scheduling des Supertask Graphen 29
3.2.5. Gruppenausgleich 30
3.2.6. Erzeugung eines CM-task Schedules 31
3.3. Experimentelle Auswertung 32
3.4. Zusammenfassung und verwandte Arbeiten 34

4. Sprachen und Schnittstellen des CM-task Compilerframeworks **37**
4.1. Spezifikationssprache 37
4.1.1. Konstanten- und Datentypdefinitionen 37
4.1.2. Datenverteilungstypdefinitionen 38
4.1.3. Basismoduldefinitionen 39
4.1.4. Verbundmoduldefinitionen 40
4.1.5. Beispiel: Iterierte Runge-Kutta Verfahren 47
4.2. Kostenmodell und Plattformbeschreibung 51
4.3. Erweiterte Spezifikationssprache 53
4.4. Rahmenprogramm 57
4.5. Erweitertes Rahmenprogramm 60
4.6. Koordinationsprogramm 63
4.6.1. Koordination im statischen Compileransatz 64
4.6.2. Koordination im semi-dynamischen Compileransatz 68
4.7. Datenumverteilungsbibliothek 70
4.8. Lastausgleichsbibliothek 74
4.9. Implementierung der Basismodule 76
4.10. Zusammenfassung und verwandte Arbeiten 78

5. Realisierung des CM-task Compilerframeworks **81**
5.1. Analysephase des CM-task Compilers . . . 81
5.1.1. Festlegen eindeutiger Identifikatoren . . . 82
5.1.2. Bestimmung der Kommunikationsabhängigkeiten . . . 83
5.1.3. Bestimmung der Datenabhängigkeiten . . . 85
5.2. Schedulingphase des CM-task Compilers . . . 90
5.2.1. Konstruktion einer Hierarchie von CM-task Graphen . . . 90
5.2.2. Hierarchisches Scheduling der CM-task Anwendung . . . 93
5.2.3. Konstruktion der Syntaxbäume des Rahmenprogrammes . . . 97
5.2.4. Übersetzung der Prozessorgruppenannotationen . . . 98
5.2.5. Schleifenrekonstruktion . . . 99
5.2.6. Annotation der Konstruktdaten . . . 102
5.2.7. Annotation der Daten- und Kommunikationsabhängigkeiten . . . 103
5.3. Datenverteilungsphase des CM-task Compilers . . . 104
5.3.1. Bestimmung der Datenverteilungen für Schleifen und Bedingungen . . . 105
5.3.2. Einfügen der Datenumverteilungsoperationen . . . 107
5.3.3. Annotation von Lastausgleichsoperationen . . . 109
5.4. Codegenerierungsphase des CM-task Compilers . . . 110
5.4.1. Codegenerator für statische Koordinationsprogramme . . . 110
5.4.2. Codegenerator für semi-dynamische Koordinationsprogramme . . . 110
5.5. Datenumverteilungsbibliothek . . . 113
5.6. Lastausgleichsbibliothek . . . 114
5.7. Zusammenfassung und verwandte Arbeiten . . . 116

6. Anwendungen und Experimente **119**
6.1. Parallele Plattformen . . . 119
6.2. Anwendungsprogramme . . . 120
6.2.1. Löser für Systeme gewöhnlicher Differentialgleichungen . . . 121
6.2.2. NAS-MZ Benchmarks . . . 123
6.3. Auswertung der Experimente . . . 125
6.3.1. Evaluierung des statischen Compileransatzes . . . 125
6.3.2. Evaluierung semi-dynamischer Koordinationsprogramme . . . 134
6.3.3. Mappingstrategien für Multicore-Prozessoren . . . 136
6.4. Zusammenfassung . . . 141

7. Zusammenfassung **143**

A. Nutzerschnittstelle des CM-task Compilers **147**

B. Syntax der Sprachen des CM-task Compilerframeworks **151**
B.1. Spezifikationssprache . . . 151
B.2. Erweiterte Spezifikationssprache . . . 154
B.3. Syntax der Rahmenprogramme . . . 155
B.4. Syntax erweiterter Rahmenprogramme . . . 156
B.5. Plattformbeschreibungssprache . . . 157

C. Schnittstellen und Datentypen des CM-task Compilerframeworks **159**
C.1. Schnittstellen zur Einbindung von Nutzerdatentypen 159
C.2. Schnittstellen und Datentypen der Datenumverteilungsbibliothek 159
C.2.1. Statische Datenumverteilungsoperationen 159
C.2.2. Dynamische Datenumverteilungsoperationen 161
C.2.3. Nutzerdefinierte Datenumverteilungsoperationen 163
C.3. Datenstrukturen und Schnittstellen der Lastausgleichsbibliothek 164

D. Anwendungsspezifikationen **167**

Literaturverzeichnis **171**

Index **179**

Abbildungsverzeichnis

1. CM-task Graph mit P-Relationen und C-Relationen 7
2. Softwarearchitektur des CM-task Compilerframeworks 11
3. CM-task Graph mit gültigem CM-task Schedule 24
4. Supertask Graph, Aufteilung des Supertask Graph in Schichten und Schedule des Supertask Graph 27
5. Laufzeit des CM-task Schedulingalgorithmus 32
6. Vergleich der erzeugten Schedules verschiedener Schedulingalgorithmen . . . 33
7. Überblick des Kapitels 4 anhand der Architektur des Compilerframeworks . . 37
8. Erweitertee SP-Graph mit zugehörigem Spezifikationsausschnitt 43
9. Illustration des Kommunikationsmusters I 44
10. Illustration des Kommunikationsmusters II 45
11. Illustration des Kommunikationsmusters III 45
12. Illustration des Kommunikationsmusters IV 46
13. Illustration des Kommunikationsmusters V 47
14. Orthogonale Kommunikation zwischen Basismodulen 48
15. Abhängigkeiten im Spezifikationsprogramm für IRK Methoden 54
16. Illustration des Rahmenprogrammes für Iterierte Runge-Kutta Verfahren . . . 60
17. Beispiel einer Datenumverteilungsoperation 72
18. Beispiel für die Kodierung parametrisierter Datenverteilungsvektoren 73
19. Aufbau der Lastausgleichsbibliothek 75
20. Überblick des Kapitels 5 anhand der Architektur des Compilerframeworks . . 81
21. Syntaxbaum des Spezifikationsprogrammes für IRK Verfahren. 82
22. Annotation der Kommunikationsmengen für IRK Verfahren 84
23. Bestimmung der Lesemengen und Schreibmengen für IRK Verfahren 88
24. Erklärendes Beispiel für den Ablauf der Schedulingphase 91
25. Konstruierte CM-task Graphen des erklärenden Beispiels 93
26. Erzeugte CM-task Schedules für den Rumpf der `while`-Schleife im Beispiel 96
27. Erzeugter CM-task Schedule für das Hauptmodul des Beispiels 96
28. Syntaxbaum des Beispiels vor und nach der Übersetzung der Prozessornummern 98
29. Syntaxbaum des Beispiels nach Schleifenrekonstruktion und nach Annotation der Konstruktdaten 101
30. Syntaxbaum des Beispiels nach Annotation der Abhängigkeiten 105
31. Beispiel für das Einfügen von Datenumverteilungsoperationen 109
32. Semi-dynamisches Ausführungsschema für IRK Verfahren 112
33. Lastausgleich zwischen zeitgleich ausgeführten Modulen 114
34. Anwendungsstruktur für Parallele Adams Verfahren 122
35. Aufteilung des Lösungsgebietes in Zonen 123
36. Illustration der Programmvarianten für IRK Verfahren 126
37. Messergebnisse für IRK Verfahren (CHiC Cluster, SGI Altix und JuRoPA Cluster) 128
38. Messergebnisse für PABM Verfahren (CHiC Cluster und SGI Altix) 129
39. Messergebnisse des LU-MZ Benchmarks (CHiC Cluster und JuRoPA Cluster) 130
40. Messergebnisse des SP-MZ Benchmarks (CHiC Cluster) 132
41. Messergebnisse des BT-MZ Benchmarks (CHiC Cluster und SGI Altix) . . . 133
42. Relative Rechenleistung der Kerne des heterogenen Clusters 135

43. Messergebnisse für semi-dynamische Koordinationsprogramme (heterogener Cluster) 136
44. Illustration der Mappingstrategien 137
45. Ergebnisse des Allgather und des Multiallgather Benchmarks (CHiC Cluster) 139
46. Leistungsvergleich verschiedener Mappingstrategien für IRK, PAB und PABM Verfahren (CHiC Cluster und JuRoPA Cluster) 140

Tabellenverzeichnis

1. Konstruktionsregeln für Kommunikationsmengen 83
2. Konstruktionsregeln für Lese- und Schreibmengen 87
3. Konstruktionsregeln zur Annotation der Datenabhängigkeiten 89
4. Hardware- und Softwarekonfiguration der Zielplattformen 120
5. Eigenschaften der Anwendungsbenchmarks 121
6. Kollektive Kommunikationsoperationen in den IRK, PAB und PABM Benchmarks 138
7. Grammatik der Spezifikationssprache (1) 152
8. Grammatik der Spezifikationssprache (2) 153
9. Grammatik der erweiterten Spezifikationssprache 154
10. Grammatik des Rahmenprogrammes 155
11. Grammatik des erweiterten Rahmenprogrammes 156
12. Grammatik der Plattformspezifikation 157

Algorithmenverzeichnis

1. Lastbalancierungsschritt für einen Supertask. 28
2. Scheduling einer Schicht des Supertask Graph 30
3. Gruppenausgleichsphase 31
4. Konstruktion des CM-task Graph eines Verbundmoduls 92
5. Hierarchisches Scheduling im CM-task Compiler 94
6. Konstruktion der Syntaxbäume des Rahmenprogrammes 97
7. Schleifenrekonstruktion im Rahmenprogramm 100
8. Dynamischer Lastausgleichsalgorithmus 115

Verzeichnis der Listings

1. Beispiel für Konstanten- und Datentypdefinitionen 38
2. Beispiel für Datenverteilungstypdefinitionen 39
3. Beispiel für eine Basismoduldefinition 40
4. Spezifikationsprogramm für Iterierte Runge-Kutta Verfahren 50
5. Beispiel für eine Plattformbeschreibung 53
6. Erweitertes Spezifikationsprogramm für Iterierte Runge-Kutta Verfahren . . . 56
7. Rahmenprogramm für Iterierte Runge-Kutta Verfahren 59
8. Erweitertes Rahmenprogramm für Iterierte Runge-Kutta Verfahren 62
9. Statisches Koordinationsprogramm für Iterierte Runge-Kutta Verfahren (i) . . 66
10. Statisches Koordinationsprogramm für Iterierte Runge-Kutta Verfahren (ii) . 68
11. Beispiel einer Basismodulimplementierung 77
12. Spezifikationsprogramm für parallele Adams Verfahren 167
13. Spezifikationsprogramm des LU-MZ Benchmarks 168
14. Spezifikationsprogramm des SP-MZ Benchmarks 169
15. Spezifikationsprogramm des BT-MZ Benchmarks 170

1. Einleitung

Das parallele Rechnen wird für viele Simulationen aus dem naturwissenschaftlich-technischen Bereich wie z.B. für Wettervorhersagen oder bei der Flugzeugkonstruktion eingesetzt, um einerseits die Laufzeit der Anwendungen zu verkürzen und um andererseits auch Probleme detaillierter betrachten zu können. Die Entwicklung derartiger paralleler Anwendungsprogramme gestaltet sich jedoch oftmals schwieriger als das Schreiben rein sequentieller Programme, da zusätzliche Koordinations- und Synchronisationsoperationen berücksichtigt werden müssen. Die Erreichung einer höchstmöglichen Leistungsfähigkeit auf einer gegebenen parallelen Rechenplattform erfordert darüber hinaus architekturabhängige Anpassungen, die die geeignete Ausnutzung der vorhandenen Rechen- und Kommunikationsleistung sicherstellen. Diese Anpassungen sind häufig nicht auf andere Plattformen mit anderen Leistungsparametern übertragbar, so dass eine Portierung unter Beibehaltung der Effizienz einen hohen Aufwand verursachen kann.

Die Architektur von Parallelrechnern kann sich in vielen Details unterscheiden. Ein wichtiges Kriterium ist die Organisation des Speichers, wobei zwischen Rechnern mit einem gemeinsamen Speicher, der allen Prozessoren zur Verfügung steht, und Rechnern mit verteilten Speichern, die jeweils nur von einem Prozessor genutzt werden können, unterschieden wird. Viele Parallelrechner basieren auf verteilten Speichern, da damit gerade bei hohen Prozessorzahlen ein günstigeres Preis/Leistungs-Verhältnis erzielt werden kann. Auf derartigen Systemen ist ein Datenaustausch zwischen verschiedenen Prozessoren mit Kommunikation verbunden, die entweder explizit durch den Programmierer spezifiziert werden muss oder implizit durch geeignete Softwareumgebungen erkannt werden kann.

Als Standard für die Realisierung dieser Kommunikationsoperationen hat sich die *MPI* (message passing interface) Bibliothek [68] etabliert, für die viele Implementierungen von frei verfügbaren, portablen Versionen bis hin zu hoch optimierten Varianten für bestimmte Plattformen existieren. MPI unterstützt Kommunikation zwischen zwei ausgezeichneten Prozessoren (Punkt-zu-Punkt Kommunikation) und zwischen Teilmengen der vorhandenen Prozessoren (kollektive Kommunikation). Kollektive Kommunikationsoperationen können insbesondere bei einer hohen Prozessoranzahl eine hohe Ausführungszeit besitzen und damit zu einer erheblichen Verlängerung der Gesamtlaufzeit von Anwendungsprogrammen führen. Für die Parallelverarbeitung ergibt sich daraus die Herausforderung, geeignete Programmiermodelle zur Verfügung zu stellen, die zu einer Reduktion der entstehenden Kommunikationszeiten beitragen.

Für die Implementierung paralleler Anwendungsprogramme können verschiedene Programmiermodelle eingesetzt werden. Das *datenparallele Programmiermodell* partitioniert und verteilt die Daten eines Anwendungsprogramms auf die verfügbaren Recheneinheiten, wobei jede Recheneinheit für die Berechnungen auf den zugewiesenen Daten verantwortlich ist (owner-computes Regel). Die Realisierung von datenparallelen Anwendungen erfolgt häufig mit Hilfe eines SPMD-Programmieransatzes (single program, multiple data), d.h. es wird ein paralleles Programm verwendet, das von den beteiligten Recheneinheiten auf unterschiedliche Teile der Programmdaten angewendet wird.

Im rein *taskparallelen Programmiermodell* wird eine Anwendung in eine Menge von Teilaufgaben (Tasks) zerlegt, die auf die verfügbaren Prozessoren verteilt werden. Die Zerlegung und die Zuordnung auf die Recheneinheiten können sowohl statisch durch den Anwendungsentwickler oder durch ein geeignetes Compilerwerkzeug als auch dynamisch zur Laufzeit erfolgen. Die Tasks einer Anwendung können unterschiedliche Berechnungen auf verschiedenen Daten

ausführen, weswegen man auch vom MPMD-Programmierkonzept (multiple program, multiple data) spricht.

Zwischen den Tasks einer Anwendung können Abhängigkeitsbeziehungen bestehen, bspw. kann ein Task Daten produzieren, die für die Abarbeitung eines anderen Tasks benötigt werden. Die Abhängigkeiten können zu Kommunikationsoperationen führen, falls die beteiligten Tasks auf unterschiedlichen Prozessoren ausgeführt werden. Taskparallelität bietet auch Vorteile für irreguläre Berechnungen, die bspw. durch dynamisch erzeugte Tasks implementiert werden können. Zur Auslastung von großen parallelen Rechensystemen müssen entsprechend viele Tasks erzeugt und verwaltet werden, so dass die Anzahl der Rechenoperationen pro Task sinkt und ein hoher Verwaltungsaufwand entstehen kann. Für viele Anwendungen kommt erschwerend hinzu, dass der verfügbare Grad der Taskparallelität beschränkt ist.

Gemischt parallele Programmiermodelle vereinen die Eigenschaften reiner Daten- und Taskparallelität. In diesen Programmiermodellen wird eine Anwendung in eine Menge von *parallelen Tasks* zerlegt, die jeweils durch mehrere Recheneinheiten gemeinsam abgearbeitet werden können. Die parallelen Tasks einer Anwendung werden auch als malleable task, multiprocessor task oder M-task bezeichnet und können direkt als datenparallele Programmteile im SPMD Programmierstil implementiert sein oder aus anderen parallelen Tasks zusammengesetzt werden. Daraus ergibt sich eine mehrstufig parallele Anwendungsstruktur, auf deren unterster Ebene die Ausnutzung feinkörniger Parallelität durch datenparallele Berechnungen steht und auf den oberen Ebenen grobkörnige Parallelität durch zeitgleich ausgeführte parallele Tasks ausgenutzt wird.

Zwischen den parallelen Tasks einer Anwendung können wie im rein taskparallelen Modell Abhängigkeiten bestehen. Da die Daten eines Tasks üblicherweise verteilt auf allen ausführenden Prozessoren vorliegen, können durch vorhandene Abhängigkeiten Datenumverteilungsoperationen zwischen Prozessorgruppen entstehen.

Der Vorteil eines gemischt parallelen Programmieransatzes gegenüber einem rein taskparallelen Modell besteht vor allem in der geringeren Taskanzahl einer Anwendung, wodurch der Verwaltungsaufwand reduziert und das Ausnutzen hoher Prozessorzahlen erleichtert wird. Im Vergleich zu einem datenparallelen Modell besteht der Vorteil in der Beschränkung der Berechnungen und kollektiven Kommunikationsoperationen eines parallelen Tasks auf Teilmengen der Prozessoren, so dass der durch Kommunikation verursachte Laufzeitanteil reduziert werden kann.

Die vorliegende Arbeit befasst sich mit der Unterstützung der Programmentwicklung auf parallelen Rechenplattformen. Zu diesem Zweck wird ein gemischt paralleles Programmiermodell (das Modell der kommunizierenden Multiprozessor-tasks (CM-tasks)) eingeführt und ein Compilerframework zur Unterstützung der Programmentwicklung präsentiert. Das CM-task Modell kann als Erweiterung des TwoL-Modells [90, 95] um eine zusätzliche Kommunikationsebene zwischen zeitgleich ausgeführten parallelen Tasks aufgefasst werden.

Der Kern des CM-task Compilerframeworks ist der CM-task Compiler, der aus einer gegebenen plattformunabhängigen Spezifikation der Struktur einer parallelen Anwendung mit Hilfe mehrerer aufeinanderfolgender Transformationsschritte ein ausführbares Koordinationsprogramm erzeugt. Die Transformationsschritte beinhalten Schedulingentscheidungen, die festlegen, wie die spezifizierte Parallelität auf einer gegebenen Plattform ausgenutzt werden soll. Auf diese Weise kann eine gegebene Anwendungsspezifikation auf verschiedene Zielplattformen abgebildet werden, so dass die Portabilität der Anwendung gewährleistet und der erforderliche Implementierungsaufwand reduziert wird. Zwischen den einzelnen Trans-

formationsschritten bestehen klar definierte Schnittstellen in Form von Spezifikationen mit zusätzlichen Annotationen, die ein Eingreifen der Anwendungsentwicklers in den Übersetzungsvorgang ermöglichen.

Die Unterstützung von heterogenen Systemumgebungen und Anwendungsprogrammen, für die eine exakte Kostenabschätzung schwierig ist, wird durch den semi-dynamischen Ansatz des CM-task Compilers gewährleistet. In diesem Ansatz wird ein Koordinationsprogramm mit einem flexiblen Ausführungsschema erzeugt, das zur Laufzeit durch eine Lastausgleichsbibliothek an dynamisch ermittelte Leistungsdaten der zugrundeliegenden Plattform angepasst werden kann. Das CM-task Compilerframework stellt eine geeignete Lastausgleichsbibliothek bereit, die eine Heuristik geringer Komplexität enthält und damit den Verwaltungsaufwand zur Laufzeit nur unwesentlich erhöht.

Die Anwendbarkeit des CM-task Programmiermodells und des CM-task Compilerframeworks wird für verschiedene Anwendungen aus dem Bereich des wissenschaftlichen Rechnens demonstriert. Die Anwendungen umfassen verschiedene Lösungsverfahren für Systeme gewöhnlicher Differentialgleichungen und Simulationen aus dem Bereich der Strömungsdynamik. Anhand von Messergebnissen auf verschiedenen Rechenplattformen wird aufgezeigt, dass durch eine Implementierung im CM-task Programmiermodell eine Reduzierung der Anwendungslaufzeit insbesondere für hohe Prozessorzahlen erreicht werden kann. Desweiteren wird für direkt gekoppelte heterogene Plattformen nachgewiesen, dass durch eine dynamische Anpassung des Ausführungsschemas deutlich geringere Laufzeiten im Vergleich zu einem statischen Ansatz möglich sind.

Die Arbeit ist wie folgt gegliedert. Kapitel 2 stellt das CM-task Programmiermodell und das zugehörige Compilerframework vor und enthält eine Einordnung in bestehende Modelle und Werkzeuge für die Entwicklung gemischt paralleler Anwendungen. Kapitel 3 definiert das CM-task Schedulingproblem und stellt einen Schedulingalgorithmus für CM-task Programme vor. Die Schnittstellen, Datenstrukturen und Beschreibungssprachen des CM-task Compilerframeworks werden in Kapitel 4 diskutiert. Desweiteren beschreibt dieses Kapitel das zugrundeliegende Kostenmodell. Die zur Realisierung der Komponenten des Compilerframeworks wird in Kapitel 5 betrachtet. Kapitel 6 demonstriert die Spezifikation verschiedener Anwendungsprogramme mit Hilfe des CM-task Compilerframeworks und zeigt anhand von Laufzeitmessungen die Effizienz der erzeugten Implementierungen auf. Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen.

2. Das CM-task Programmiermodell

Programmiermodelle auf Basis paralleler Tasks tragen häufig zu einer Effizienzsteigerung von großen modularen Anwendungsprogrammen bei. In diesen Programmiermodellen wird eine Anwendung in eine Menge eigenständiger paralleler Tasks und eine Koordinationsstruktur zerlegt, die das Zusammenwirken der Tasks beschreibt. Jeder Task enthält parallelen Programmcode und kann auf einer variablen Prozessoranzahl ausgeführt werden. Als Koordinationsstruktur können gerichtete azyklische Graphen [82, 117], Macro Dataflow Graphen (MDG) [85] oder serien-parallele (SP)-Graphen [95] eingesetzt werden. Der Vorteil dieser Programmiermodelle liegt in der Ausnutzung feinkörniger Parallelität innerhalb der Tasks und grobkörniger Parallelität zwischen den Tasks sowie in der Flexibilität der Ausführungsreihenfolge der parallelen Tasks, wodurch die Anpassung an eine konkrete parallele Plattform ermöglicht wird.

Die *Koordinationsstruktur* einer Anwendung beschreibt Abhängigkeiten zwischen den enthaltenen parallelen Tasks, die auftreten, wenn die Ausgabe eines parallelen Tasks als Eingabe eines anderen Tasks verwendet wird. Diese Datenabhängigkeiten stellen die einzige Möglichkeit dar, Daten zwischen verschiedenen Tasks auszutauschen. Die Berechnungen innerhalb eines parallelen Tasks sind vollständig unabhängig von anderen Tasks.

Wird in einer Anwendung ein Datenaustausch zwischen verschiedenen Programmteilen benötigt, müssen die entsprechenden parallelen Tasks zunächst beendet und im Falle einer weiteren Ausführung neu gestartet werden. Daraus entstehen Nachteile speziell für Anwendungen, die aus einer Vielzahl von Zeitschritten bestehen und jeder Zeitschritt einen solchen Datenaustausch erfordert. In diesem Fall ist ein Task auf die Ausführung eines Zeitschrittes beschränkt und in jedem Zeitschritt entsteht zusätzlicher Verwaltungsaufwand für das Starten und das Beenden der Tasks sowie für Kommunikationsoperationen zur Bereitstellung der benötigten Eingabedaten. Zur Vermeidung dieses Zusatzaufwands kann die Unterstützung eines Datenaustauschs zwischen parallelen Tasks beitragen, der nicht die Beendigung der Tasks erfordert.

Dieses Problem greift das Programmiermodell der kommunizierenden Multiprozessortasks (communicating multiprocessor tasks; CM-tasks) [28, 29] auf, das ein erweitertes Taskkonzept in Form von CM-tasks beinhaltet, die zusätzlich während der Ausführung miteinander kommunizieren können. Das CM-task Modell ermöglicht damit eine bessere Modellierung von Anwendungen, die regelmäßig Informationen zwischen verschiedenen Programmteilen austauschen müssen. Dieser Datenaustausch kann direkt zwischen den entsprechenden CM-tasks erfolgen, ohne deren Abarbeitung unterbrechen zu müssen. Damit trägt das Modell zur Reduzierung des Verwaltungsaufwands für die Ausführung der Tasks bei.

Ein weiterer Vorteil des CM-task Modells ist der mögliche Einsatz spezialisierter Kommunikationsmuster zwischen CM-tasks. Ein Beispiel ist die orthogonale Kommunikation, bei der die Anwendungsprozesse derart in einem zweidimensionalen Gitter angeordnet werden, dass die Kommunikation innerhalb von CM-tasks und die Kommunikation zwischen CM-tasks entlang verschiedener Gitterdimensionen verlaufen und somit orthogonal zueinander ist. Orthogonale Kommunikation spielt eine wichtige Rolle bei der effizienten Implementierung von Lösungsverfahren für gewöhnliche Differentialgleichungssysteme.

Beispiele für Anwendungen des CM-task Modells kommen aus verschiedenen Bereichen. Große gekoppelte Anwendungen kombinieren Algorithmen aus verschiedenen Wissenschaftsbereichen, z.B. Umweltsimulationen, die auf Modellen für die Atmosphäre, das Regenwasser und das Grundwasser basieren, oder die Flugzeugentwicklung, die Aerodynamik-, Antriebs- und

Strukturanalysemodelle benötigt [11]. In der Bildverarbeitung [107] und bei der Auswertung von periodisch aktualisierten Sensordaten [107, 74] treten pipelineartige Berechnungen auf Eingabestreams auf. Die Berechnung der einzelnen Pipelinestufen kann durch zeitgleich ausgeführte CM-tasks realisiert werden, wobei Kommunikation zwischen den Tasks zur Weitergabe der Zwischenergebnisse an die nächste Stufe stattfindet. Ein weiteres Anwendungsgebiet sind Löser für Systeme gewöhnlicher Differentialgleichungen, z.B. Extrapolationsverfahren [95], iterierte Runge-Kutta Verfahren [93] oder Parallele Adams Methoden [100]. Diese Verfahren berechnen in jedem Zeitschritt eine feste Anzahl unabhängiger Stufenvektoren, die am Ende des Schrittes zu einem neuen Lösungsvektor kombiniert werden.

Dieses Kapitel gliedert sich wie folgt. Abschnitt 2.1 umfasst die Vorstellung der grundlegenden Eigenschaften des CM-task Programmiermodells und der verwendeten Koordinationsstruktur, die auch als CM-task Graph bezeichnet wird. Abschnitt 2.2 gibt einen Überblick über das CM-task Compilerframework, das den Anwender bei der Erstellung von effizienten Implementierungen von CM-task Programmen unterstützt. Den Abschluss des Kapitels bildet ein Überblick über Modelle und Programmieransätze für gemischt parallele Anwendungen in Abschnitt 2.3.

2.1. CM-task Programme

Ein **CM-task Programm** besteht aus einer Menge kooperierender CM-tasks, die Teilaufgaben des Programmes berechnen, und einer Koordinationsstruktur, die die Zusammenarbeit der CM-tasks beschreibt.

Ein **CM-task** ist ein eigenständiges paralleles Programmmodul, das auf einer parametrisierten Prozessoranzahl ausgeführt werden kann. Intern besteht ein CM-task aus datenparallelen Berechnungen im SPMD Stil, die bspw. durch ein threadbasiertes Programmiermodell wie OpenMP oder Pthreads auf Rechnern mit gemeinsamem Speicher oder durch ein nachrichtenbasiertes Programmiermodell wie MPI realisiert sein können. Im Folgenden wird von einer MPI basierten Implementierung ausgegangen.

Durch einen CM-task können zwei verschiedene Arten von Kommunikationsoperationen ausgeführt werden. *Interne Kommunikation* ermöglicht einen Datenaustausch zwischen den Prozessoren desselben CM-tasks, wohingegen durch *externe Kommunikation* Daten zwischen den Prozessoren verschiedener, zeitgleich ausgeführter CM-tasks ausgetauscht werden können. Externe Kommunikation ist eine Erweiterung gegenüber Modellen auf Basis paralleler Tasks.

Ein CM-task benötigt eine Menge von *Eingabeparametern* und produziert eine Menge von *Ausgabeparametern* , wobei ein Parameter auch für Ein- und Ausgabe genutzt werden kann. Jeder Parameter besitzt einen festen *Datentyp* , der die Größe und die Struktur beschreibt und einen festen *Datenverteilungstyp* , der angibt, wie die Daten auf die ausführenden Prozessoren des entsprechenden CM-tasks aufgeteilt werden. Ein Beispiel für Datentypen sind mehrdimensionale Felder, die Datenverteilung kann bspw. blockweise erfolgen, d.h. jeder Prozessor besitzt exklusiv einen zusammenhängenden Teil der Daten. Zur korrekten Abarbeitung eines CM-tasks müssen die Eingabeparameter in der geforderten Datenverteilung auf den entsprechenden Prozessoren vorliegen.

Die Interaktionen zwischen den CM-tasks eines Programmes werden durch zwei Relationen erfasst:

1) **P-Relation**
   Zwischen zwei CM-tasks *A* und *B* besteht eine Vorrangbeziehung (precedence relation,

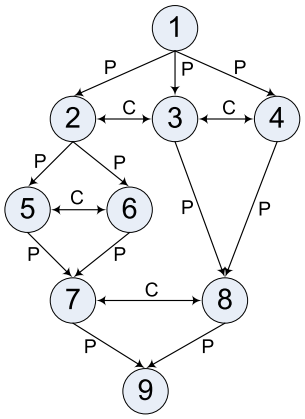


Abbildung 1: Beispiel für einen CM-task Graph mit P-Relationen (Annotation P) und C-Relationen (Annotation C).

P-Relation) , wenn A Ausgabedaten erzeugt, die als Eingabe für B benötigt werden. Die P-Relation ist asymmetrisch und wird durch $A\delta_P B$ gekennzeichnet.

2) **C-Relation**
Eine Kommunikationsbeziehung (communication relation, C-Relation) zwischen zwei CM-tasks A und B spezifiziert einen Datenaustausch zwischen A und B während ihrer Ausführung. Die C-Relation ist symmetrisch und wird durch $A\delta_C B$ gekennzeichnet.

Aus den Relationen zwischen den CM-tasks eines CM-task Programmes ergeben sich für die korrekte Abarbeitung des entsprechenden CM-task Programmes die folgenden Einschränkungen:

- Besteht zwischen zwei CM-tasks A und B eine P-Relation, so müssen A und B zeitlich nacheinander ausgeführt werden. Zusätzlich muss sichergestellt werden, dass die Eingabedaten von B in der erwarteten Datenverteilung auf der korrekten Prozessorgruppe vorliegen. Werden diese Eingabedaten von A in einer anderen Datenverteilung erzeugt oder unterscheiden sich die Prozessorgruppen von A und B, ist eine *Datenumverteilung* zwischen A und B notwendig.

- Zwei CM-tasks A und B, die durch eine C-Relation in Verbindung stehen, müssen zeitgleich ausgeführt werden, um den spezifizierten Datenaustausch zu ermöglichen. Daraus folgt, dass disjunkte Prozessorgruppen für die Ausführung von A und B verwendet werden müssen. Zur Realisierung des Datenaustauschs muss ein gemeinsamer Kommunikationskontext, z.B. in Form eines MPI-Kommunikators, für A und B erzeugt werden.

Die Abhängigkeitsbeziehungen eines CM-task Programmes lassen sich durch einen **CM-task Graph** $G = (V,E)$ darstellen, wobei die Knotenmenge $V = \{A_1,\ldots,A_n\}$ die CM-tasks des Programmes repräsentiert. Die Kantenmenge E des Graphen besteht aus zwei disjunkten Teilmengen E_P und E_C mit $E = E_P \cup E_C$ und $E_P \cap E_C = \emptyset$. E_P enthält gerichtete Kanten entsprechend den P-Relationen des Programmes, d.h. $(A_i,A_j) \in E_P$ falls $A_i\delta_P A_j$ gilt. Die

Menge bidirektionaler Kanten E_C symbolisiert die C-Relationen zwischen den CM-tasks, d.h. $(A_i, A_j) \in E_C$ falls $A_i \delta_C A_j$ gilt. Abbildung 1 zeigt ein Beispiel eines CM-task Graph.

Die Kanten des CM-task Graph symbolisieren die Restriktionen in der möglichen Ausführungsreihenfolge der CM-tasks eines Programmes. Aus den Beobachtungen für die Relationen eines Programmes folgt, dass CM-tasks A und B, die durch einen Pfad im gerichteten Teilgraphen $G_P = (V, E_P)$ verbunden sind, nacheinander ausgeführt werden müssen, wohingegen ein Pfad zwischen A und B im Teilgraphen $G_C = (V, E_C)$ eine gleichzeitige Abarbeitung von A und B erfordert. Existiert jedoch kein Pfad zwischen A und B im CM-task Graph G, so sind A und B unabhängig voneinander und ihre Ausführungsreihenfolge unterliegt keinen Restriktionen, d.h. sowohl eine Abarbeitung nacheinander als auch eine gleichzeitige Ausführung auf disjunkten Prozessorgruppen ist erlaubt.

Ein CM-task Graph enthält einen Konflikt, wenn keine Ausführungsreihenfolge der enthaltenen CM-tasks gefunden werden kann, die alle Restriktionen des Graphen erfüllt. In diesem Fall kann das zugehörige CM-task Programm nicht abgearbeitet werden. Im Folgenden wird eine Teilmenge der möglichen CM-task Graphen definiert, die derartige Konflikte ausschließt.

Definition 1 (Gültiger CM-task Graph) Ein CM-task Graph $G = (V, E)$ heißt gültig, wenn für alle CM-tasks $A, B \in V, A \neq B$ gilt:
Falls es in G einen Pfad von A nach B mit mindestens einer gerichteten Kante gibt, dann existiert kein Pfad von B nach A. □

Aus der Definition folgt, dass in gültigen CM-task Graphen der Teilgraph G_P azyklisch ist und somit eine Ausführungsreihenfolge gefunden werden kann, die keine der Forderungen der gerichteten Kanten verletzt. Eine weitere Folgerung ist, dass falls zwei CM-tasks A und B durch einen Pfad bidirektionaler Kanten verbunden sind, weder ein Pfad von A nach B noch ein Pfad von B nach A mit einer gerichteten Kante existieren kann. Damit wird gewährleistet, dass die gleichzeitige Ausführung von A und B keine Restriktionen der gerichteten Kanten des Graphen verletzt. Gültige CM-task Graphen können jedoch mehrere bidirektionale Pfade zwischen A und B und somit Zyklen innerhalb des Teilgraphen G_C besitzen.

2.2. Werkzeugunterstützung für CM-task Programme

Die Entwicklung von Anwendungsprogrammen im CM-task Programmiermodell gestaltet sich häufig aufwendiger und fehleranfälliger als rein datenparallele oder rein taskparallele Implementierungen. Die Gründe dafür liegen sowohl in der komplexen mehrstufigen Kommunikationsstruktur mit der Unterscheidung von interner und externer Kommunikation als auch in der Koordination der CM-tasks eines Programmes, d.h. der Erzeugung geeigneter Prozessorgruppen, der Zuweisung von CM-tasks an Prozessorgruppen und das Ausführen erforderlicher Datenumverteilungsoperationen. Zusätzlich stellt die Portabilität von CM-task Programmen eine Herausforderung für den Anwendungsentwickler dar, da auf verschiedenen Plattformen eine unterschiedliche Prozessorgruppenaufteilung oder unterschiedliche Ausführungsreihenfolgen für unabhängige CM-tasks zu den geringsten Laufzeiten führen können.

Zur Unterstützung des Anwendungsprogrammierers bei der Realisierung von CM-task Programmen wurde das **CM-task Compilerframework** [29] entwickelt. Das Framework beinhaltet

- eine plattformunabhängige Spezifikationssprache für parallele Algorithmen;

- eine anwendungsunabhängige Beschreibungssprache für parallele Plattformen;
- den CM-task Compiler, der ausgehend von einer gegebenen Algorithmenspezifikation und einer gegebenen Plattformbeschreibung durch mehrere aufeinanderfolgende Transformationsschritte ein auf die Zielplattform angepasstes Koordinationsprogramm erzeugt;
- eine Datenumverteilungsbibliothek zur Realisierung von Datenumverteilungsoperationen und
- eine Lastausgleichsbibliothek zur dynamischen Anpassung der Größen der verwendeten Prozessorgruppen an das Ausführungsverhalten einer Anwendung.

Im Folgenden werden die einzelnen Bestandteile des Compilerframeworks kurz vorgestellt, eine ausführliche Beschreibung der Sprachen und Schnittstellen wird in Kapitel 4 gegeben. Die Implementierung des Frameworks wird in Kapitel 5 beschrieben.

Die **Spezifikationssprache** ermöglicht die Beschreibung der Struktur eines parallelen Algorithmus in Form von Basismodulen und Verbundmodulen. Die **Basismodule** repräsentieren die CM-tasks einer Anwendung und werden als black-box Funktionen mit bekannter Schnittstelle und Ausführungszeit definiert. Die Schnittstellen umfassen die Eingabe- und Ausgabeparameter, die die Auswirkungen auf die übergebenen Datenstrukturen beschreiben, und spezielle Kommunikationsparameter, mit deren Hilfe ein Datenaustausch mit zeitgleich ausgeführten Modulen spezifiziert wird. Die Ausführungszeit wird durch eine *symbolische Laufzeitformel* abhängig von der Anzahl ausführender Prozessoren und plattformspezifischen Parametern definiert. Die konkreten Werte dieser Parameter werden getrennt in der Plattformbeschreibung zur Verfügung gestellt.

Ein **Verbundmodul** drückt die vorhandene Taskparallelität eines Programmteils aus und kann durch einen CM-task Graph repräsentiert werden. Die Spezifikation erfolgt durch einen hierarchischen *Modulausdruck*, auf dessen unterster Hierarchiestufe Aufrufe von Basismodulen und anderen Verbundmodulen stehen. Auf den höheren Hierarchiestufen können durch vorgegebene Konstruktoren Teilberechnungen zusammengefasst werden. Der verwendete Konstruktor definiert dabei, ob die entsprechenden Berechnungen nacheinander ausgeführt werden müssen, zeitgleich ausgeführt werden müssen oder unabhängig voneinander sind und sowohl nacheinander als auch parallel zueinander ausgeführt werden können. Weitere Konstruktoren erlauben die Spezifikation von bedingt ausgeführten Programmteilen und wiederholter Abarbeitung in Form von Schleifen.

Die **Plattformbeschreibungssprache** ermöglicht die Angabe von Eigenschaften der Zielplattform, wie der Rechen- und Kommunikationsleistung und der Anzahl verfügbarer Prozessoren. Die Rechenleistung wird über die mittlere Ausführungsdauer einer arithmetischen Operation definiert, während die Kommunikationsleistung durch eine symbolische Formel beschrieben wird. Eine derartige Formel gibt die Ausführungszeit einer MPI-Kommunikationsoperation in Abhängigkeit von der Datenmenge und der Anzahl beteiligter Prozessoren an. Die Ermittlung der Leistungswerte muss durch den Nutzer erfolgen und kann durch geeignete Benchmarkprogramme unterstützt werden.

Der **CM-task Compiler** ermöglicht die schrittweise Transformation eines gegebenen Spezifikationsprogrammes in ein ausführbares Koordinationsprogramm. Das Koordinationsprogramm ist auf eine konkrete durch eine separate Plattformbeschreibung definierte Zielplattform angepasst. Für die Erzeugung des Ausgabeprogrammes stehen zwei Ausgabemodi mit unterschiedlichen Zielstellungen zur Verfügung:

(a) Der **statische Compileransatz** erzeugt ein Koordinationsprogramm mit einem festen Abarbeitungsplan, d.h. sowohl die Ausführungsreihenfolge der Modulaufrufe als auch die verwendeten Prozessorgruppen werden zur Compilezeit festgelegt und können zur Laufzeit nicht mehr verändert werden. Durch diese Festlegung werden zusätzliche Optimierungen ermöglicht, die häufig zu einem geringeren Koordinationsoverhead zur Laufzeit führen. Der statische Ansatz ist besonders für Anwendungen mit regelmäßigem Berechnungsmuster und homogene Zielplattformen mit exklusiver Nutzung geeignet.

(b) Der **semi-dynamische Compileransatz** erstellt ein Koordinationsprogramm, dessen Abarbeitungsplans durch eine flexible Datenstruktur beschrieben wird. Diese Datenstruktur definiert eine feste Ausführungsreihenfolge der Modulaufrufe, erlaubt aber eine Anpassung der ausführenden Prozessorgruppen zur Laufzeit. Dadurch können basierend auf dynamischen Leistungsdaten auftretende Lastungleichgewichte zwischen Prozessorgruppen ausgeglichen werden. Lastungleichgewichte können bspw. aus ungleichen Prozessorgeschwindigkeiten oder ungenauen Laufzeitvorhersagen für die verwendeten Basismodule entstehen. Daher eignet sich der semi-dynamische Ansatz besonders für heterogene Zielplattformen und für Anwendungen, die Basismodule mit einer sich dynamisch ändernden Ausführungszeit enthalten.

Der schematische Aufbau des CM-task Compilers und der Ablauf des Übersetzungsvorgangs sind in Abbildung 2 dargestellt. Der Transformationsprozess gliedert sich in vier aufeinanderfolgende Schritte: die *Analysephase*, die *Schedulingphase*, die *Datenverteilungsphase* und die *Codegenerierungsphase*. Jeder Schritt generiert neue Informationen und fügt sie der jeweiligen Eingabespezifikation hinzu.

Diese Spezifikationen dienen als Schnittstelle zwischen den einzelnen Transformationsschritten, die als getrennte Werkzeugkomponenten realisiert sind. Der Nutzer kann über diese Schnittstellen in den Transformationsprozess eingreifen und Entscheidungen, die vom Compiler getroffen wurden, analysieren und gegebenenfalls modifizieren. Zusätzlich können an diesen Schnittstellen externe Werkzeuge, etwa zur Visualisierung der Zwischenergebnisse, ansetzen. Der komponentenbasierte Aufbau des Compilers stellt die Erweiterbarkeit und die Adaptierbarkeit sicher, da bestehende Komponenten ausgetauscht und neue Komponenten, etwa zur softwaregestützten Ermittlung von Kostenwerten, hinzugefügt werden können.

Die *Analysephase* erkennt die Daten- und Kommunikationsabhängigkeiten, die im Spezifikationsprogramm implizit über die als aktuelle Parameter von Basis- und Verbundmodulaufrufen verwendeten Variablennamen definiert sind. Eine Datenabhängigkeit besteht zwischen zwei Modulaufrufen A und B, falls A eine Variable schreibt, die nachfolgend als Eingabe von B benötigt wird. Eine Kommunikationsabhängigkeit besteht zwischen Modulaufrufen, die zeitgleich ausgeführt werden müssen und einen gemeinsamen Kommunikationsparameter besitzen. Die Daten- und Kommunikationsabhängigkeiten werden durch Attributierung der abstrakten Syntaxbäume der im gegebenen Spezifikationsprogramm definierten Verbundmodule ermittelt und explizit in Form eines *erweiterten Spezifikationsprogrammes* ausgegeben.

Die *Schedulingphase* entscheidet, wie die vorhandene Taskparallelität des spezifizierten Algorithmus auf einer konkreten parallelen Zielplattform ausgenutzt werden soll. Dazu werden Schedulingtechniken zur Festlegung der Ausführungsreihenfolge unabhängiger Modulaufrufe und Lastbalancierungstechniken zur Bestimmung geeigneter Größen für die verwendeten Prozessorgruppen verwendet. Die Entscheidungen basieren auf einem Kostenmodell, das die

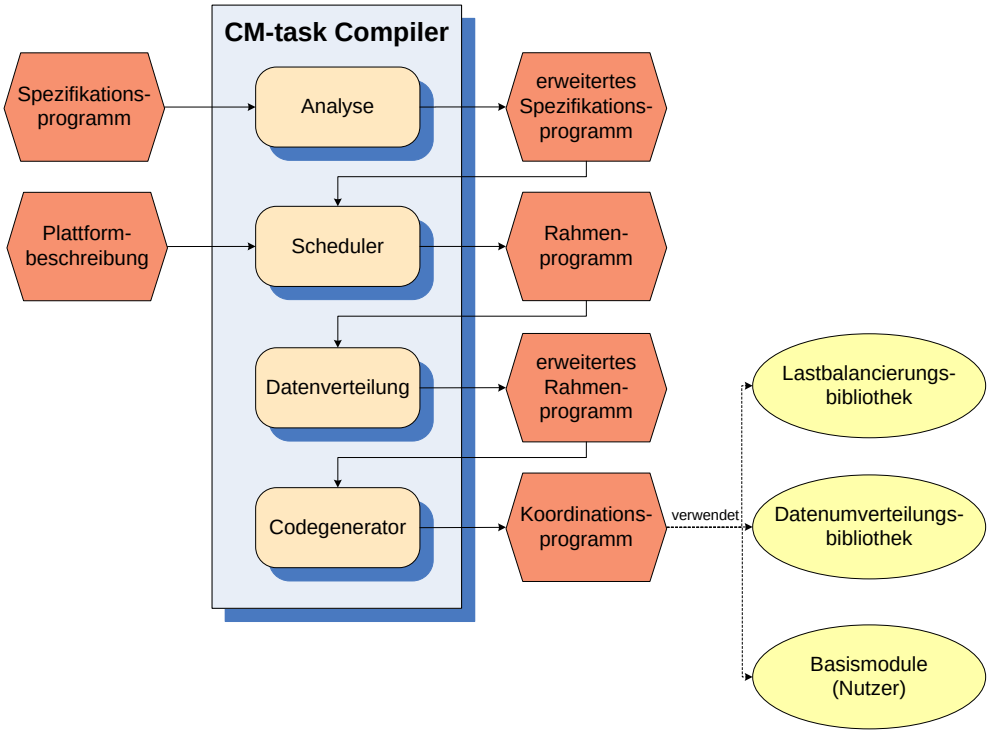


Abbildung 2: Aufbau des CM-task Compilers und Einbindung in den Transformationsprozess des CM-task Compilerframeworks.

Auswirkungen auf die Ausführungszeit der Anwendung vorhersagt. Die Ausgabe der Schedulingphase beinhaltet bereits plattformabhängige Anpassungen und wird daher als *Rahmenprogramm* bezeichnet.

Die *Datenverteilungsphase* legt Datenverteilungen für die innerhalb von Verbundmodulen verwendeten Variablen fest und bestimmt benötigte Datenumverteilungsoperationen, um zwischen Modulaufrufen einen korrekten Datenfluss sicherzustellen. Die Arbeitsweise dieser Phase hängt vom gewählten Compileransatz ab. Im statischen Ansatz werden Heuristiken eingesetzt, um die Anzahl der Umverteilungsoperationen zu reduzieren. Im semi-dynamischen Ansatz kann dagegen aufgrund der dynamischen Anpassbarkeit der Prozessorgruppen erst zur Laufzeit entschieden werden, welche Datenumverteilungsoperationen benötigt werden. Zusätzlich werden im semi-dynamischen Ansatz die Programmpunkte festgelegt, an denen zur Laufzeit ein Lastausgleich erfolgen soll. Das ausgegebene *erweiterte Rahmenprogramm* reflektiert alle Designentscheidungen des CM-task Compilers und kann direkt in ein ausführbares Koordinationsprogramm übersetzt werden.

Die *Codegenerierungsphase* ist für die abschließende Ausgabe des CM-task Compilers zuständig und beinhaltet ein syntaxgerichtetes Übersetzungsschema zur Erzeugung des *Koordinationsprogrammes* in Form eines *C*-Quelltexts mit Aufrufen der MPI Bibliothek. Das ausgegebene Programm verbindet die Ausführung von Basismodulen mit Koordinationscode, der für die Umsetzung des in der Schedulingphase ermittelten Abarbeitungsplans, die Verwaltung der benötigten Prozessorgruppen, die Ausführung von Datenumverteilungsoperationen und (im semi-dynamischen Ansatz) für dynamische Lastausgleichsoperationen verantwortlich ist.

Die Implementierungen der verwendeten Basismodule müssen durch den Anwendungsprogrammierer in Form von parallelen Funktionen, deren Parameterliste mit der spezifizierten Basismodulschnittstelle übereinstimmt, bereitgestellt werden. Zur Laufzeit werden diesen Funktionen zwei Arten von Kommunikatoren übergeben:

- Der *Gruppenkommunikator* definiert die ausführende Prozessorgruppe des Basismoduls und kann für interne Kommunikation genutzt werden.
- Für jeden spezifizierten Kommunikationsparameter des entsprechenden Basismodulaufrufs wird ein *externer Kommunikator* bereitgestellt, der die ausführenden Prozessoren aller Basismodule mit demselben Kommunikationsparameter enthält und für externe Kommunikation genutzt werden kann.

Durch diese Kommunikatoren kann sich ein Basismodul an die vom CM-task Compiler festgelegte Prozessorgruppenaufteilung anpassen.

Die **Datenumverteilungsbibliothek** des CM-task Compilerframeworks unterstützt regelmäßige Datenverteilungen für mehrdimensionale Felddatentypen. Beispiele möglicher Verteilungen sind eine replizierte Speicherung aller Feldelemente auf allen Prozessoren und blockzyklische Verteilungen, bei der die Elemente in eine feste Menge von Blöcken zerlegt und zyklisch über die Prozessoren verteilt werden. Die enthaltenen Datenumverteilungsoperationen erlauben beliebige Quell- und Zielprozessorgruppen und können im statischen Compileransatz vorberechnete Kommunikationsmuster zur Reduzierung des Kommunikationsoverheads verwenden. Weitere Datentypen und Datenverteilungen, bspw. irregeläre und dynamische Strukturen wie dünn besetzte Matrizen oder Graphen, können über eine Schnittstelle des Compilerframeworks integriert werden.

Die **Lastausgleichsbibliothek** unterstützt die Abarbeitung semi-dynamischer Koordinationsprogramme durch Funktionen zum Sammeln dynamischer Lastinformationen und zur Anpassung der verwendeten Prozessorgruppen. Den Kern der Bibliothek bildet eine Algorithmus zur Erkennung von Lastungleichgewichten und zur Wahl angepasster Prozessorgruppengrößen. Der Aufbau der Bibliothek ist modular, so dass die Einbindung nutzerdefinierter Lastausgleichsalgorithmen vereinfacht wird.

2.3. Programmiermodelle für gemischt parallele Anwendungen

Eine Vielzahl von Modellen und Programmieransätzen wurde für die Entwicklung gemischt paralleler Anwendungen für große parallele Rechenplattformen vorgeschlagen. Dazu gehören Spracherweiterungen für bestehende Programmiersprachen, skeletonbasierte Ansätze, bibliotheksbasierte Ansätze und koordinationsbasierte Ansätze. Ein Überblick dieser Ansätze wird in [31] gegeben. Im Folgenden wird eine Auswahl näher vorgestellt.

2.3.1. Spracherweiterungen

Spracherweiterungen basieren auf existierenden Programmiersprachen und definieren zusätzliche Annotationen und Sprachkonstrukte zur Beschreibung einer gemischt parallelen Ausführung. Als Grundlage dienen häufig rein datenparallele Sprachen, die mit taskparallelen Konstrukten erweitert werden, oder rein taskparallele Sprachen, denen Konstrukte für Datenparallelität hinzugefügt werden. Die meisten derartigen Ansätze verwenden einen speziellen source-to-source Compiler zur Übersetzung der zusätzlichen Spracheigenschaften in ein Programm der

zugrundeliegenden Sprache und eine Laufzeitbibliothek zur Realisierung der eingeführten Erweiterungen.

Im Vergleich zum CM-task Programmiermodell erfolgt die Koordination der parallelen Tasks direkt durch den Anwendungsprogrammierer mit Hilfe entsprechender Sprachkonstrukte. Eine explizite Spezifikationssprache und eine explizite Koordinationsstruktur (wie sie der CM-task Graph darstellt) sind nicht vorhanden. Damit ist in diesen Ansätzen kein werkzeuggestütztes Scheduling zur Anpassung an eine konkrete Plattform möglich. Desweiteren bieten die im Folgenden vorgestellten Spracherweiterungen im Gegensatz zum semi-dynamischen Ansatz des CM-task Compilerframeworks keine Softwareunterstützung für Lastausgleichsoperationen zur Laufzeit einer Anwendung.

Fortran M [40] ist eine taskparallele Sprache basierend auf Fortran 77, die Sprachkonstrukte zur Erzeugung von Prozessen und Kommunikationskanälen einführt. Die Kommunikationskanäle ermöglichen Punkt-zu-Punkt Kommunikation zwischen den Anwendungsprozessen unter der Verwendung von Nachrichten. Das Prozessmodell von Fortran M ist dynamisch, d.h. neue Prozesse und Kommunikationskanäle können zur Laufzeit erzeugt werden. Fortran D [41] und High Performance Fortran (HPF) [45] sind datenparallele Sprachen, die auf Fortran 90 basieren und Primitive zur Verteilung von mehrdimensionalen Feldern auf die verfügbaren Prozessoren und datenparallele Operationen wie parallele Schleifen enthalten. Ein gemischt paralleles Programmiermodell, das Fortran M zur Realisierung der Taskparallelität und Fortran D oder HPF zur Unterstützung der datenparallelen Programmteile verwendet, wurde in [10] vorgeschlagen. Dabei wird Fortran M zur Verwaltung der Ressourcen verwendet, z.B. zum Starten datenparalleler Tasks auf Teilmengen der Prozessoren. Die zu verwendenden Prozessorgruppen müssen explizit vom Nutzer angegeben werden. Fortran D oder HPF werden zur Implementierung der Tasks, also zum Verteilen der Daten und Berechnungen auf die zugewiesenen Prozessoren, eingesetzt. Eine Kommunikation zwischen zeitgleich ausgeführten parallelen Tasks ist mit Hilfe von durch Fortran M bereitgestellten Kommunikationskanälen möglich.

Opus [12, 11] definiert eine Reihe von Erweiterungen der datenparallelen Sprache HPF, die die Koordinierung von unabhängigen datenparallelen Modulen unterstützen. Die anvisierten Zielanwendungen sind große gekoppelte Simulationsprogramme, die aus unabhängigen Teilsimulationen bestehen und in regelmäßigen Abständen Daten austauschen, z.B. für die gleichzeitige Optimierung der aerodynamischen Eigenschaften und des zulässigen Gesamtgewichts bei der Flugzeugkonstruktion. Taskparallelität wird in Opus durch gesonderte Routinen ermöglicht, die auf durch den Anwender spezifizierten Teilmengen der Prozessoren ausgeführt werden. Der Kern der Opus Erweiterungen sind die ShareD Abstractions (SDAs), die Daten und Methoden in Form von Objekten kapseln. Eine SDA kann durch mehrere parallel ausgeführte Tasks genutzt werden und erlaubt auf diese Weise einen Datenaustausch zwischen diesen Tasks. Das OpusJava-Framework [62, 61] erlaubt die Integration von Opus Komponenten in große verteilte Java-Anwendungen und bietet damit eine Unterstützung für lose gekoppelte heterogene Systemlandschaften.

Fx [107] ist eine Fortran-basierte Sprache, die ähnliche Direktiven zur Partitionierung und Aufteilung von Daten wie HPF bereitstellt und zusätzliche Direktiven zur Koordinierung einer taskparallelen Ausführung bietet. Taskparallelität kann nur innerhalb bestimmter Programmfragmente (*task regions*) verwendet werden. Innerhalb eines derartigen Fragments können

datenparallele Unterprogramme auf einer Teilmenge der verfügbaren Prozessoren abgearbeitet werden. Die Anzahl der ausführenden Prozessoren und die konkrete Prozessorgruppe können dynamisch zur Laufzeit ermittelt werden. Jedes datenparallele Unterprogramm kann weitere task regions enthalten, so dass eine mehrstufig parallele Ausführung möglich ist. Das Fx-Framework enthält ein Werkzeug, das zur Compilezeit ein optimiertes Ausführungsschema für die Tasks einer task region mit Hilfe dynamischer Programmierung ermitteln kann [106]. Die dafür benötigten Kosteninformationen werden durch Messung verschiedener Ausführungsschemata gewonnen.

High Performance Fortran 2.0(HPF 2.0) [46] ist eine Sprachstandard basierend auf Fortran 95, der auch Erweiterungen für gemischt parallele Anwendungen definiert. Das zugrundeliegende Modell ist ähnlich zum Fx Ansatz, d.h. es können *task regions* definiert werden, in denen grobgranulare datenparallele Tasks oder Tasks mit untergeordneten task regions ausgeführt werden. Mit Hilfe der *on*-Direktive kann der Programmierer die Verteilung der Berechnungen auf die verfügbaren Prozessoren kontrollieren. Die Datenverteilung auf beliebige Teilmengen der Prozessoren wird durch die *distribute*- und *align*-Direktiven unterstützt. Die verwendeten Prozessorgruppen können wie in Fx zur Laufzeit erzeugt und in den entsprechenden Direktiven verwendet werden.

Orca [5] definiert eine Spezifikationssprache, die in *C*-Code mit Aufrufen einer speziellen Laufzeitbibliothek übersetzt wird. Datenparallelität wird in Form von Objekten spezifiziert, die partitioniert und auf eine beliebige Teilmenge der Prozessoren verteilt werden. Die mit einem Objekt verbundenen Berechnungen werden nach der owner-computes Regel ausgeführt und erforderliche Kommunikationsoperationen zum Zugriff auf nichtlokale Daten werden durch den Compiler eingefügt. Taskparallelität wird durch Prozesse unterstützt, die dynamisch erzeugt und gestartet werden können. Die zugrundeliegende Datenverteilung und die zur Ausführung verwendete Prozessorgruppe müssen explizit durch den Anwender spezifiziert werden. Die Kommunikation zwischen Prozessen ist über *shared objects*, die als Instanzen von abstrakten Datentypen implementiert werden, möglich. Jeder Prozess kann Daten innerhalb eines shared objects mit Hilfe atomarer Operationen lesen und modifizieren, so dass auch Daten zwischen zeitgleich ausgeführten Prozessen ausgetauscht werden können.

Braid [120] erweitert Mentat, eine objektorientierte Sprache basierend auf *C++* mit Unterstützung für Taskparallelität, um datenparallele Eigenschaften. Mentat enthält Sprachkonstrukte zur Definition von taskparallelen Objekten und bietet Unterstützung für die dynamische Erzeugung, Synchronisation, Kommunikation und das Scheduling dieser Objekte. Braid fügt datenparallele Elemente hinzu, bspw. *overlay* Methoden zur Initialisierung von Daten, *aggregate* Methoden zur Anwendung einer Operation auf alle oder eine bestimmte Teilmenge von Datenelementen oder *reduction* Methoden, um Informationen aus einer Menge von Datenelementen zu gewinnen. Der Nutzer kann den Compiler über Annotationen über das Kommunikationsverhalten der Objekte informieren. Dies beinhaltet lokale Kommunikation innerhalb von datenparallelen Methoden, z.B. Kommunikation unter benachbarten Prozessen, als auch Interaktionen zwischen verschiedenen Objekten. Zusätzlich kann angegeben werden, welche Kommunikationsoperationen in einer Anwendung dominant sind. Das Laufzeitsystem verwendet diese Annotationen zusammen mit plattformspezifischen Parametern, um eine geeignete Verteilung der Daten zu ermitteln.

2.3.2. Skeletonbasierte Ansätze

Skeletonbasierte Ansätze definieren eine vorgegebene Menge von Koordinationsmustern, die die Kombination von sequentiellen oder parallelen Programmteilen zu komplexen parallelen Anwendungen ermöglichen. Parallele Skeletons können dabei datenparallele (z.B. die Anwendung eines konkreten Codefragmentes auf verschiedene Elemente der Eingabedaten) oder taskparallele Muster (z.B. die Anordnung verschiedener Programmteile als Pipeline) definieren. Mehrstufige Parallelität kann durch eine geeignete Schachtelung der Skeletons spezifiziert werden.

Im Unterschied zum CM-task Programmiermodell können in den im Folgenden vorgestellten Ansätzen keine Interaktionen zwischen zeitgleich ausgeführten parallelen Tasks spezifiziert werden. Desweiteren bietet im Gegensatz zum CM-task Compilerframework keiner dieser Ansätze eine integrierte Unterstützung für das globale Scheduling einer Anwendung, das Erkennen und Realisieren von benötigten Datenumverteilungsoperationen und die dynamische Anpassung der verwendeten Prozessorgruppen zur Laufzeit einer Anwendung.

P3L [78] ist eine Koordinationssprache für Skeletons, die sequentielle Programmteile in der Programmiersprache *C* ausdrückt. Die unterstützten Skeletons beinhalten datenparallele, taskparallele und Kontrollskeletons, die ineinander verschachtelt werden können. Datenparallele Skeletons sind bspw. *map*, das Daten verteilt und ein spezifiziertes Skeleton auf diese Daten anwendet, *reduce*, das verteilte Daten in einen Wert zusammenfasst, *scan*, das den parallelen Präfix eines verteilten Feldes berechnet und *comp*, das Skeletons in Form einer Funktionskomposition kombiniert. Die taskparallelen Skeletons umfassen *pipe*, das eine Reihe von Skeletons nacheinander in Form einer Pipeline auf die Eingabedaten anwendet, und *farm*, das ein konkretes Skeleton auf verschiedene Elemente der Eingabedaten anwendet. Kontrollskeletons sind *seq* zur Spezifikation von sequentiellen Programmteilen und *loop* für die wiederholte Anwendung eines konkreten Skeletons. Das P3L Framework enthält einen Compiler zur Erzeugung von ausführbaren *C+MPI* Programmen, die eine Bibliothek mit optimierten Templates für die verwendeten Skeletons verwenden. Für die einzelnen Skeletons stehen Kostenformeln zur Verfügung, die Kosten für die sequentiellen Programmteile durch Profiling bestimmt werden. Durch Kombination dieser Kostenformeln gemäß des hierarchischen Aufbaus einer Anwendung kann daraus ein Kostenwert für das gesamte Anwendungsprogramm berechnet werden.

taskHPF [14] kombiniert Task- und Datenparallelität über einen zweischichtigen Ansatz. Die taskparallele Koordinationsstruktur wird auf einer hohen Abstraktionsebene durch eine spezielle Sprache beschrieben, die datenparallele Tasks mit Ein- und Ausgabeparametern und Interaktionen zwischen diesen Tasks mit Hilfe vordefinierter Skeletons darstellen kann. Die verfügbaren Skeletons unterstützen die Definition pipelineartiger Berechnungen sowie die Erzeugung und Verwendung von mehreren Instanzen von Pipelinestufen mit schlechter Skalierbarkeit. Die Spezifikationssprache beinhaltet die *on processors*-Direktive, mit der die ausführenden Prozessoren der datenparallelen Tasks angegeben werden. Die Implementierung der datenparallelen Tasks und die Definition der zugrundeliegenden Datenverteilungen erfolgt durch Verwendung von HPF. Die zwischen verschiedenen parallelen Tasks erforderlichen Datenumverteilungsoperationen werden durch den Compiler erkannt und durch die $COLT_{HPF}$ [75, 74] Kommunikationsbibliothek realisiert.

LLC [22, 23] ist eine parallele Programmiersprache basierend auf *C* mit zusätzlichen Direktiven, die eine ähnliche Syntax wie die OpenMP Erweiterungen besitzen. Diese Direktiven ermöglichen die Definition verschiedener Skeletons und die Bereitstellung zusätzlicher Informationen für die Abarbeitung der Anwendung. Der Compiler *llCoMP* übersetzt ein gegebenes Eingabeprogramm in ein paralleles *C+MPI* Programm. Die elementaren datenparallelen Skeletons in LLC beinhalten bspw. das *forall*-Skeleton zur Definition paralleler Schleifen. Taskparallelität wird durch das *sections*-Skeleton, das unabhängige Berechnungen spezifiziert, das *pipeline*-Skeleton zur Definition pipelineartiger Berechnungen und das *taskq*-Skeleton, zur Kombination von Tasks in Form eines Master/Slave Schemas zur Verfügung gestellt. Die Implementierung der verfügbaren Skeletons partitioniert die verfügbaren Prozessoren in Teilgruppen entsprechend der Anzahl der verfügbaren Tasks, also bspw. der Anzahl der Pipelinestufen oder der Anzahl der Schleifeniterationen. Der Nutzer kann den Tasks Gewichte zuweisen, um die Anzahl der zugewiesenen Prozessoren zu beeinflussen.

ASSIST [115] ist ein Framework für eine skeletonbasierte Zusammensetzung von sequentiellen und parallelen Modulen zu komplexen parallelen Anwendungsprogrammen. Sequentielle Module operieren auf Streams von Eingabedaten und können in einer von ASSIST unterstützten Sprache (*C*, *C++* oder *Fortran*) definiert werden. Parallele Module werden durch das *parmod*-Konstrukt spezifiziert, das Eingabe- und Ausgabestreams, eine Menge virtueller Prozessoren und eine virtuelle Prozesstopologie beschreibt. Zusätzlich können Module auf externe Objekte zugreifen, die auch durch mehrere Module gemeinsam verwendet werden dürfen und damit einen Datenaustausch zwischen zeitgleich ausgeführten Modulen ermöglichen. Die Interaktionen zwischen den Modulen einer Anwendung werden mit Hilfe der *ASSIST-CL* Koordinationssprache in Form eines gerichteten, azyklischen Graphen spezifiziert. Die Knoten des Graphen entsprechen den Modulen und die Kanten repräsentieren Datenstreams, die zwischen den Modulen ausgetauscht werden. Für die Ausführung eines Anwendungsprogrammes müssen die virtuellen Prozessoren der parallelen Module auf physische Prozessoren abgebildet werden. Diese Abbildung kann dynamisch zur Laufzeit verändert werden [116].

DIP [20] ist eine patternbasierte Koordinationssprache, die auf Gebietsaufteilung und multiblock Anwendungen, bspw. für die Lösung partieller Differentialgleichungen, fokussiert ist. Die Implementierung von DIP basiert auf der *border-based coordination language(BCL)* [19], d.h. der DIP Compiler übersetzt ein gegebenes Spezifikationsprogramm in ein BCL Programm. BCL unterstützt numerische Lösungsverfahren für Anwendungsprobleme mit mehreren eigenständigen Lösungsgebieten durch automatische Erzeugung von benötigten Kommunikationsoperationen für den Randwertaustausch zwischen den einzelnen Gebieten. Die zugrundeliegenden datenparallelen Tasks, bspw. zur Implementierung eines Lösungsverfahrens für ein konkretes Gebiet, werden in HPF realisiert. DIP stellt das *multiblock*-Skeleton zur Verfügung, das die Definition einer Menge von *k*-dimensionalen Gebieten mit vorgegebenen Koordinaten erlaubt und einen regelmäßigen Randwertaustausch zwischen Gebieten mit benachbarten Koordinaten unterstützt. Zusätzlich wird das *pipe*-Muster zur Beschreibung einer Menge aufeinanderfolgender Pipelinestufen und das *replicate*-Muster zur Erzeugung mehrerer unabhängiger Instanzen eines Programmteils, die jeweils auf unterschiedlichen Elementen eines Eingabestreams arbeiten, bereitgestellt. DIP unterstützt verschiedene Implementierungstemplates für die verfügbaren Muster, wobei der Nutzer das zu verwendende Template und die Anzahl ausführender Prozessoren für jeden Task explizit angeben muss.

SBASCO [21] ist eine Erweiterung des DIP-Ansatzes, die ebenfalls die *multiblock*-, *pipe*- und *farm*-Skeletons unterstützt. Zusätzlich beinhaltet SBASCO ein Kostenmodell für die Laufzeitabschätzung der Skeletons in Abhängigkeit von Parametern der Zielplattform. SBASCO unterscheidet zwei verschiedene Sichten auf eine gegebene Anwendungsspezifikation, die *Anwendungssicht* und die *Konfigurationssicht*. Die Anwendungssicht beschreibt die Struktur der Anwendung mit Hilfe der verfügbaren Skeletons und elementaren datenparallelen Modulen unter Angabe der Ein- und Ausgabeparameter. Die Konfigurationssicht erweitert die Anwendungssicht mit Informationen über Datenverteilungen, Prozesslayout und die interne Struktur von Modulen. Die Anwendungssicht wird durch den Programmierer bereitgestellt, wohingegen die Konfigurationssicht durch ein Werkzeug genutzt wird, um ein geeignetes Ausführungsschema für die Module einer Anwendung auf einer gegebenen parallelen Plattform unter Berücksichtigung der Laufzeitabschätzung zu finden.

2.3.3. Bibliotheksbasierte Ansätze

Bibliotheksbasierte Ansätze unterstützen gemischt parallele Ausführungsschemata durch Bereitstellung von Bibliotheksfunktionen. Diese Funktionen können bspw. die Koordination und Synchronisation von datenparallelen Tasks unterstützen, Datenumverteilungsoperationen bereitstellen, die Erzeugung und Verwaltung von Prozessorgruppen ermöglichen oder die Abarbeitung von datenparallelen Tasks auf Prozessorgruppen erlauben. Analog zu den Spracherweiterungen ist der Programmierer für die Umsetzung einer gemischt parallelen Ausführung durch Einfügen geeigneter Bibliotheksaufrufe in den Programmquelltext verantwortlich.

Die meisten der im Folgenden vorgestellten Ansätze erlauben analog zum CM-task Programmiermodell die Kommunikation zwischen zeitgleich ausgeführten datenparallelen Modulen. Im Gegensatz zum CM-task Programmiermodell steht jedoch weder Werkzeugunterstützung für das Scheduling einer Anwendung noch für das Ausführen von Lastausgleichsoperationen zur Laufzeit einer Anwendung oder für das Erkennen und Einfügen benötigter Datenumverteilungsoperationen zur Verfügung.

HPF/MPI [39] ist eine Bibliothek zur Kopplung von HPF und MPI, die damit die Ausführung von MPI Kommunikationsoperationen innerhalb von HPF Programmen ermöglicht. Dadurch ist auch Kommunikation und Synchronisation zwischen verschiedenen datenparallelen HPF Programmen möglich. Für die durch HPF/MPI bereitgestellten Kommunikationsoperationen können beliebige Variablen des entsprechenden HPF Programmes verwendet werden. Diese Variablen können auch mehrdimensionale Felder sein, deren Elemente verteilt über die ausführenden Prozessoren vorliegen. Daher muss bspw. bei der Kommunikation zwischen zwei Modulen der Fall beachtet werden, dass sich die Datenverteilung der Quell- und Zielvariablen unterscheiden. Die HPF/MPI Bibliothek realisiert diesen Fall durch einen vorherigen Austausch eines Deskriptors, der die zugehörigen Datenverteilungstypen beschreibt.

Die **HPF_TASK_LIBRARY** [8] ermöglicht die Interaktion zwischen zeitgleich ausgeführten, datenparallelen HPF Tasks durch Bereitstellung von Punkt-zu-Punkt und kollektiven Kommunikationsoperationen. Die Bibliothek ist auf das HPF 2.0 Taskmodell ausgerichtet, das die Erzeugung von datenparallelen Tasks auf disjunkten Prozessorgruppen erlaubt, jedoch keine Kommunikation zwischen diesen Tasks unterstützt. Die HPF_TASK_LIBRARY erlaubt auch

den Austausch von verteilt vorliegenden Datenstrukturen und muss daher vor der eigentlichen Kommunikationsoperation Informationen über die verwendeten Datenverteilungstypen austauschen und ein geeignetes Kommunikationsmuster ermitteln. Durch HPF 2.0 können auch geschachtelte parallele Tasks definiert werden, wobei die HPF_TASK_LIBRARY aber nur die Kommunikation zwischen Tasks auf derselben Schachtelungsebene unterstützt.

KeLP-HPF [67] nutzt die *C++* Klassenbibliothek KeLP [35] zur Koordination von datenparallelen Tasks, die mit Hilfe von HPF realisiert werden. KeLP bietet hochsprachige Unterstützung zur Vereinfachung der Entwicklung von blockstrukturierten Algorithmen auf SMP Clustern. Dabei baut KeLP auf MPI auf und bietet Mechanismen zur Verteilung von Daten, zum Verschieben von Daten und zur Verwaltung des parallelen Kontrollflusses. Für die Datenverteilung werden allgemeine blockweise Verteilungen unterstützt und das Verschieben erfolgt durch Berechnung eines Kommunikationsschedules zur Laufzeit. Im KeLP-HPF Programmiermodell wird KeLP zur dynamischen Erzeugung von Prozessorgruppen und zum Starten neuer datenparalleler HPF Tasks verwendet. Daher ist dieses Programmiermodell besonders für Anwendungen geeignet, die regelmäßige datenparallele Operationen auf unregelmäßigen oder dynamisch erzeugten Gebieten ausführen, wie z.B. Methoden mit lose gekoppelten Lösungsgebieten oder Methoden mit einer adaptiven Gebietsverfeinerung. Die Eingabeparameter für die datenparallelen Programmteile werden durch KeLP zusammen mit einem *mapping*-Deskriptor, der dem HPF Code Informationen über die verwendete Datenverteilung übermittelt, bereitgestellt.

Die **ORT** [87] Bibliothek verwendet ein Group-SPMD Programmiermodell, bei dem die Gesamtmenge der Prozessoren in disjunkte Teilmengen zerlegt wird, die jeweils einen parallelen Task ausführen. ORT ermöglicht mehrerere derartige Zerlegungen, die die spezifische Eigenschaft besitzen, dass die enthaltenen Prozessoren orthogonal zueinander in einem virtuellen zwei- oder höherdimensionalen Gitter angeordnet sind. Ein typisches ORT Programm besteht aus Berechnungsphasen und Kommunikationsphasen, die jeweils für genau eine der möglichen Zerlegungen ausgeführt werden, und nur Berechnungen oder Kommunikation innerhalb der entsprechenden Prozessorgruppen verwenden. Ein derartiges Programmiermodell bietet spezielle Vorteile für Anwendungen mit regelmäßigen Kommunikationsmustern, an denen jeweils nur Prozesse entlang einer festen Gitterdimension beteiligt sind. Beispiele für solche Anwendungen sind Algorithmen der linearen Algebra, die auf mehrdimensionalen Feldern operieren, z.B. die LR-Zerlegung, oder parallele Lösungsverfahren für Systeme gewöhnlicher Differentialgleichungen.

Die **TLib** [98] Bibliothek unterstützt die Programmierung mit hierarchischen parallelen Tasks und ist insbesondere für die Implementierung hierarchischer Divide-&-Conquer Algorithmen geeignet. In einem TLib-Programm gibt es parallele Basisfunktionen, die im SPMD Programmierstil implementiert sind, und Koordinationsfunktionen, die für die Ausführung von Basisfunktionen auf einer bestimmten Prozessorgruppe verantwortlich sind. Koordinationsfunktionen können beliebig ineinander geschachtelt werden, so dass ein mehrstufiges Group-SPMD Ausführungsschema entsteht. TLib unterstützt die Implementierung von Koordinationsfunktionen durch Bereitstellung von Bibliotheksfunktionen zur Erzeugung und Verwaltung der benötigten Prozessorgruppen, zur dynamischen Abbildung von auszuführenden Basis- und Kontrollfunktionen auf diese Prozessorgruppen und zur Verwaltung des Kontrollflusses der Anwendung. Die Größen der verwendeten Prozessorgruppen können abhängig

von der verfügbaren Prozessorzahl zur Laufzeit festgelegt werden. Die erforderlichen Datenumverteilungsoperationen zwischen verschiedenen parallelen Tasks werden durch eine separate Datenumverteilungsbibliothek [99] realisiert. Die softwaregestützte Erzeugung von TLib-Programmen aus einer Spezifikation der Anwendungsstruktur wird durch ein funktionales Koordinationssystem [73] unterstützt.

2.3.4. Koordinationsbasierte Ansätze

Koordinationsbasierte Ansätze verwenden eine statische Koordinationsstruktur, die das Zusammenwirken der parallelen Tasks einer Anwendung beschreibt und damit eine globale Sicht auf die Anwendung bietet. Die Koordinationsstruktur kann durch spezielle Sprachen oder zusätzliche Annotationen in einem gegeben Quelltext spezifiziert und mit Hilfe eines Compilers oder eines transformationsbasierten Werkzeugs in ausführbaren Programmcode übersetzt werden. Basierend auf der Koordinationsstruktur sind im Übersetzungsvorgang statische Optimierungen, wie die Berechnung eines geeigneten Schedules, möglich. Das CM-task Modell beinhaltet den CM-task Graphen als Koordinationsstruktur und stellt den CM-task Compiler zur Erzeugung ausführbaren Programmcodes bereit und kann daher ebenfalls als koordinationsbasierter Ansatz aufgefasst werden. Im Gegensatz zum CM-task Modell erlauben die nachfolgend beschriebenen Ansätze jedoch keine Kommunikation zwischen zeitgleich abgearbeiteten Tasks.

Paradigm [56] ist ein Framework, das einen parallelen Compiler für HPF Programme enthält und zusätzliche taskparallele Erweiterungen unterstützt [85]. Die Erweiterungen umfassen Annotationen im Programmquelltext, die die automatisierte Extrahierung der Taskstruktur der Anwendung in Form eines Macro Dataflow Graphen (MDG) ermöglichen. Der MDG ist hierarchisch aufgebaut und besitzt einfache Knoten, die mit der Ausführung einer Berechnung assoziiert sind, Knoten, die Schleifen oder Bedingungen symbolisieren und nutzerdefinierte Knoten. Die Kanten des MDG repräsentieren Daten- oder Kontrollabhängigkeiten zwischen Berechnungen. Zusätzlich enthält der MDG Kosteninformationen in Form von Annotationen an den einfachen Knoten und den Kanten, die die Ausführungszeit der entsprechenden Berechnungen bzw. die Kommunikationszeit für Datenumverteilungsoperationen, die sich aus Datenabhängigkeiten ergeben können, angeben. Die Kosten für die einfachen Knoten werden durch Profiling und Kurvenanpassung auf ein Kostentemplate nach dem Amdahlschen Gesetz ermittelt. Die Kosten für Datenumverteilungsoperationen hängen von der Größe der beteiligten Datenstruktur, dem Overhead für das Senden und das Empfangen einer Nachricht und der Bandbreite des verwendeten Netzwerkes ab. Das Paradigm Framework enthält zwei Schedulingalgorithmen (TSAS und SAS [84]) für die Abbildung eines gegebenen MDG auf eine bestimmte Plattform unter Berücksichtigung der annotierten Kosteninformationen. Die Ausgabe des verwendeten Compilers ist ein optimierter MPMD-Code, der für die benötigten Datenumverteilungsoperationen mehrdimensionaler Felder eine spezielle Bibliothek einsetzt. Das Kommunikationsmuster und der zugehörige Kommunikationsschedule der Datenumverteilungsoperationen werden zur Laufzeit mit dem *FALLS* Algorithmus [86] berechnet.
Im Vergleich zum CM-task Compilerframework beinhaltet der Paradigm-Compiler nur einen statischen Ansatz zur Koordinationscodegenerierung, d.h. Lastausgleichsoperationen zur Laufzeit der erzeugten Anwendung werden nicht unterstützt. Ein weiterer Unterschied besteht bei der Spezifikation eines parallelen Algorithmus. Während der Paradigm-Compiler die verfügbare Taskparallelität eines Algorithmus aus den Annotationen und Variablenzugriffen eines Eingabe-

programmes automatisch erkennt, wird die Taskparallelität im CM-task Compilerframework explizit durch den Anwender durch vorgegebene Konstruktoren in Form eines Spezifikationsprogrammes definiert.

Network of Tasks [79] ist ein Programmiermodell, das eine Koordinationssprache für grobgranulare Tasks mit Schwerpunkt auf der Laufzeitvorhersage enthält. Eine Anwendung wird als gerichteter azyklischer Graph modelliert, dessen Knoten beliebige parallele möglicherweise heterogene Programme darstellen. Für einen bestimmten Knoten können verschiedene parallele Implementierungen existieren und verschiedene Prozessoranzahlen können zur Ausführung verwendet werden. Die gerichteten Kanten des Graphen kennzeichnen gerichtete Kommunikationsoperationen mit verteilten Datenstrukturen oder Streams. Das Scheduling eines gegebenen Taskgraphen erfolgt mit Hilfe eines speziellen Algorithmus [105], der versucht, durch Pipelining und Farming die Anwendungsleistung zu optimieren. Pipelining bedeutet in diesem Zusammenhang die gleichzeitige Ausführung aller Knoten eines Teilgraphen, z.B. einer Schleife, für einen Eingabestream und Farming ermöglicht die replizierte Ausführung von schlecht skalierenden Knoten auf verschiedenen Elementen eines Datenstreams. Die Kostenwerte für das gesamte Anwendungsprogramm werden aus den vom Nutzer bereitgestellten Kosten für die Graphknoten und den basierend auf dem BSP Modell [110] ermittelten Kommunikationskosten zusammengesetzt.
Im Vergleich zum CM-task Compilerframework erfolgt die Spezifikation einer Anwendung direkt als gerichteter Graph, während die CM-task Spezifikationssprache auf vordefinierten Konstruktoren zur Definition möglicher Ausführungsreihenfolgen basiert. Desweiteren beinhaltet das NOT Modell im Gegensatz zum CM-task Compilerframework keine Datenumverteilungsbibliothek und bietet keine Unterstützung für die dynamische Anpassung von Prozessorgruppen.

Das **Kompositionsframework** [58] unterstützt die Kombination von parallelen und sequentiellen Komponenten zu parallelen Anwendungen. Der Schwerpunkt des Frameworks liegt dabei auf der Vorhersage der Laufzeit. Jede verwendete Komponente muss eine Funktionsschnittstelle zur Spezifikation der Eingabe- und Ausgabeparameter und eine Kostenschnittstelle zur Spezifikation des Laufzeitverhaltens in Abhängigkeit von der Anzahl ausführender Prozessoren bereitstellen. Für ein bestimmtes Teilproblem können verschiedene Komponenten zur Verfügung stehen, die eine identische Funktionsschnittstelle bieten, sich aber in der Kostenschnittstelle unterscheiden können. Die Struktur der Anwendung wird durch zusätzliche Sprachannotationen definiert, die durch ein spezielles Compilerwerkzeug ausgewertet werden. Die Implementierung der parallelen Komponenten kann durch ein SPMD-Programmiermodell erfolgen und kann zusätzlich den *compose_parallel*-Operator zur Spezifikation unabhängiger Teilberechnungen, die durch parallel ausgeführte Komponenten bearbeitet werden können, enthalten. Komponenten, die außerhalb dieses Operators verwendet werden, werden sequentiell in der Reihenfolge der Spezifikation abgearbeitet. Die Ausführung einer Anwendung erfolgt unter Verwendung einer *variant dispatch* Tabelle, die die beste Implementierungsvariante für jede Kombination aus Prozessoranzahl und Komponententyp enthält. Zusätzlich enthält die Tabelle einen Schedule für jeden definierten *compose_parallel*-Operator und jede mögliche Prozessoranzahl. Der Schedule wird durch Schedulingtechniken für unabhängige parallele Tasks ermittelt und legt die Ausführungsreihenfolge und die Anzahl ausführender Prozessoren für die entsprechenden Teilberechnungen fest. Zur Laufzeit wird aus der Tabelle der beste Schedule in Abhängigkeit von der Anzahl verfügbarer Prozessoren ausgewählt.

Im Vergleich zum CM-task Programmiermodell besitzt das Kompositionsframework keine separate Spezifikationssprache und bietet keine Unterstützung zur Erkennung und Ausführung benötigter Datenumverteilungsoperationen. Desweiteren ist die *variant dispatch* Tabelle des Kompositionsframeworks statisch, d.h. es ist keine Anpassung der Größe der verwendeten Prozessorgruppen zur Laufzeit möglich. Im Gegensatz dazu ist eine derartige Anpassung in mit CM-task Compiler erzeugten semi-dynamischen Koordinationsprogrammen möglich.

Das **TwoL**-Modell [90, 95] ist ein top-down Ansatz zur Entwicklung strukturierter paralleler Anwendungen, der zwei getrennte Ebenen der Parallelität unterscheidet. Die untere (datenparallele) Ebene definiert die Schnittstellen der Module, die vom Anwendungsentwickler zur Verfügung gestellt werden. Diese Basismodule werden als black-box Codes betrachtet, die auf einer variablen Prozessoranzahl ausgeführt werden können. Für jedes Basismodul können verschiedene Implementierungen, die sich bspw. in der Datenverteilung oder im verwendeten Algorithmus unterscheiden, angegeben werden. Die obere (taskparallele) Ebene nutzt die plattformunabhängige TwoL-Spezifikationssprache zur Definition von Verbundmodulen, die hierarchisch aus anderen Modulen zusammengesetzt sind. Die Struktur eines Verbundmoduls wird mit Hilfe vordefinierter Konstruktoren beschrieben, die zur hierarchischen Kombination von Teilberechnungen verwendet werden. Konstruktoren stehen zur sequentiellen Komposition von Teilberechnungen ($\circ$-Operator), zur parallelen Komposition von Teilberechnungen ($\|$-Operator), für sequentielle Schleifen (for- und *while*-Konstruktoren), für parallele Schleifen ($parfor$-Konstruktor) und für die bedingte Ausführung von Teilberechnungen (if-Konstruktor) zur Verfügung.

Im TwoL-Modell spezifiziert der Programmierer den maximalen Grad verfügbarer Taskparallelität einer Anwendung. Über die Ausnutzung dieser Parallelität auf einer bestimmten Zielplattform entscheiden die Transformationsschritte des TwoL-Frameworks, die eine Spezifikation schrittweise in ein ausführbares Programm übersetzen. Die Transformationsschritte sind durch ein Kostenmodell unterlegt, das die Auswirkung verschiedener Designentscheidungen auf die Ausführungszeit der resultierenden Implementierung abschätzt. Das Kostenmodell für Basismodule basiert auf *parametrisierten Laufzeitformeln*, die aus einem Term zur Beschreibung der Berechnungskosten in Abhängigkeit der ausführenden Prozessoranzahl und einem Term für die Kosten der internen Kommunikationsoperationen bestehen. Die benötigten Laufzeitformeln können bspw. durch ein Werkzeug wie SCAPP [59] aus dem Quelltext des Moduls gewonnen werden. Die Kosten für Datenumverteilungsoperationen zwischen Modulen werden anhand einer plattformspezifischen *startup* Zeit und *byte-transfer* Zeit modelliert. Für die Kosten der Verbundmodule werden die Kosten der enthaltenen Module und Datenumverteilungsoperationen gemäß der hierarchischen Modulstruktur kombiniert.

Die Transformationsschritte des TwoL-Frameworks beinhalten Entscheidungen wie die Festlegung eines geeigneten Schedules und die Bestimmung von Datenverteilungen für die verwendeten Module. Das Framework beinhaltet zwei Schedulingalgorithmen, TwoL-Level [91] und TwoL-Tree [94], die auch in Form eines Scheduling Toolkits (STK) verfügbar sind [26]. Die optimalen Datenverteilungen können mit Hilfe dynamischer Programmierung unter Ausnutzung der Anwendungsstruktur bestimmt werden [101].

Das TwoL-Modell wurde unter verschiedenen Zielstellungen realisiert. Ein Übersetzungssystem für eine TwoL-Spezifikation unter Einbindung eines genetischen Schedulingalgorithmus wurde in [37] entwickelt. In [88, 102] wird eine komponentenbasierte Realisierung der TwoL-Spracharchitektur vorgestellt, in der die einzelnen Transformationsschritte durch separate

Komponenten realisiert sind.

Das CM-task Programmiermodell ist eine Erweiterung des TwoL-Modells, die zusätzlich eine Kommunikation zwischen zeitgleich ausgeführten Tasks unterstützt. Desweiteren besitzt der CM-task Compiler sowohl einen statischen als auch einen semi-dynamischen Ansatz, wohingegen der TwoL-Compiler nur statische Koordinationsprogramme erzeugen kann.

3. Scheduling von CM-task Programmen

Die Ausführung eines CM-task Programmes auf einer parallelen Plattform erfordert einen *Abarbeitungsplan (Schedule)*, der festlegt in welcher Reihenfolge und auf welchen Prozessorgruppen die CM-tasks des entsprechenden Programmes ausgeführt werden. Der Schedule muss dabei die durch P-Relationen und C-Relationen definierten Abhängigkeiten zwischen den CM-tasks des entsprechenden Programmes berücksichtigen. Im Allgemeinen exisitieren für ein CM-task Programm viele verschiedene Schedules, die zu jeweils unterschiedlichen Ausführungszeiten des entsprechenden Programmes führen. In diesem Kapitel wird ein Schedulingalgorithmus vorgestellt, der für ein in Form eines CM-task Graph mit annotierten Kostenfunktionen gegebenes Programm einen Schedule berechnet, der zu einer möglichst geringen Laufzeit dieses Programmes führt.

Das Kapitel ist wie folgt gegliedert. In Abschnitt 3.1 wird das Schedulingproblem für CM-task Programme definiert. Der Schedulingalgorithmus wird in Abschnitt 3.2 vorgestellt. Abschnitt 3.3 enthält Simulationsergebnisse für den Schedulingalgorithmus und Abschnitt 3.4 fasst die Ergebnisse dieses Kapitels zusammen und diskutiert verwandte Arbeiten.

3.1. Das CM-task Schedulingproblem

Für die Definition des CM-task Schedulingproblems wird die Ausführung eines CM-task Programmes auf einer homogenen Zielplattform mit der Menge verfügbarer Prozessoren Q betrachtet. Das CM-task Programm wird durch einen CM-task Graph $G = (V,E)$ beschrieben, dessen Knotenmenge $V = \{A_1,\ldots,A_n\}$ die n CM-tasks des entsprechenden Programmes repräsentiert.

Den Knoten des CM-task Graph sind Kosten zugeordnet, die die Ausführungszeit des zugehörigen CM-tasks abhängig von der Anzahl ausführenden Prozessoren angeben. Die Kosten werden durch die Funktion

$$T : V \times \{1,\ldots,q\} \to \mathbb{R}$$

beschrieben, wobei $q = |Q|$ die Prozessoranzahl der Zielplattform ist. Der Wert $T(A,|R|)$ gibt die Ausführungszeit von CM-task A auf Prozessorgruppe $R \subseteq Q$ an, wobei sich die Ausführungszeit eines CM-tasks A aus der Zeit für interne Berechnungen, der Zeit für interne Kommunikationsoperationen und der Zeit für externe Kommunikationsoperationen mit zeitgleich ausgeführten CM-tasks zusammensetzt.

Die gerichteten Kanten des CM-task Graph sind mit Kommunikationskosten assoziiert, die durch die Funktion

$$T_P : E_P \times \{1,\ldots,q\} \times \{1,\ldots,q\} \to \mathbb{R}$$

beschrieben werden. Dabei gibt $T_P(e,|R_1|,|R_2|)$ mit $e = (A_1,A_2)$ die Kommunikationskosten zwischen CM-task A_1 ausgeführt auf Prozessorgruppe R_1 und CM-task A_2 ausgeführt auf Prozessorgruppe R_2 mit $R_1,R_2 \subseteq Q$ an. Die Funktion T_P repräsentiert die Kosten für Datenumverteilungsoperationen, die aufgrund der P-Relationen des CM-task Programmes entstehen. Eine Datenumverteilungsoperation zwischen zwei durch eine P-Relation verbundene CM-tasks A_i und A_j wird benötigt, falls A_i und A_j auf unterschiedlichen Prozessorgruppen ausgeführt werden oder A_i die Ausgabedaten in einer anderen Datenverteilung erzeugt als sie für die

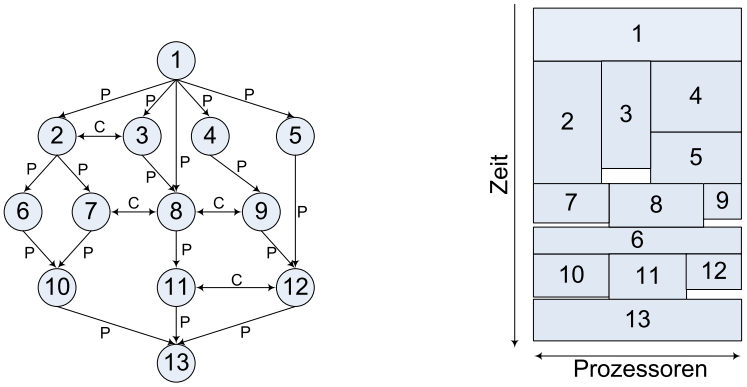


Abbildung 3: Links: Beispiel für einen CM-task Graph mit gerichteten Kanten für P-Relationen (Annotation P) und bidirektionalen Kanten für C-Relationen (Annotation C). Rechts: Illustration eines CM-task Schedule.

Eingabe von A_j benötigt wird, $i, j \in \{1,\ldots,n\}, i \neq j$.

Ein *CM-task Schedule S* eines CM-task Programmes ordnet jedem CM-task $A_i, i = 1,\ldots,n$, ein Zeitintervall zur Ausführung beginnend mit der Startzeit s_i und eine Prozessorgruppe R_i mit $R_i \subseteq Q$ zu, d.h.

$$\begin{aligned} S : \{A_1,\ldots,A_n\} &\rightarrow \mathbb{R} \times 2^Q \\ S(A_i) &= (s_i, R_i). \end{aligned}$$

Abbildung 3 zeigt ein Beispiel für einen CM-task Graph mit einem möglichen CM-task Schedule.

Aus den P- und C-Relationen eines CM-task Programmes ergeben sich die folgenden Einschränkungen für den zugehörigen CM-task Schedule:

(I) Nacheinanderausführung durch P-Relationen verbundener CM-tasks:
Falls es eine P-Relation $A_i \delta_P A_j$ zwischen zwei CM-tasks A_i und A_j, $i, j \in \{1,\ldots,n\}, i \neq j$, gibt, kann die Ausführung von A_j erst beginnen, wenn A_i komplett abgearbeitet wurde und alle zwischen A_i und A_j benötigten Datenumverteilungsoperationen ausgeführt wurden. Daraus ergibt sich für die Startzeiten s_i und s_j von A_i bzw. A_j und die jeweils zugeordneten Prozessorgruppen R_i bzw. R_j die folgende Bedingung:

$$s_i + T(A_i, |R_i|) + T_P(e, |R_i|, |R_j|) \leq s_j$$

mit $e = (A_i, A_j)$.

(II) Gleichzeitige Ausführung durch C-Relationen verbundener CM-tasks:
Falls es eine C-Relation $A_i \delta_C A_j$ zwischen zwei CM-tasks A_i und A_j, $i, j \in \{1,\ldots,n\}, i \neq j$, gibt, müssen A_i und A_j auf disjunkten Prozessorgruppen R_i bzw. R_j mit sich überlappenden Ausführungszeitintervallen abgearbeitet werden, d.h. die folgenden beiden Bedin-

gungen müssen erfüllt sein:

$$R_i \cap R_j = \emptyset \qquad \text{und}$$
$$[s_i, s_i + T(A_i, |R_i|)] \cap [s_j, s_j + T(A_j, |R_j|)] \neq \emptyset.$$

Die überlappenden Ausführungszeitintervalle garantieren, dass A_i und A_j während ihrer Ausführung Daten miteinander austauschen können.

(III) Beliebige Ausführungsreihenfolge für unabhängige CM-tasks:
Falls es keine P-Relation und keine C-Relation zwischen A_i und A_j, $i, j \in \{1, \ldots, n\}, i \neq j$, gibt, dann können A_i und A_j in beliebiger Reihenfolge ausgeführt werden, d.h. A_i und A_j können zeitgleich oder nacheinander abgearbeitet werden. Im Fall einer zeitgleichen Ausführung müssen die jeweils zugeordnet Prozessorgruppen R_i bzw. R_j disjunkt sein, d.h.

$$\text{falls } [s_i, s_i + T_g(A_i, |R_i|)] \cap [s_j, s_j + T_g(A_j, |R_j|)] \neq \emptyset$$
$$\text{dann gilt } R_i \cap R_j = \emptyset,$$

wobei T_g sowohl die Ausführungszeit von A_i als auch die Kommunikationszeit mit nachfolgend ausgeführten CM-tasks A_k,$A_i \delta_P A_k$, umfasst, d.h.

$$T_g(A_i, |R_i|) = T(A_i, |R_i|) + \sum_{\substack{k=1 \\ \text{mit } e=(A_i, A_k) \in E_P}}^{n} T_P(e, |R_i|, |R_k|).$$

Gültiger Schedule Im Folgenden wird ein Schedule, der die Bedingungen (I), (II) und (III) erfüllt, *gültig* genannt. Aus jedem gültigen Schedule S ergibt sich eine Gesamtausführungszeit $T_{max}(S)$ des gegebenen CM-task Programmes, die als derjenige Zeitpunkt definiert ist, an dem alle CM-tasks des Programmes vollständig abgearbeitet sind, d.h.

$$T_{max}(S) = \max_{i=1,\ldots,n} \{s_i + T_g(A_i, |R_i|)\}.$$

Das Finden eines gültigen Schedule S, der die Gesamtausführungszeit $T_{max}(S)$ minimiert, wird als *Schedulingproblem* eines CM-task Programmes bezeichnet. Dieses Optimierungsproblem ist bereits stark NP-schwierig für den Fall, dass der CM-task Graph aus einer linearen Kette von gerichteten Kanten besteht und die Zielplattform 2 Prozessoren besitzt [24].

3.2. CM-task Schedulingalgorithmus

In diesem Abschnitt wird ein Schedulingalgorithmus für CM-task Graphen mit annotierten Kosten vorgestellt, der aus den folgenden vier Phasen besteht:

(a) In der ersten Phase werden durch C-Relationen verbundene CM-tasks zu sogenannten *Supertasks* zusammengefasst. Die CM-tasks innerhalb eines Supertasks müssen aufgrund der durch die C-Relationen spezifizierten Kommunikation zeitgleich ausgeführt werden, vgl. Bedingung (II) in Abschnitt 3.1. Die Abhängigkeiten zwischen den erzeugten Su-

pertasks werden durch einen gerichteten Graphen beschrieben, der auch als *Supertask Graph* bezeichnet wird.

(b) Die zweite Phase berechnet für jeden Supertask und jede Prozessoranzahl p, $p = 1, \ldots, q$, eine Prozessorgruppenaufteilung, die beschreibt, wie die p Prozessoren des Supertasks auf die im jeweiligen Supertask enthaltenen CM-tasks verteilt werden. Mit Hilfe der berechneten Prozessorgruppenaufteilungen werden die Kosten für die Knoten und Kanten des Supertask Graph bestimmt.

(c) Die dritte Phase berechnet eine geeignete Ausführungsreihenfolge und geeignete Prozessorgruppen für die in Phase (a) erzeugten Supertasks, wobei die durch den Supertask Graph beschriebenen Restriktionen der Ausführungsreihenfolge berücksichtigt werden.

(d) Die letzte Phase fügt die Ergebnisse der vorherigen Phasen zusammen und erzeugt den resultierenden CM-task Schedule.

In den folgenden Abschnitten werden die einzelnen Phasen des Schedulingalgorithmus detailliert beschrieben.

3.2.1. Transformation des CM-task Graphen

Die erste Phase des Schedulingalgorithmus identifiziert durch C-Relationen verbundene CM-tasks und fasst diese zu Supertasks entsprechend der folgenden Definition zusammen.

Definition 2 (Supertask) Sei $G = (V, E)$ ein CM-task Graph. Ein Supertask ist ein maximaler Teilgraph $\hat{G} = (\hat{V}, \hat{E})$ mit $\hat{V} \subseteq V$ und $\hat{E} \subseteq E_C$, so dass jedes Paar von CM-tasks $A, B \in \hat{V}, A \neq B$, durch einen Pfad bidirektionaler Kanten aus $\hat{E}$ verbunden ist. □

Jeder CM-task und jede bidirektionale Kante eines CM-task Graph gehört zu genau einem Supertask. Ein einzelner CM-task ohne C-Relationen mit anderen CM-tasks stellt für sich alleine einen Supertask dar. Das Auffinden aller Supertasks eines CM-task Graph ist äquivalent mit dem Finden der Zusammenhangskomponenten eines ungerichteten Graph, wobei die C-Relationen als ungerichtete Kanten betrachtet werden. Die Zusammenhangskomponenten eines gegebenen Graph können durch einen Tiefensuchlauf mit Komplexität $\mathcal{O}(|V| + |E|)$ aufgefunden werden [103].

Definition 3 (Supertask Graph) Sei $G = (V, E)$ ein CM-task Graph mit m Supertasks $\hat{G}_1 = (\hat{V}_1, \hat{E}_1), \ldots, \hat{G}_m = (\hat{V}_m, \hat{E}_m)$. Der zu G gehörige Supertask Graph ist ein gerichteter Graph $G' = (V', E')$ mit der Menge von m Knoten $V' = \{\hat{G}_1, \ldots, \hat{G}_m\}$ und der Menge gerichteter Kanten

$$E' = \{(\hat{G}_i, \hat{G}_j) \mid \exists A \in \hat{V}_i, B \in \hat{V}_j \text{ mit } A \delta_P B\}.$$ □

Abbildung 4 (links) zeigt den zugehörigen Supertask Graph des CM-task Graph aus Abbildung 3 (links). Ein Supertask Graph enthält drei Ebenen der Parallelität, die sich gegenseitig beeinflussen können:

- Der **Supertask Graph** selbst repräsentiert die Vorrangbeziehungen zwischen den Supertasks. Auf dieser Ebene muss der Schedulingalgorithmus die Ausführungsreihenfolge und die Prozessorgruppen der Supertasks bestimmen. Diese Entscheidung wird mit Hilfe der dem Supertask Graph zugeordneten Kosten getroffen, die weiter unten definiert werden.

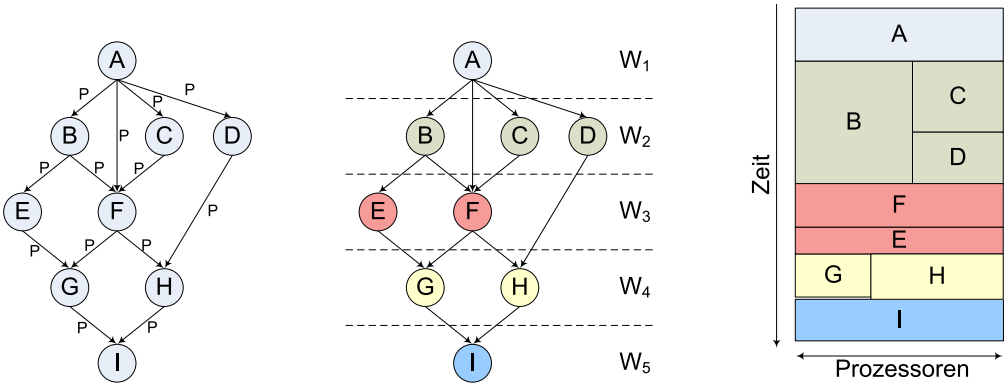


Abbildung 4: Links: Zugehöriger Supertask Graph des CM-task Graph aus Abbildung 3 (links). Bei der Transformation werden bspw. die CM-tasks 2 und 3 zu dem Supertask B zusammengefasst. Mitte: Beispiel für die Partitionierung des Supertask Graph in Schichten. Rechts: Illustration der für die Schichten des Supertask Graph erzeugten Schedules. Beispielsweise werden für die Schicht W_2 $\kappa = 2$ Prozessorgruppen erzeugt. Der resultierende CM-task Schedule nach Erzeugen der Teilgruppen zur Ausführung der in den Supertasks enthaltenen CM-tasks ist in Abbildung 3 (rechts) dargestellt.

- Jeder **Supertask** enthält eine Teilmenge der CM-tasks des CM-task Programmes, die aufgrund von spezifizierten Kommunikationsoperationen zeitgleich ausgeführt werden müssen. Auf dieser Ebene muss der Schedulingalgorithmus die Anzahl der Prozessoren zur Ausführung der in einem Supertask enthaltenen CM-tasks festlegen. Diese Entscheidung hängt von der Anzahl der für den jeweiligen Supertask verfügbaren Prozessoren ab und beeinflusst die Kosten der Knoten des Supertask Graph.

- Jeder CM-task besitzt eine **interne parallele Implementierung**, die die auszuführenden Berechnungen auf die für den jeweiligen CM-task verfügbaren Prozessoren verteilt und die benötigten internen Kommunikationsoperationen bestimmt. Die interne parallele Implementierung eines CM-tasks bestimmt die resultierende Ausführungszeit und beeinflusst damit die Entscheidungen auf den beiden oberen Parallelitätsebenen.

3.2.2. Lastbalancierung für Supertasks

In diesem Unterabschnitt wird ein einzelner Supertask $\hat{G} = (\hat{V}, \hat{E})$ betrachtet und ein Algorithmus zur Berechnung der Prozessoranzahl für jeden in $\hat{G}$ enthaltenen CM-task vorgestellt. Das Ziel dieses Lastbalancierungsalgorithmus ist es, eine gegebene Anzahl an Prozessoren derart auf die in $\hat{G}$ enthaltenen CM-tasks zu verteilen, dass eine minimale Ausführungszeit des gesamten Supertasks $\hat{G}$ resultiert. Die Ausgabe des Algorithmus ist eine *Supertask Allokation* $L_{\hat{G}}(A, p)$ des Supertasks $\hat{G}$ die angibt, durch wieviele Prozessoren CM-task A, $A \in \hat{V}$, ausgeführt wird, wenn p Prozessoren für den gesamten Supertask $\hat{G}$ zur Verfügung stehen.

Im Folgenden wird angenommen, dass m CM-tasks im Supertask $\hat{G}$ enthalten sind und p Prozessoren für die Ausführung von $\hat{G}$ zur Verfügung stehen. Da alle m CM-tasks aus $\hat{G}$ zeitgleich ausgeführt werden müssen, muss die Prozessoranzahl p größer oder gleich m sein,

Algorithmus 1 : Lastbalancierungsschritt für einen Supertask.

```
1 begin
2     sei Ĝ = (V̂, Ê) ein Supertask mit V̂ = {A_1, ..., A_m};
3     setze L_Ĝ(A_i, m) = 1, i = 1, ..., m;
4     for (p = m + 1, ..., q) do
5         setze L_Ĝ(A_i, p) = L_Ĝ(A_i, p − 1), i = 1, ..., m;
6         wähle CM-task A_k ∈ V̂, so dass T(A_k, L_Ĝ(A_k, p − 1)) maximal ist;
7         erhöhe L_Ĝ(A_k, p) um 1;
8 end
```

d.h. $p \geq m$. Für $p = m$ weist der Lastbalancierungsalgorithmus jedem CM-task genau einen Prozessor zu. Für $p > m$ werden die p Prozessoren in m Prozessorgruppen aufgeteilt, wobei jede Gruppe einen CM-task ausführt. Die Prozessorgruppenaufteilung wird derart gewählt, dass die auszuführenden Berechnungen gleichmäßig auf die Prozessorgruppen verteilt sind.

Zu diesem Zweck wird ein iterativer Algorithmus verwendet, dessen Ausgangspunkt eine Prozessorgruppenaufteilung in m Gruppen mit jeweils einem Prozessor ist. In jedem Schritt des Algorithmus wird die Anzahl der verfügbaren Prozessoren um eins erhöht und der zusätzliche Prozessor der Prozessorgruppe mit der aktuell höchsten Ausführungszeit hinzugefügt. Algorithmus 1 zeigt den Ablauf des iterativen Lastbalancierungsalgorithmus.

Eine alternative Methode zur Bestimmung der Supertask Allokation basiert auf den sequentiellen Ausführungszeiten der im Supertask enthaltenen CM-tasks, d.h. die Zuweisung von

$$L_{\hat{G}}(A_i, p) = p \cdot \frac{T(A_i, 1)}{\sum_{A_j \in \hat{V}} T(A_j, 1)}$$

Prozessoren zu einem CM-task $A_i \in \hat{V}$. Algorithmus 1 besitzt verglichen mit diesem Ansatz den Vorteil, dass durch die Verwendung der parallelen Ausführungszeit Skalierbarkeitseffekte berücksichtigt werden.

Die Ausgabe von Algorithmus 1 ist eine Supertask Allokation $L_{\hat{G}}$ für jeden Supertask $\hat{G}$ für $m \leq p \leq q$ Prozessoren sowie die entsprechenden Ausführungszeiten $T(A, L_{\hat{G}}(A, p))$ für jeden CM-task A aus $\hat{G}$. Diese Ausführungszeiten bestimmen die Kosten für den Supertask Graph, die vom Schedulingalgorithmus zur Bestimmung einer geeigneten Prozessoranzahl für die Supertasks des gegebenen CM-task Programmes verwendet werden.

3.2.3. Kosten des Supertask Graphen

Die Kosten für die Knoten und Kanten des in Abschnitt 3.2.1 erzeugten Supertask Graph werden mit Hilfe der im vorherigen Schritt berechneten Supertask Allokationen $L_{\hat{G}}$ wie folgt definiert:

Definition 4 (Kosten von Supertask Graphen) Sei $G = (V, E)$ ein CM-task Graph und $G' = (V', E')$ der zugehörige Supertask Graph. Ein Knoten $\hat{G} = (\hat{V}, \hat{E})$ von G' besitzt die Kosten

$$T'(\hat{G}, p) = \begin{cases} \infty & \text{falls } p < |\hat{V}| \\ \max\limits_{A \in \hat{V}} T(A, L_{\hat{G}}(A, p)) & \text{sonst.} \end{cases}$$

Eine gerichtete Kante $\hat{e}_{ij} = (\hat{G}_i, \hat{G}_j)$ mit $\hat{G}_i = (\hat{V}_i, \hat{E}_i)$ und $\hat{G}_j = (\hat{V}_j, \hat{E}_j)$, $i \neq j$, besitzt die Kosten

$$T'_P(\hat{e}_{ij}, p_i, p_j) = \sum_{e \in RE} T_P(e, L_{\hat{G}_i}(A, p_i), L_{\hat{G}_j}(B, p_j))$$

mit

$$RE = \{e = (A,B) \mid \exists A \in \hat{V}_i, B \in \hat{V}_j \text{ mit } A \delta_P B\}.$$ □

Die Kosten T' werden vom im nächsten Unterabschnitt beschriebenen Schedulingalgorithmus verwendet.

3.2.4. Scheduling des Supertask Graphen

Der Supertask Graph ist ein gerichteter Graph, der den Taskgraphen in Programmiermodellen für parallele Tasks mit Abhängigkeiten (z.B. [85, 82, 118]) ähnelt. Ein wichtiger Unterschied muss jedoch beim Scheduling berücksichtigt werden. Die Knoten des Supertask Graph besitzen jeweils eine Mindestforderung an Prozessoren, die die Wahl der ausführenden Prozessorgruppe einschränkt, wohingegen die Knoten der Taskgraphen für parallele Tasks auf beliebigen Prozessorgruppen ausgeführt werden können.

Die Einschränkung für Supertask Graphen resultiert aus der Zusammensetzung der Supertasks aus einer Menge zeitgleich auszuführender CM-tasks. Jedem Supertask müssen durch den Schedulingalgorithmus genügend Prozessoren zugewiesen werden, so dass für jeden im jeweiligen Supertask enthaltenen CM-task mindestens ein Prozessor zur Verfügung steht, d.h. einem Supertask mit m CM-tasks müssen mindestens m Prozessoren zugewiesen werden. Durch geeignete Berücksichtigung dieser zusätzlichen Bedingung können Schedulingalgorithmen für Supertask Graphen aus Schedulingalgorithmen für parallele Tasks mit Abhängigkeiten abgeleitet werden.

Im Folgenden wird ein schichtenbasierter Schedulingalgorithmus für einen Supertask Graph G' vorgestellt, der auf einem Schedulingalgorithmus für parallele Tasks mit Abhängigkeiten [91] basiert. Der neue Algorithmus besteht aus mehreren Phasen, die im Folgenden beschrieben werden.

In der ersten Phase wird G' in eine Menge von *Schichten* partitioniert, so dass jede Schicht eine Menge unabhängiger Supertasks enthält. Die Schichten werden derart konstruiert, dass eine Nacheinanderausführung der Schichten zu einem gültigen Schedule für den Supertask Graph führt. Im Allgemeinen existieren viele Möglichkeiten für eine Partitionierung eines Graphen in Schichten. Die höchste Flexibilität für die Entscheidungen in den nächsten Phasen des Schedulingalgorithmus wird erreicht, wenn die erzeugten Schichten eine möglichst hohe Anzahl an Supertasks enthalten. Aus diesem Grund wird ein greedy Algorithmus verwendet, der den Supertask Graph mit einer Breitensuche durchläuft und der jeweils aktuellen Schicht soviele Supertasks wie möglich hinzufügt.

In der zweiten Phase des Schedulingalgorithmus werden die zuvor erzeugten Schichten nacheinander behandelt. Zur Ausführung einer Schicht W werden κ, $1 \leq \kappa \leq |W|$, Prozessorgruppen $R_1, \ldots, R_\kappa$ erzeugt, wobei jede Gruppe eine bestimmte Teilmenge der Supertasks von W nacheinander ausführt. Falls ein einer Prozessorgruppe R_i, $i = 1, \ldots, \kappa$, zugeordneter Supertask $\hat{G}$ mehrere CM-tasks enthält, wird die Prozessorgruppe R_i in m Teilgruppen aufgeteilt, wobei m die Anzahl der CM-tasks in $\hat{G}$ ist. Die Größe der erzeugten Teilgruppen wird durch die Supertask Allokation $L_{\hat{G}}$ definiert. Für das Scheduling einer Schicht W muss die Anzahl der

Algorithmus 2 : Scheduling einer Schicht des Supertask Graph.

```
1  begin
2      sei W = {Ĝ_1, ..., Ĝ_r} eine Schicht des Supertask Graph G' = (V', E');
3      sei f = max_{i=1,...,r} |V̂_i| die maximale Anzahl an CM-tasks in einem Supertasks aus W;
4      T_min = ∞;
5      for (κ = 1, ..., min{P − f + 1, r}) do
6          partitioniere die Prozessormenge Q in disjunkte Teilmengen R_1, ..., R_κ, so dass
               |R_1| = max{ ⌈P/κ⌉, f } und R_2, ..., R_κ ungefähr die gleiche Größe besitzen;
7          sortiere {Ĝ_1, ..., Ĝ_r}, so dass |V̂_i| > |V̂_{i+1}| oder |V̂_i| = |V̂_{i+1}| und
               T(Ĝ_i, |R_1|) ≥ T(Ĝ_{i+1}, |R_1|) für i = 1, ..., r − 1 gilt;
8          for (j = 1, ..., r) do
9              ordne Supertask Ĝ_j derjenigen Gruppe R_l zu, so dass |R_l| ≥ |V̂_j| und die
                   kumulierte Ausführungszeit T_act(R_l) minimal ist, 1 ≤ l ≤ κ;
10         group_adjustment();   // Gruppenausgleich, s. Algorithmus 3
11         T_κ = max_{j=1,...,κ} T_act(R_j);
12         if (T_κ < T_min) then T_min = T_κ;
13 end
```

Prozessorgruppen κ, die Größe der κ Prozessorgruppen und die Zuordnung von Supertasks zu Prozessorgruppen festgelegt werden. Für jede dieser drei Entscheidungen existieren im Allgemeinen viele verschiedene Möglichkeiten, die zu unterschiedlichen Ausführungszeiten der Schicht W führen.

Algorithmus 2 zeigt das Scheduling für eine einzelne Schicht W des Supertask Graph, die $|W| = r$ Supertasks enthält. Der Algorithmus probiert alle Werte für die Prozessorgruppenanzahl κ durch und wählt denjenigen Wert für κ, der zu der geringsten Ausführungszeit der Schicht W führt. Bei der Partitionierung der verfügbaren Prozessoren in κ Prozessorgruppen wird berücksichtigt, dass einige Supertasks eine hohe Anzahl an Prozessoren zur Ausführung erfordern. Deshalb erzeugt der Partitinierungsschritt eine Prozessorgruppe R_1, die genügend Prozessoren zur Ausführung jedes Supertasks der Schicht W enthält.

Die Zuordnung von Supertasks zu Prozessorgruppen erfolgt durch einen List-Scheduling Algorithmus, der in jedem Schritt einen Supertask derjenigen Prozessorgruppe R_l zuweist, die die geringste kumulierte Ausführungszeit $T_{act}(R_l)$, $l \in \{1, \ldots, \kappa\}$, besitzt. Die kumulierte Ausführungszeit einer Prozessorgruppe R_l ist als Summe der Ausführungszeiten der R_l zugeordneten CM-tasks auf $|R_l|$ Prozessoren definiert. Die Reihenfolge, in der die Supertasks vom List-Scheduling Algorithmus betrachtet werden, wird durch die Anzahl der enthaltenen CM-tasks und durch die Ausführungszeit der Supertasks bestimmt. Supertasks mit einer hohen Anzahl enthaltener CM-tasks werden zuerst betrachtet, da diese Supertasks die Auswahl der Prozessorgruppe einschränken können.

3.2.5. Gruppenausgleich

Die *Gruppenausgleichsphase* des Schedulingalgorithmus reduziert Lastungleichgewichte, die aufgrund einer ungleichmäßigen Verteilung der Berechnungsarbeit auf die κ in der vorherigen

Algorithmus 3 : Gruppenausgleichsphase.

```
1  procedure group_adjustment()
2  begin
3      done=false;
4      repeat
5          berechne kumulierte Ausführungszeit T_act(R_i), i = 1,...,κ;
6          berechne reduzierte kumulierte Ausführungszeit T_red(R_i) als Summe der Ausfüh-
             rungszeiten der R_i zugeorneten Supertasks auf |R_i| − 1 Prozessoren,i = 1,...,κ;
7          wähle Prozessorgruppe R_k mit T_act(R_k) maximal;
8          wähle Prozessorgruppe R_l mit T_red(R_l) minimal;
9          if (T_red(R_l) < T_act(R_k)) then verschiebe einen Prozessor von R_l zu R_k;
10         else done = true;
11     until done ;
12 end
```

Phase erzeugten Prozessorgruppen $R_1,\ldots,R_\kappa$ entstehen können. Dazu werden in jedem Schritt der Gruppenausgleichsphase zwei Prozessorgruppen R_k und R_l, $k,l \in \{1,\ldots,\kappa\}, k \neq l$, derart bestimmt, dass das Verschieben eines Prozessors von R_l zu R_k zu einer Reduzierung der Gesamtausführungszeit der Schicht W führt. Der Gruppenausgleich wird beendet, falls keine derartigen Paare mehr existieren.

Algorithmus 3 zeigt den Ablauf der Gruppenausgleichsphase. Die Auswahl der Prozessorgruppen R_k und R_l erfolgt mit Hilfe der kumulierten Ausführungszeiten $T_{act}(R_i)$ und der reduzierten kumulierten Ausführungszeiten $T_{red}(R_i)$, $i = 1,\ldots,\kappa$. Die reduzierte kumulierte Ausführungszeit einer Gruppe R_i entspricht der Summe der Ausführungszeiten der R_i zugeordneten Supertasks für den Fall, dass statt $|R_i|$ nur $|R_i|-1$ Prozessoren zur Verfügung stehen.

Für das Verschieben eines Prozessors werden zunächst zwei Kandidaten für die Gruppen R_k und R_l bestimmt. Der beste Kandidat für R_k ist die Gruppe mit der höchsten kumulierten Ausführungszeit, da diese Gruppe die Gesamtausführungszeit der Schicht bestimmt. Der beste Kandidat für R_l ist die Prozessorgruppe mit der geringsten reduzierten kumulierten Ausführungszeit. Ein Prozessor wird nur dann von R_l zu R_k verschoben, wenn durch dieses Verschieben die kumulierte Ausführungszeit von R_l nicht größer als die kumulierte Ausführungszeit von R_k wird, also R_l nicht zu der denjenigen Gruppe wird, die die Gesamtausführungszeit der Schicht bestimmt. Für die Gruppe R_k wird angenommen, dass das Hinzufügen eines zusätzlichen Prozessors zu einer geringeren kumulierten Ausführungszeit dieser Gruppe führt. Ist diese Annahme nicht erfüllt, müssen zusätzliche Tests in den Algorithmus integriert werden, die das zyklische Verschieben von Prozessoren zwischen Prozessorgruppen verhindern.

3.2.6. Erzeugung eines CM-task Schedules

Der letzte Schritt des CM-task Schedulingalgorithmus erzeugt den resultierenden CM-task Schedule, indem die für die einzelnen Schichten des Supertask Graph berechneten Schedules nacheinander eingefügt werden. Für die CM-tasks innerhalb eines Supertasks werden die ausführenden Prozessorgruppen mit Hilfe der entsprechenden Supertask Allokation und der

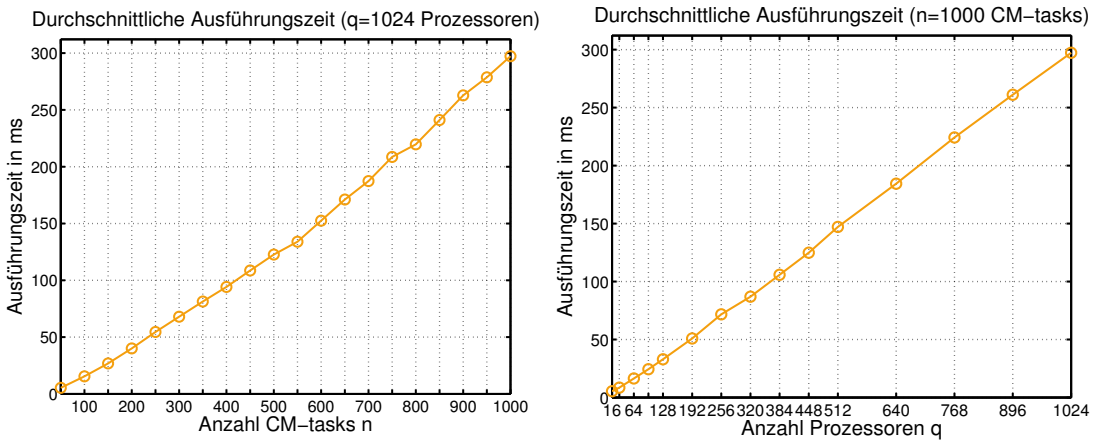


Abbildung 5: Laufzeit des CM-task Schedulingalgorithmus abhängig von der Knotenanzahl n des CM-task Graph (links) und von der Prozessoranzahl q der Zielplattform (rechts).

dem jeweiligen Supertask zugeordneten Prozessorgruppe bestimmt. Für die Festlegung dieser Prozessorgruppen gibt es im Allgemeinen mehrere Möglichkeiten, die zu unterschiedlichen Kosten für die Datenumverteilungsoperationen zwischen den CM-tasks verschiedener Schichten führen können. Um diese Kosten zu reduzieren, werden in diesem Schritt durch P-Relationen verbundenen CM-tasks nach Möglichkeit identische oder sich überlappende Prozessorgruppen zugewiesen.

3.3. Experimentelle Auswertung

In diesem Abschnitt wird die Laufzeit des CM-task Schedulingalgorithmus und die Güte der von diesem Algorithmus erzeugten CM-task Schedules evaluatiert. Dazu werden mit Hilfe eines Graphgenerierungsalgorithmus [66] synthetische CM-task Graphen mit 50 bis 1000 Knoten mit einer Schrittweite von 50 Knoten erzeugt. In den generierten Graphen beträgt die Anzahl der eingehenden Kanten und die Anzahl der ausgehenden Kanten für jeden Knoten maximal vier. Diese Festlegung ist sinnvoll, da CM-tasks meistens nur eine geringe Anzahl an Eingabe- und Ausgabeparameter besitzen.

Die Kosten für die Knoten der synthetischen CM-task Graphen werden mit Hilfe des Amdahlschen Gesetzes [2] festgelegt, d.h. es wird eine Funktion der Form $T_{par}(p) = (\alpha + (1 - \alpha)/p) * T_{seq}$ verwendet, wobei $T_{par}(p)$ die parallele Ausführungszeit auf p Prozessoren, T_{seq} die sequentielle Ausführungszeit und α, $0 \leq \alpha \leq 1$, den inhärent sequentiellen Anteil am Programmcode angibt. Für T_{seq} und α wird für jeden Knoten eines synthetischen Graphen jeweils ein zufälliger Wert verwendet. Um Effekte aufgrund bestimmter Graphstrukturen oder Kostenfunktionen weitgehend auszuschließen werden für jede Knotenanzahl 100 verschiedene Graphen generiert. Die im Folgenden vorgestellten Ergebnisse geben das arithmetische Mittel über die erzeugten Graphen mit der jeweiligen Knotenanzahl an.

Zunächst wird die Laufzeit des CM-task Schedulingalgorithmus auf einem AMD Opteron 8425 HE „Istanbul“ Prozessor mit 2.1 GHz betrachtet. Wie Abbildung 5 (links) zeigt, steigt die Laufzeit des Algorithmus ungefähr linear mit der Knotenanzahl im CM-task Graph an. Dieses

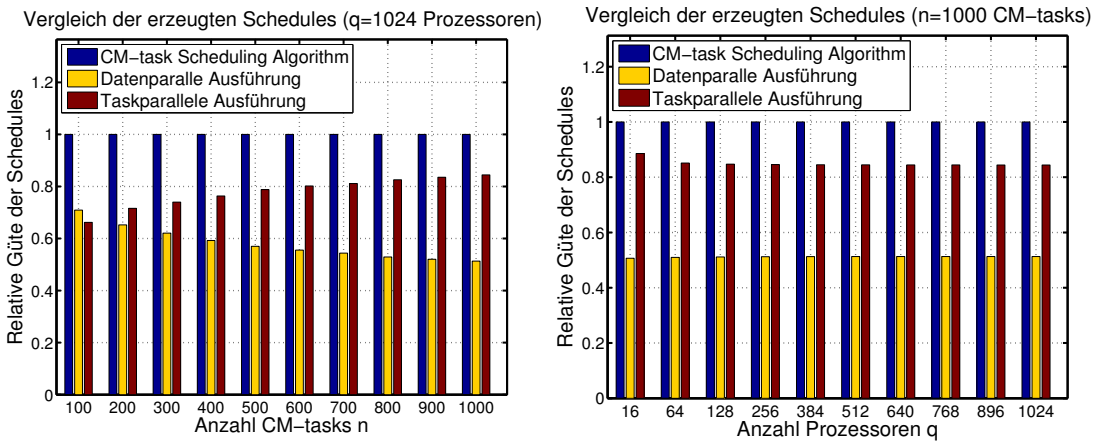


Abbildung 6: Vergleich der relativen Güte der erzeugten Schedules abhängig von der Knotenanzahl n im CM-task Graph (links) und von der Prozessoranzahl q der Zielplattform (rechts).

Verhalten resultiert aus der linearen Komplexität der Algorithmenschritte, die einen gegebenen CM-task Graph in den zugehörigen Supertask Graph transformieren bzw. den Supertask Graph in Schichten unterteilen. Das Scheduling der einzelnen Schichten wird mit einer Knotenanzahl ausgeführt, die meistens deutlich geringer als die Gesamtknotenanzahl des CM-task Graph ist.

Abbildung 5 (rechts) zeigt, dass die Laufzeit des Schedulingalgorithmus ebenfalls linear mit der Prozessoranzahl der Zielplattform zunimmt. Dieses Laufzeitverhalten resultiert hauptsächlich aus der Anzahl der betrachteten Werte für die Prozessorgruppenanzahl κ beim Scheduling einer Schicht des Supertask Graph. Insgesamt zeigen die Simulationsergebnisse, dass der vorgestellte CM-task Schedulingalgorithmus auch für sehr große CM-task Graphen und für große Zielplattform eine Laufzeit von weniger als $0,3$ Sekunden besitzt. Damit eignet sich dieser Algorihtmus besonders gut für die Implementierung in einem Werkzeug wie dem CM-task Compiler.

Die in Abbildung 6 dargestellte relative Güte gibt den Quotient aus der Gesamtausführungszeit eines vom CM-task Schedulingalgorithmus erzeugten Schedule und einem rein datenparallelen bzw. einem rein taskparallelen Schedule an. Der datenparallele Schedule führt alle Supertasks nacheinander auf allen verfügbaren Prozessoren aus. Der taskparallele Schedule wird durch einen List-Scheduling Algorithmus berechnet, wobei jeder CM-task auf genau einem Prozessor ausgeführt wird.

Die Ergebnisse zeigen, dass eine vom CM-task Schedulingalgorithmus berechnete gemischt parallele Ausführung sowohl einer rein datenparallelen als auch einer rein taskparallelen Ausführung überlegen ist. Der taskparallele Schedule erreicht eine höhere relative Güte für CM-task Graphen mit einer hohen Knotenanzahl, da mehr CM-tasks zur Ausnutzung der Prozessoren der Plattform zur Verfügung stehen und damit die ungenutzte Prozessorzeit reduziert wird. Die relative Güte des datenparallelen Schedules nimmt mit der Knotenanzahl des CM-task Graphen ab. In CM-task Graphen mit einer hohen Knotenanzahl ist meistens auch eine hohe Anzahl von CM-tasks unabhängig voneinander und kann zeitgleich ausgeführt werden, woraus sich häufig eine Reduzierung der Gesamtausführungszeit des gemischt parallelen Schedules ergibt.

Die relative Güte der erzeugten Schedules ist beinahe unabhängig von der Prozessoranzahl

der Zielplattform (vgl. Abbildung 6 (rechts)). Dieses Verhalten resultiert aus den Kostenfunktionen für die Knoten der synthetischen Graphen. Durch Verwendung eines zufälligen Wertes für den inhärent sequentiellen Anteil α in den Funktionstemplates nach dem Amdahlschen Gesetz gibt es in den betrachteten CM-task Graphen Knoten mit einer sehr schlechten Skalierbarkeit, d.h. dass das Hinzufügen zusätzlicher Prozessoren zu diesen Knoten die entsprechende Ausführungszeit nur unwesentlich reduziert. Bei einer hohen Prozessoranzahl wird die Gesamtausführungszeit der berechneten Schedules für alle drei betrachteten Schedulingalgorithmen durch die Ausführungszeit dieser schlecht skalierenden Knoten dominiert.

3.4. Zusammenfassung und verwandte Arbeiten

In diesem Kapitel wurde das Schedulingproblem für CM-task Programme definiert und ein Schedulingalgorithmus zur Berechnung eines CM-task Schedules vorgestellt. Der Schedulingalgorithmus transformiert einen gegebenen CM-task Graph zunächst in einen Supertask Graph, dessen Knoten Mengen von zeitgleich auszuführenden CM-tasks repräsentieren. Die Kosten des Supertask Graph werden durch einen Lastbalancierungsalgorithmus bestimmt. Für das Scheduling des Supertask Graph wurde eine modifizierte Version eines Schedulingalgorithmus für parallele Tasks mit Abhängigkeiten genutzt, die zusätzlich die durch die Knoten des Supertask Graph geforderte Mindestanzahl an zuzuordnenden Prozessoren berücksichtigt. Experimente des vorgestellten Schedulingalgorithmus mit synthetischen Taskgraphen zeigen, dass der Algorihtmus auch für eine hohe Knotenanzahl und eine hohe Prozessoranzahl niedrige Ausführungszeiten erzielt und dass die erzeugten Schedules sowohl einer datenparallelen als auch einer taskparallelen Ausführung überlegen sind.

Für das Scheduling von parallelen Tasks mit Abhängigkeiten wurden verschiedene Algorithmen vorgeschlagen. Diese Algorithmen berücksichtigen keine C-Relationen wie sie zwischen CM-tasks bestehen können und sind somit nicht direkt für das Scheduling von CM-task Programmen einsetzbar. Schedulingalgorithmen für parallele Tasks auf homogenen Plattformen lassen sich in zwei Hauptkategorien untergliedern: Allokations- und Schedulingalgorithmen und schichtenbasierte Algorithmen. Im Folgenden wird ein kurzer Überblick dieser beiden Kategorien gegeben und anschließend Schedulingalgorithmen für parallele Tasks auf heterogenen Plattformen betrachtet.

Allokations- und Schedulingalgorithmen bestehen aus zwei Teilschritten. Der zuerst ausgeführte Allokationsschritt weist jedem parallelen Task eine Prozessoranzahl zu. Der nachfolgend ausgeführte Schedulingschritt verwendet einen angepassten List-Scheduling Algorithmus, um die Ausführungsreihenfolge und die konkreten Prozessorgruppen für die parallelen Tasks zu bestimmen. Die beiden Schritte können auch gekoppelt sein, bspw. indem der durch den Schedulingschritt berechnete Schedule zur Optimierung der durch den Allokationsschritt berechneten Prozessoranzahl genutzt wird.

Die Berechnung der Prozessoranzahl im Allokationsschritt kann durch Definition eines konvexen Optimierungsproblems wie in TSAS [84, 85] und SAS [84], durch Definition eines linearen Optimierungsproblems [54] oder durch einen iterativen Algorithmus wie in CPA [83], CPR [82], MCPA [3], iCASLB [117], Loc-MPS [118] und RATS [50] erfolgen. Die iterativen Algorithmen beginnen mit der Zuordnung von einem Prozessor zu jedem parallelen Task und wählen in jedem Iterationsschritt einen parallelen Task, dessen Prozessoranzahl um eins erhöht wird. Durch Anpassung der anfänglichen Prozessorzuordnung an die geforderte Mindestprozessoranzahl der Supertasks können diese Algorithmen für das Scheduling des Supertask Graph

eingesetzt werden.

Weitere Allokations- und Schedulingalgorithmen wurden für Spezialfälle wie Abhängigkeitsgraphen mit einer bestimmten Struktur oder für spezielle Kostenfunktionen der parallelen Tasks vorgeschlagen. Abhängigkeitsgraphen mit begrenzter Breite und SP-Graphen werden in [64] betrachtet und jeweils ein $(3+\sqrt{5})/2$-Approximationsalgorithmus definiert. Ein Schedulingalgorithmus für Abhängigkeitsgraphen in Form von Bäumen wird in [63] vorgestellt. Für parallele Tasks mit Speedupwerten in Form einer konkaven Funktion existiert ein 3.29-Approximationsalgorithmus basierend auf linearer Programmierung [53]. Ein Überblick und Vergleich verschiedener Allokations- und Schedulingalgorithmen wird in [25] gegeben.

Schichtenbasierte Schedulingalgorithmen partitionieren die Menge paralleler Tasks in disjunkte Teilmengen unabhängiger Tasks und Lösen das Schedulingproblem getrennt für jede dieser Teilmengen. Diese Strategie wurde für die Algorithmen TwoL-Level [91] und TwoL-Tree [94] entwickelt und kann genutzt werden, um Schedulingalgorithmen für unabhängige parallele Tasks für das Scheduling von Abhängigkeitsgraphen einzusetzen [27]. Beispielsweise wurden erweiterte Versionen der für unabhängige parallele Tasks definierten Approximationsalgorithmen mit Faktor 2 [65], mit Faktor $(\sqrt{3}+\varepsilon)$ [69] und mit Faktor $(1.5+\varepsilon)$ [70] entwickelt. Ein Vergleich dieser Algorithmen anhand von synthetischen Abhängigkeitsgraphen ist in [27] enthalten.

Weitere Schedulingalgorithmen für homogene Plattformen basieren auf genetischen Algorithmen [36] oder erlauben nur eine eingeschränkte Anzahl von Prozessorgruppen für die Ausführung paralleler Tasks [7].

Schedulingalgorithmen für parallele Tasks mit Abhängigkeiten auf heterogenen Plattformen zielen meist auf große Cluster-of-Clusters Plattformen bestehend aus mehreren homogenen Teilclustern ab und erlauben die Ausführung einzelner paralleler Tasks nur innerhalb von Teilclustern. Beispiele für derartige Algorithmen sind M-HEFT[34, 108], HCPA[71] und MCGAS [33]. Mehrere Verbesserungen für M-HEFT und HCPA werden in [72] vorgeschlagen. Eine Kombination aus Scheduling und Mapping zur Ausführung von parallelen Tasks mit Abhängigkeiten auf Multicore-Plattformen wird in [32] vorgestellt. Für das Scheduling von dynamisch erzeugten Abhängigkeitsgraphen auf Cluster-of-Clusters Plattformen wurden RePA [49] und DMHEFT [51] entwickelt.

4. Sprachen und Schnittstellen des CM-task Compilerframeworks

Dieses Kapitel befasst sich mit den Sprachen und den Schnittstellen des CM-task Compilerframeworks und stellt die zugrundeliegenden Konzepte vor. Abbildung 7 zeigt die Gliederung dieses Kapitels anhand der Softwarearchitektur des Frameworks. Den Abschluss des Kapitels bildet eine Zusammenfassung in Abschnitt 4.10.

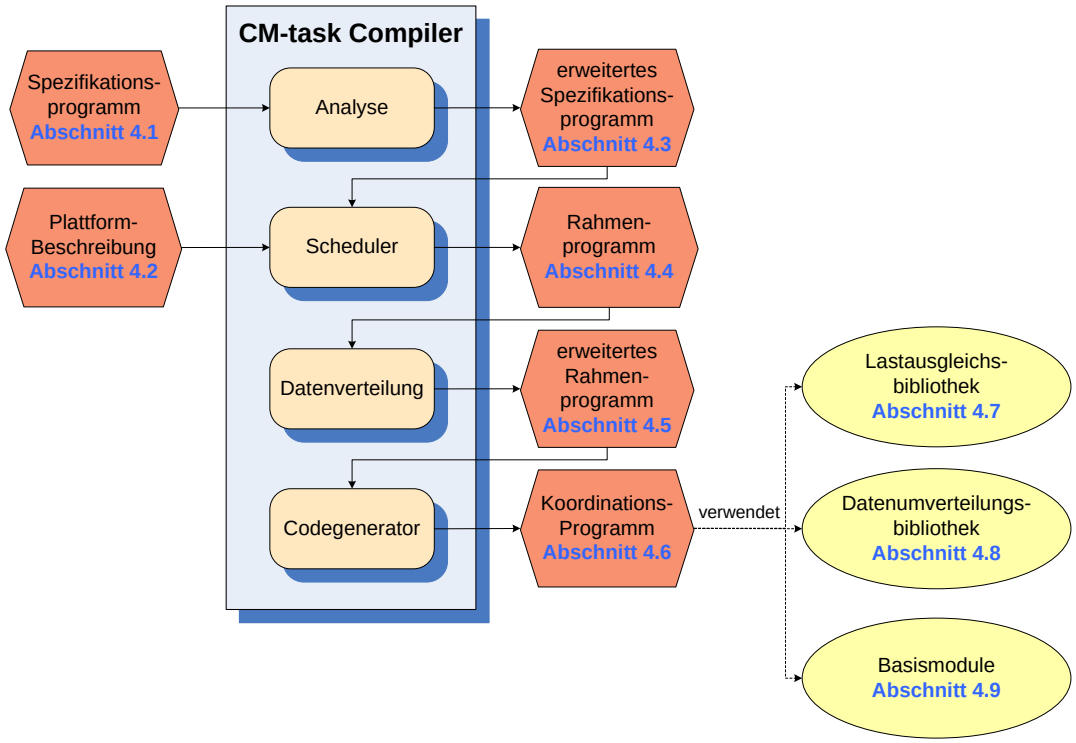


Abbildung 7: Architektur des CM-task Compilerframeworks mit Angabe der Abschnitte, in denen die Bestandteile des Frameworks behandelt werden.

4.1. Spezifikationssprache

Das Spezifikationsprogramm beschreibt die inhärente Parallelität eines Algorithmus und wird vom Anwender in der plattformunabhängigen Spezifikationssprache angegeben. Die Sprache beinhaltet Konstantendefinitionen, Datentypdefinitionen, Datenverteilungstypdefinitionen, Basismoduldefinitionen und Verbundmoduldefinitionen. Im Folgenden werden diese Definitionsarten überblicksartig vorgestellt und anhand von Beispielen verdeutlicht. Die Syntax der Spezifikationssprache wird mit Hilfe einer kontextfreien Grammatik im Anhang Abschnitt B definiert.

4.1.1. Konstanten- und Datentypdefinitionen

Eine **Konstantendefinition** erfolgt durch das Schlüsselwort `const` und besteht aus einem Bezeichner für die Konstante und einem arithmetischen Ausdruck, der den Wert der Konstante

Listing 1: Beispiel für die Definition einer Konstante `n`, eines zweidimensionalen Felddatentyps `matrix` und eines Nutzerdatentyps `graph` mit Identifikationsparameter `1`.

```
const n = 1000;
type matrix = array [n][n] of double;
type graph = usertype(1);
```

angibt. Dieser Bezeichner kann in nachfolgenden Definitionen verwendet werden und ist äquivalent mit der direkten Angabe des zugewiesenen Wertes.

Datentypen werden in der Spezifikationssprache zur Beschreibung der Schnittstellen von Basis- und Verbundmodulen verwendet. Die Spezifikationssprache unterstützt drei Arten von Datentypen: vordefinierte Basisdatentypen sowie vom Anwender definierte Felddatentypen und Nutzerdatentypen.

Basisdatentypen werden für einzelne Zeichen (Typname: `char`), ganze Zahlen (Typname: `int`) und für Gleitkommazahlen mit einfacher Genauigkeit (Typname: `float`) und doppelter Genauigkeit (Typname: `double`) bereitgestellt.

Felddatentypen repräsentieren mehrdimensionale Felder von Basisdatentypen, wie sie in vielen regelmäßigen wissenschaftlichen Anwendungen verwendet werden. Die Definition in der Spezifikationssprache erfolgt mit Hilfe des `array`-Konstruktors unter Angabe der Felddimensionen. Die Größe eines definierten Feldes ist statisch, d.h. Anzahl und Ausdehnung der Felddimensionen stehen zum Spezifikationszeitpunkt fest.

Nutzerdatentypen erlauben die Verwendung von Typen, die nicht durch die zuvor genannten Kategorien abgedeckt werden, z.B. unregelmäßige oder dynamische Typen. Die Spezifikation erfolgt durch das Schlüsselwort `usertype` und der Angabe eines eindeutigen Typidentifikators. Der Aufbau eines Nutzerdatentypen ist für das Compilerframework nicht sichtbar. Daher müssen vom Anwender zusätzliche Funktionen in Form einer Bibliothek zur Verfügung gestellt werden, um eine korrekte Behandlung im erzeugten Koordinationsprogramm sicherzustellen.

Listing 1 zeigt Beispieldefinitionen für eine Konstante `n`, einen Felddatentyp `matrix` und einen Nutzerdatentyp `graph`.

4.1.2. Datenverteilungstypdefinitionen

Datenverteilungstypen geben an, wie die Elemente einer Datenstruktur auf eine Gruppe von Prozessoren verteilt werden. Jeder Datenverteilungstyp besitzt einen zugrundeliegenden Datentyp, der die für die Verteilung verfügbaren Datenelemente beschreibt. Abhängig von der Art dieses Datentyps können Datenverteilungstypen als Felddatenverteilung (für Felddatentypen) oder als Nutzerdatenverteilung (für Nutzerdatentypen) definiert werden. Für Felddatenverteilungen werden Datenumverteilungsoperationen durch das Compilerframework bereitgestellt, während für Nutzerdatenverteilungen der Anwender die entsprechende Funktionalität in Form einer Bibliothek zur Verfügung stellen muss.

Felddatenverteilungen stellen eine Abbildung der Feldelemente auf eine Menge von p Prozessoren dar. Da die Spezifikationssprache unabhängig von einer spezifischen Plattform ist, ist der konkrete Wert von p zum Zeitpunkt der Spezifikation nicht bekannt. Daher werden zur Definition von Felddatenverteilungen *parametrisierte Datenverteilungsvektoren* [96] verwendet,

Listing 2: Beispiel für die Definition einer zeilenblockweisen (`rowblock`) und einer spaltenzyklischen (`colcyclic`) Felddatenverteilung für den zuvor definierten Felddatentyp `matrix` und Definition einer Nutzerdatenverteilung (`partitioned`) mit Identifikationsparameter `2` für den zuvor definierten Nutzerdatentyp `graph`.

```
distrib matrix:rowblock = [block on p][block on 1];
distrib matrix:colcyclic = [block on 1][cyclic on p];
distrib graph:partitioned = userdistrib(2);
```

die die Beschreibung vieler regelmäßiger Datenverteilungen, die häufig in wissenschaftlichen Anwendungen genutzt werden, unterstützen.

Ein parametrisierter Datenverteilungsvektor eines d-dimensionalen Feldes der Größe $n_1 \times \ldots \times n_d$ besitzt die Form $((m_1,b_1),\ldots,(m_d,b_d))$. Die Einträge m_i des Datenverteilungsvektors ordnen p Prozessoren in einem virtuellen d-dimensionalen Gitter der Größe $m_1 \times \ldots \times m_d$ an, wobei $\prod_{i=1,\ldots,d} m_i = p$ gelten muss. Die Blockgrößen b_i geben an, wie viele zusammenhängende Feldelemente einer Dimension $i, i = 1,\ldots,d$, zu einem Block zusammengefasst werden. Die entstehenden Blöcke werden reihum auf die m_i Prozessoren in Dimension i verteilt. Daraus ergibt sich eine Speicherung des Feldelementes mit Indexvektor $(e_1,\ldots,e_d)$, $0 \leq e_i < n_i$ für $1 \leq i \leq d$ auf dem Prozessor mit Gitterkoordinaten

$$\left(\left\lfloor \frac{e_1}{b_1} \right\rfloor \bmod m_1, \ldots, \left\lfloor \frac{e_d}{b_d} \right\rfloor \bmod m_d \right).$$

Im Spezifikationsprogramm wird ein Datenverteilungsvektor durch Angabe eines Musters $[bs_i$ `on` $ms_i]$ für jede Dimension i des zugrundeliegenden Felddatentyps definiert. Dabei ist ms_i ein arithmetischer Ausdruck abhängig von der konkret verfügbaren Prozessoranzahl p, der die Gittergröße m_i beschreibt. Für die Definition der Blockgröße bs_i stehen die folgenden Möglichkeiten zur Verfügung:

- `cyclic` führt zu einer zyklischen Datenverteilung mit Blockgröße $b_i = 1$;
- `block` definiert eine blockweise Datenverteilung mit Blockgröße $b_i = \lceil \frac{n_i}{m_i} \rceil$;
- `blockcyclic`(b_i) erlaubt die direkte Angabe der Blockgröße b_i und
- `replic` definiert eine replizierte Abspeicherung, d.h. alle Prozessoren der Dimension erhalten alle Feldelemente.

Nutzerdatenverteilungen werden durch das Schlüsselwort `userdistrib` und der Angabe eines eindeutigen Identifikators spezifiziert. Listing 2 zeigt Beispieldefinitionen für zwei Felddatenverteilungen und eine Nutzerdatenverteilung.

4.1.3. Basismoduldefinitionen

Die Definition eines Basismoduls erfolgt durch das Schlüsselwort `cmtask` und der Angabe eines Modulnamens, einer Schnittstellenbeschreibung und eines Kostenausdrucks.

Listing 3: Beispiel für die Definition eines Basismoduls `BM` mit einem Eingabeparameter `a` vom Datentyp `matrix` und Datenverteilungstyp `rowblock`, einem Ausgabeparameter `b` mit Datentyp `graph` und Datenverteilungstyp `partitioned`, einem Kommunikationsparameter `c` mit Datentyp `matrix` und Kostenausdruck `1000/p`.

```
cmtask BM (a:matrix:in:rowblock, b:graph:out:partitioned,
  c:matrix:comm) runtime 1000/p;
```

Die Schnittstellenbeschreibung wird als Liste formaler Parameter angegeben, wobei jeder Parameter einen Datentyp und einen *Zugriffstyp* besitzt. Mögliche Zugriffstypen sind `in` für Eingabeparameter, `out` für Ausgabeparameter, `inout` für Ein- und Ausgabeparameter und `comm` für *Kommunikationsparameter*, die zum Datenaustausch mit zeitgleich ausgeführten Modulen genutzt werden. Parameter, die für die Moduleingabe oder Modulausgabe genutzt werden, können zusätzlich einen Datenverteilungstyp zur Spezifikation der erwarteten Eingabeverteilung bzw. der produzierten Ausgabeverteilung besitzen. Parameter mit einem Basisdatentyp werden als Wertparameter übergeben und können daher nur für die Moduleingabe genutzt werden.

Der Kostenausdruck gibt eine Abschätzung der Ausführungszeit des Moduls abhängig von der Anzahl der ausführenden Prozessoren p und plattformabhängiger Parameter an. Eine ausführliche Beschreibung wird im Rahmen des Kostenmodells in Abschnitt 4.2 gegeben. Die Definition des Kostenausdrucks erfolgt nach der Schnittstellenbeschreibung und wird durch das Schlüsselwort `runtime` eingeleitet. Listing 3 zeigt eine Beispielspezifikation eines Basismoduls `BM`.

4.1.4. Verbundmoduldefinitionen

Ein Spezifikationsprogramm beinhaltet ein Hauptmodul, das als Einsprungspunkt für das Koordinationsprogramm dient und durch das Schlüsselwort `cmmain` definiert wird, und eine beliebige Anzahl zusätzlicher Verbundmodule, die durch das Schlüsselwort `cmgraph` definiert werden. Dieses Schlüsselwort verdeutlicht, dass Verbundmodule analog zu den CM-task Graphen die Abhängigkeiten zwischen CM-tasks beschreiben. Alle Verbundmodule besitzen eine analog zu den Basismodulen definierte Schnittstellenbeschreibung und einen Modulrumpf.

Der Modulrumpf enthält Deklarationen lokaler Variablen und einen hierarchisch strukturierten *Modulausdruck*, der aus Aufrufen anderer Basis- und Verbundmodule und Konstruktoren zur Beschreibung der möglichen Ausführungsreihenfolgen besteht. Die folgende vereinfachte Grammatik gibt einen Überblick über den Aufbau eines Modulausdrucks M.

M	$\rightarrow$	**seq** $\{ M_1 M_2 \ldots M_n \}$	Ausführung von $M_1, \ldots, M_n$ nacheinander
	$\mid$	**par** $\{ M_1 M_2 \ldots M_n \}$	Unabhängigkeit von $M_1, \ldots, M_n$
	$\mid$	**for** $(i = 1 : n) \{ M_1 \}$	n-malige Ausführung von M_1 nacheinander
	$\mid$	**while** $(cond)\#It \{ M_1 \}$	wiederholte Ausführung von M_1 nacheinander, It gibt die Größenordnung der Iterationsanzahl an
	$\mid$	**parfor** $(i = 1 : n) \{ M_1 \}$	Unabhängigkeit für n Instanzen von M_1
	$\mid$	**if** $(cond) \{ M_1 \}$	bedingte Ausführung von M_1
	$\mid$	**if** $(cond) \{ M_1 \}$ **else** $\{ M_2 \}$	Ausführung entweder von M_1 oder von M_2
	$\mid$	C	

C	$\rightarrow$	BM $(a_1,\ldots,a_n)$;	Ausführung des Basismoduls BM
	$\mid$	VM $(a_1,\ldots,a_n)$;	Ausführung des Verbundmoduls VM
	$\mid$	**cpar** $\{\ C_1C_2\ldots C_n\ \}$	zeitgleiche Ausführung von $C_1,\ldots,C_n$
	$\mid$	**cparfor** $(i=1:n)\ \{\ C_1\ \}$	zeitgleiche Ausführung von n Instanzen von C_1

Der Aufbau eines Modulausdrucks ist zweistufig. Die untere Stufe, die durch das Nichtterminalsymbol C repräsentiert wird, beinhaltet Modulaufrufe und Konstruktoren zur Spezifikation einer zeitgleichen Ausführung von Modulaufrufen. Die obere Stufe, für die das Nichtterminalsymbol M verwendet wird, spezifiziert, welche Teilberechnungen nacheinander ausgeführt werden müssen und welche Teilberechnungen unabhängig voneinander sind. Die Konstruktoren der oberen Stufe sind in Anlehnung an das TwoL Modell [92] gewählt, während die Konstruktoren der unteren Stufe eine Erweiterung dieses Modells darstellen.

Modulaufrufe werden unter Angabe des Modulnamens und einer Liste aktueller Parameter $a_1,\ldots,a_n$ definiert. Die Anzahl und der Datentyp der aktuellen Parameter muss zu der zuvor definierten Modulschnittstelle passen, wobei abhängig vom spezifizierten Datentyp eines formalen Parameters die folgenden *Äquivalenzregeln* verwendet werden.

- Besitzt ein formaler Parameter einen Basisdatentyp, kann entweder eine Variable des entsprechenden Typs oder ein arithmetischer Ausdruck übergeben werden.

- Für Felddatentypen müssen Anzahl und Größe der Felddimensionen und der zugrundeliegende Basisdatentyp von aktuellem und formalem Parameter übereinstimmen (strukturelle Äquivalenz). Die Spezifikationssprache unterstützt die Verwendung von *Zugriffsdimensionen*, um Teile eines Feldes auszuwählen und als Parameter zu übergeben. Beispielsweise bewirkt ein aktueller Parameter $a[2]$ für eine $n \times m$ Matrix a die Übergabe eines Vektors der Länge m.

- Nutzerdatentypen sind nur zu sich selbst äquivalent (reine Namensäquivalenz), d.h. der Datentyp von aktuellem und formalem Parameter muss übereinstimmen.

Die *Semantik der Konstruktoren* ist wie folgt festgelegt. Die Anwendung des `seq`-Konstruktors auf eine Liste von Teilmodulausdrücken $M_1,\ldots,M_n$ erfordert die vollständige Abarbeitung aller durch M_i definierten Teilberechnungen, bevor eine durch M_{i+1} repräsentierte Berechnung begonnen werden kann ($i=1,\ldots,n-1$). Die `for`- und die `while`-Schleife erfordern eine Nacheinanderausführung der einzelnen Schleifeniterationen, d.h. eine Iteration muss komplett abgearbeitet werden, bevor mit der Ausführung der nächsten Iteration begonnen werden kann. Die `while`-Schleife besitzt eine zusätzliche Annotation, die eine Abschätzung der Anzahl der Schleifeniterationen angibt und zur Berechnung der Ausführungskosten der Schleife in der Schedulingphase des CM-task Compilers verwendet wird. Die zur Laufzeit der Anwendung tatsächlich ausgeführte Iterationsanzahl kann vom angegebenen Wert abweichen, woraus sich jedoch ungenaue Kostenvorhersagen für die gesamte Anwendung ergeben.

Der `par`-Konstruktor definiert eine Menge von n Teilberechnungen $M_1,\ldots,M_n$ als voneinander unabhängig. Konkret bedeutet dies, dass keine Interaktionen zwischen Modulaufrufen in den Teilausdrücken M_i und M_j mit $i \neq j$ stattfinden und keine Einschränkungen bezüglich der Ausführungsreihenfolge von M_i und M_j existieren. Analog gelten diese Festlegungen auch für die einzelnen Iterationen einer `parfor`-Schleife.

Die `cpar`- und `cparfor`-Konstruktoren erlauben die Definition von zeitgleich auszuführenden Modulaufrufen, zwischen denen Interaktionen in Form von Kommunikation während der Ausführung möglich sind.

Aufgrund der Semantik der Konstruktoren können nicht beliebige CM-task Graphen in der Spezifikationssprache abgebildet werden. Die Klasse der erzeugbaren Graphen wird in Anlehnung an serien-parallele(SP) Graphen [109] im Folgenden als **erweiterte SP-Graphen** bezeichnet und kann wie folgt induktiv definiert werden.

Definition 5 (Erweiterter SP-Graph)

(i) Ein CM-task Graph $G = (V,E)$ mit $V = \{A_1,\ldots,A_n\}$, $E = E_P \cup E_C$, $E_P = \emptyset$ und $E_C = \{(A_i,A_{i+1}) \mid i = 1,\ldots,n-1\}$ ist ein erweiterter SP-Graph.

(ii) Seien $G_1 = (V_1,E_1),\ldots,G_n = (V_n,E_n)$ erweiterte SP-Graphen mit $E_i = E_{iP} \cup E_{iC}$ für $i = 1,\ldots,n$.

(a) *Parallele Komposition*: Der CM-task Graph $G = (V,E)$ mit $V = \cup_{i=1,\ldots,n} V_i$ und $E = \cup_{i=1,\ldots,n} E_i$ ist ebenfalls ein erweiterter SP-Graph.

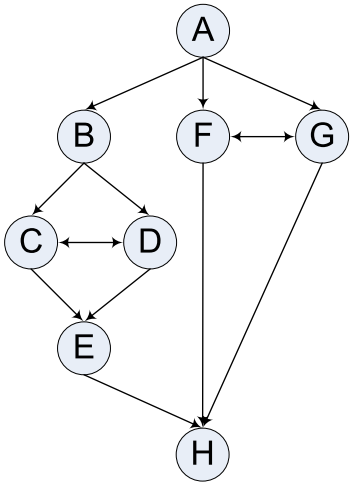
(b) *Sequentielle Komposition*: Der CM-task Graph $G = (V,E)$ mit $V = \cup_{i=1,\ldots,n} V_i$, $E = E_P \cup E_C$, $E_C = \cup_{i=1,\ldots,n} E_{iC}$ und $E_P = (\cup_{i=1,\ldots,n} E_{iP}) \cup (\cup_{i=1,\ldots,n-1} N_i \times R_{i+1})$ ist ein erweiterter SP-Graph, wobei $N_i \subseteq V_i$ die Menge der Knoten aus G_i ohne ausgehende gerichtete Kante und $R_{i+1} \subseteq V_{i+1}$ die Menge der Knoten aus G_{i+1} ohne eingehende gerichtete Kante ist. □

Fall (i) erfasst einzelne Knoten und CM-task Graphen ohne gerichtete Kanten, deren Knoten als Kette durch bidirektionale Kanten verbunden sind. In der Spezifikationssprache werden derartige Graphen durch Zusammenfassen von Modulaufrufen durch einen `cpar`-Konstruktor definiert. Die parallele Komposition (ii,a) entspricht der Anwendung des `par`-Konstruktors auf n Teilmodulausdrücke und die sequentielle Komposition (ii,b) korrespondiert mit der Anwendung des `seq`-Konstruktors auf n Teilmodulausdrücke.

Ein Beispiel für einen erweiterten SP-Graph und der entsprechende Spezifikationsausschnitt sind in Abbildung 8 dargestellt. Gemäß Fall (i) der Definition ist der Teilgraph bestehend aus den Knoten C und D und der verbindenden Kante ein erweiterter SP-Graph. Durch sequentielle Komposition des Knotens B, des zuvor genannten Teilgraphen und des Knotens E entsteht der erweiterte SP-Graph mit Knoten B,C,D und E. Eine parallele Komposition mit dem erweiterten SP-Graphen, der durch die Knoten F und G gemäß Fall (i) gebildet wird, resultiert in dem Teilgraphen, der die Knoten B,C,D,E,F und G enthält. Der letzte Schritt ist eine sequentielle Komposition des Knotens A, des zuvor genannten Graphen und des Knotens H.

Viele Anwendungen aus dem Bereich des wissenschaftlichen Rechnens besitzen eine regelmäßige Berechnungsstruktur mit aufeinander folgenden Berechnungsphasen, die unabhängige oder miteinander kommunizierende Teilberechnungen beinhalten. Beispielsweise treten diese Strukturen bei den in dieser Arbeit betrachteten Lösern für Systeme gewöhnlicher Differentialgleichungen oder den Anwendungsbenchmarks aus der Strömungsdynamik auf (s. auch Kapitel 6). Derartige Abhängigkeitsbeziehungen können direkt als erweiterter SP-Graph dargestellt und durch ein entsprechendes Spezifikationsprogramm definiert werden.

Die Spezifikation von Anwendungen mit einer unregelmäßigen Struktur setzt die Umwandlung in einen erweiterten SP-Graphen voraus. Diese Umwandlung kann durch Einfügen




```
cmmain beispiel([...]) {
  seq {
    A([...]);
    par {
      seq {
        B([...]);
        cpar {
          C([...]);
          D([...]);
        }
        E([...]);
      }
      cpar {
        F([...]);
        G([...]);
    } }
    H([...]);
} }
```

Abbildung 8: Beispiel für einen erweiterten SP-Graph mit acht Knoten (links) und Ausschnitt der zugehörigen Verbundmodulspezifikation (rechts).

zusätzlicher gerichteter Kanten in den zugehörigen CM-task Graphen erreicht werden. Ein mögliches Vorgehen ist wie folgt. Zunächst wird der gegebene CM-task Graph in den zugehörigen Supertask Graph transformiert (vgl. Abschnitt 3.2.1). Der entstehende Supertask Graph wird durch Einfügen zusätzlicher Kanten in einen SP-Graph umgewandelt. Ein Algorithmus für die Transformation allgemeiner gerichteter Graphen in SP-Graphen wird in [42] vorgestellt und dabei aufgezeigt, dass der damit einhergehende Verlust an Parallelität gering ist. Anschließend werden die Supertasks durch die enthaltenen CM-tasks mit bidirektionalen Kanten gemäß Punkt (i) von Definition 5 ersetzt und die gerichteten Kanten entsprechend angepasst.

Kommunikationsmuster zwischen Basismodulen

Die Konstruktoren der Spezifikationssprache und der daraus resultierende erweiterte SP-Graph beschreiben die möglichen Ausführungsreihenfolgen der Module eines Programmes. Die konkreten Interaktionen zwischen den Modulen werden durch die sich aus den aktuellen Parametern der Modulaufrufe ergebenden Daten- und Kommunikationsabhängigkeiten beschrieben.

Eine *Kommunikationsabhängigkeit* besteht zwischen zwei oder mehr Basismodulaufrufen, die aufgrund der verwendeten Konstruktoren (`cpar` oder `cparfor`) zeitgleich ausgeführt werden müssen und einen identischen Kommunikationsparameter besitzen. Das erzeugte Koordinationsprogramm stellt sicher, dass die an einer Kommunikationsabhängigkeit beteiligten Basismodule zur Laufzeit der Anwendung miteinander kommunizieren können. Die konkrete Implementierung der Kommunikationsoperationen erfolgt durch den Anwendungsentwickler innerhalb der Basismodulimplementierungen (s. Abschnitt 4.9 für ein Beispiel). Ein Basismodul kann mehrere Kommunikationsparameter besitzen und somit an mehreren Kommunikationsabhängigkeiten beteiligt sein. Damit können zwischen zeitgleich ausgeführten Basismodulen

```
cpar {
 C(a/* in */, b/* out */, x/* comm */);
 D(c/* in */, d/* out */, x/* comm */);
}
```

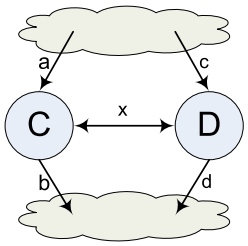


Abbildung 9: Beispiel für eine Spezifikation des Kommunikationsmusters I (links) und Illustration der resultierenden Abhängigkeiten zwischen den Modulen(rechts). Die Module C und D kommunizieren mit Hilfe des Kommunikationsparameters x, der beiden Modulen beim Aufruf zur Verfügung gestellt wird.

eine Vielzahl verschiedener Kommunikationsmuster spezifiziert werden. Im Folgenden wird eine Auswahl dieser Muster näher vorgestellt.

Kommunikationsmuster I Der einfachste Fall ist eine Kommunikation zwischen zwei zeitgleich ausgeführten Modulen. Diese Kommunikation kann sowohl einseitig als auch bidirektional ablaufen. In beiden Fällen müssen die beteiligten Module durch einen `cpar` oder `cparfor`-Konstruktor kombiniert werden und einen gemeinsamen Kommunikationsparameter verwenden. Abbildung 9 zeigt eine Möglichkeit für die Spezifikation einer derartigen Kommunikation und veranschaulicht die resultierenden Abhängigkeiten. Die Zugriffstypen der Parameter sind zum besseren Verständnis als Kommentar angegeben.

Kommunikationsmuster II In *pipelineartigen Berechnungen* wird eine feste Anzahl von Stufen verwendet, wobei jedes Datenelement der Eingabe nacheinander alle Stufen durchläuft. Daraus resultieren gerichtete Kommunikationsoperationen zwischen benachbarten Pipelinestufen zur Weitergabe der Zwischenergebnisse.

Die Spezifikation einer Pipeline, in der jede Stufe durch ein Basismodul realisiert wird, erfolgt durch Verwendung eines `cpar`-Konstruktors zur Kombination der entsprechenden Modulaufrufe. Der Datenaustausch zwischen benachbarten Stufen wird durch jeweils einen gemeinsam genutzten Kommunikationsparameter spezifiziert. Daraus folgt, dass jeder Modulaufruf (mit Ausnahme der ersten und der letzten Pipelinestufe) zwei Kommunikationsparameter besitzen muss. Ein Ausschnitt eines Beispielprogrammes mit vier Stufen ist zusammen mit den Kommunikationsabhängigkeiten in Abbildung 10 dargestellt.

Kommunikationsmuster III Bei einem *Master/Slave* Programmieransatz wird zwischen einem ausgezeichneten Master und einer Menge von Slaves unterschieden. Der Master ist für die Verwaltung der zu erledigenden Rechenaufgaben und die Verteilung von Arbeitspaketen an die Slaves verantwortlich. Die Slaves bearbeiten die zugewiesenen Aufgaben und senden das Ergebnis an den Master. Daraus resultiert eine bidirektionale Kommunikation zwischen dem Master und jedem der Slaves, während die einzelnen Slaves unabhängig voneinander arbeiten.

Im CM-task Programmiermodell können sowohl der Master als auch jeder Slave durch ein eigenes Modul dargestellt werden. Zur Realisierung der erforderlichen Kommunikationsoperationen müssen die entsprechenden Module zeitgleich ausgeführt werden. Eine mögliche

```
cpar {
 C(a1/* in */,b1/* out */,x1/* comm */);
 D(a2/* in */,b2/* out */,x1,x2/* comm */);
 E(a3/* in */,b3/* out */,x2,x3/* comm */);
 F(a4/* in */,b4/* out */,x3/* comm */);
}
```

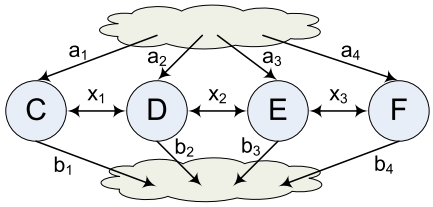


Abbildung 10: Beispielspezifikation für das Kommunikationsmuster II (links) und Veranschaulichung der resultierenden Abhängigkeiten (rechts). Die Module C, D, E und F bilden eine Pipeline und die Kommunikationsoperationen zwischen benachbarten Pipelinestufen werden durch die Kommunikationsparameter $x1, x2$ und $x3$ angegeben.

```
cpar {
  M(a/* in */, b/* out */,
      x[1],...,x[n]/* comm */);
  cparfor (i=1:n) {
    S(x[i]/* comm */);
  }
}
```

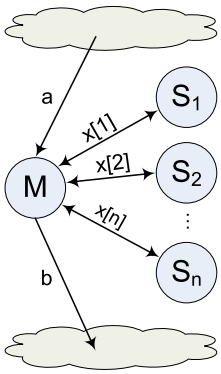


Abbildung 11: Beispiel einer Spezifikation des Kommunikationsmusters III (links) und Illustration der zugehörigen Abhängigkeiten (rechts). Die `cparfor`-Schleife definiert n Instanzen S_i des Slave-Moduls S mit jeweils einem Kommunikationsparameter $x[i], i = 1,\ldots,n$. Der `cpar`-Konstruktor definiert die zeitgleiche Ausführung des Master-Moduls M mit n Kommunikationsparametern $x[1],\ldots,x[n]$ und aller Slaves S_i. Kommunikationsoperationen sind zwischen M und S_i aufgrund des gemeinsamen Kommunikationsparameters $x[i]$ möglich, während die einzelnen Instanzen der Slaves nicht miteinander kommunizieren können.

Spezifikation eines Master/Slave Ansatzes mit den resultierenden Abhängigkeiten ist in Abbildung 11 dargestellt.

Kommunikationsmuster IV Mit Hilfe der Spezifikationssprache kann ebenfalls ein *gitterbasiertes Kommunikationsmuster* spezifiziert werden. In diesem Kommunikationsmuster wird eine feste Anzahl von Basismodulen in einem virtuellen Gitter angeordnet, wobei Basismodule, die in einer Gitterdimension benachbart sind, miteinander kommunizieren können. Die Nachbarschaftsbeziehung kann dabei auch periodisch definiert sein, d.h. dass ebenfalls eine Kommunikation zwischen dem Modul mit minimaler und dem Modul mit maximaler Gitterkoordinate in einer Dimension stattfindet.

Ein Anwendungsbeispiel für dieses Kommunikationsmuster sind Simulationsprogramme mit einem globalen Lösungsgebiet, das in einer oder mehreren Koordinatenrichtungen in teilweise

```
cparfor (i=0:m-1) {
 cparfor (j=0:n-1) {
  C(i, j/* in */,
    x[i][j]/* comm, rechts */,
    y[i][j]/* comm, unten */,
    x[i][(j-1)%n]/* comm, links */,
    y[(i-1)%m][j]/* comm, oben */);
 }
}
```

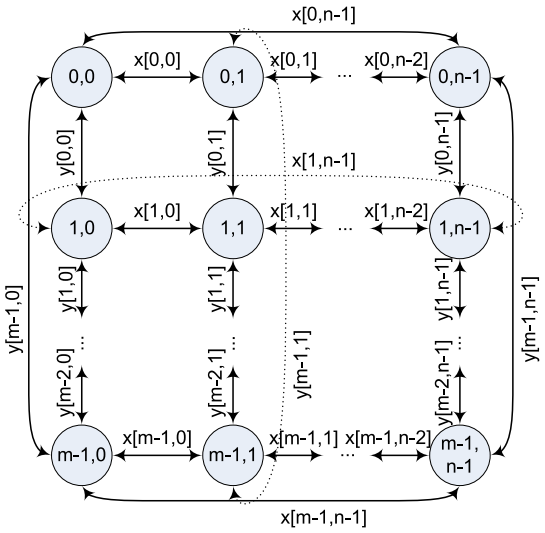


Abbildung 12: Beispiel für eine Spezifikation des Kommunikationsmusters IV (links) und Illustration der entstehenden Kommunikationsabhängigkeiten (rechts). Die `cparfor`-Schleifen erzeugen nm Instanzen des Basismoduls C, die in einem virtuellen $n \times m$ Gitter angeordnet sind. Der Kommunikationsparameter x definiert einen Datenaustausch zwischen Modulen mit benachbarten horizontalen Gitterkoordinaten und der Kommunikationsparameter y spezifiziert Kommunikation zwischen vertikal benachbarten Modulen.

überlappende Teilgebiete aufgeteilt ist und in regelmäßigen Abständen ein Randwertaustausch zwischen den Teilgebieten benötigt wird (s. auch die in Abschnitt 6.2.2 betrachteten Anwendungsbeispiele).

Abbildung 12 veranschaulicht die Spezifikation eines zweidimensionalen Gitters mit periodischer Kommunikation. Da jedes Modul im zweidimensionalen Fall Daten mit vier benachbarten Modulen austauscht, werden für jeden Modulaufruf vier Kommunikationsparameter benötigt, wobei jeder Parameter mit genau einem Nachbarmodul gemeinsam genutzt wird.

Kommunikationsmuster V Die bisher betrachteten Kommunikationsmuster basieren auf Kommunikationsoperationen, die zwischen jeweils zwei Modulen ausgeführt werden. Zusätzlich erlaubt die Spezifikationssprache auch die Definition eines *kollektiven Datenaustauschs* zwischen mehr als zwei zeitgleich ausgeführten Modulen, um bspw. ein berechnetes Zwischenergebnis eines Moduls global zur Verfügung zu stellen.

Die Spezifikation dieses Kommunikationsmusters erfolgt durch Verwendung eines identischen Kommunikationsparameters für alle beteiligten Modulaufrufe. Ein Beispiel für die Definition kollektiver Kommunikation (z.B. eines Broadcasts) zwischen n Modulen ist in Abbildung 13 dargestellt.

Kommunikationsmuster VI Ein Spezialfall des Kommunikationsmusters V ist die *orthogonale Kommunikation*, bei der nur bestimmte Prozessoren von zeitgleich ausgeführten Modulen Daten untereinander austauschen. Orthogonale Kommunikation tritt bspw. bei Gleichungslösern

```
cparfor (i=1:n) {
  C(i,a[i]/* in */,
      b[i]/* out */,
      x/* comm */);
}
```

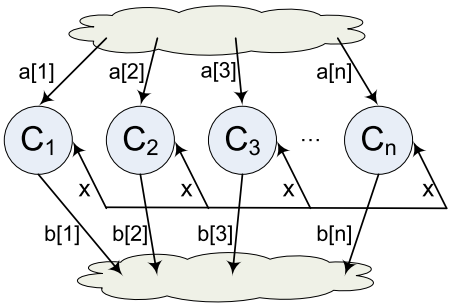


Abbildung 13: Beispiel für eine Spezifikation des Kommunikationsmusters V (links) und Illustration der zugehörigen Kommunikationsabhängigkeiten (rechts). Die `cparfor`-Schleife definiert n Instanzen $C_1, \ldots, C_n$ des Basismoduls C, die den identischen Kommunikationsparameter x verwenden. Daraus resultiert eine einzige Kommunikationsabhängigkeit, die alle n Modulinstanzen umfasst.

für gewöhnliche Differentialgleichungssysteme auf (s. auch Abschnitt 4.1.5 für ein Beispiel). Der Datenaustausch zwischen den Modulen dieser Anwendungsprogramme kann durch eine Anordnung der Prozessoren in einem virtuellen zweidimensionalen Gitter optimiert werden, falls jedes Modul durch eine identische Prozessorzahl ausgeführt wird [93, 100]. Jeder Prozessor ist dabei an zwei verschiedenen Kommunikationstypen beteiligt:

(i) der Kommunikation zwischen Prozessoren, die dasselbe Modul ausführen, und

(ii) der Kommunikation zwischen Prozessoren mit derselben Position innerhalb eines Moduls.

Eine Illustration ist in Abbildung 14(links) für drei Module mit jeweils sechs Prozessoren angegeben.

Die Spezifikation des orthogonalen Kommunikationsmusters erfolgt analog zu Kommunikationsmuster V, d.h. es wird ein allgemeiner kollektiver Datenaustausch spezifiziert, so dass das CM-task Compilerframework den Basismodulen einen externen Kommunikator mit den Prozessoren aller beteiligter Module bereitstellt. Zur Realisierung orthogonaler Kommunikation muss innerhalb der Basismodule zunächst ein Kommunikator erzeugt werden, der die Prozessoren mit identischer Position innerhalb der die Module ausführenden Prozessorgruppen enthält. Abbildung 14(rechts) zeigt ein Beispiel einer Spezifikation und einer Basismodulimplementierung. Eine genauere Beschreibung der Basismodulparameter wird in Abschnitt 4.9 gegeben.

4.1.5. Beispiel: Iterierte Runge-Kutta Verfahren

Als Beispiel für die Spezifikation eines CM-task Programmes werden *iterierte Runge-Kutta (IRK) Verfahren* betrachtet. IRK Verfahren sind explizite Lösungsverfahren für Anfangswertprobleme nichtsteifer Systeme von gewöhnlichen Differentialgleichungen (ordinary differential equations, ODEs), die speziell für eine parallele Verarbeitung entwickelt wurden [97]. Die Grundlage bilden implizite Runge-Kutta Verfahren, die in jedem Zeitschritt s Stufenvektoren durch ein System impliziter Gleichungen berechnen. Die Anzahl der berechneten Stufenvektoren wird durch das zugrundeliegende RK Verfahren festgelegt. IRK Verfahren berechnen

```
/* Spezifikation */
cparfor (i=1:3) { C(i, ort/* comm */); }

/* Basismodulimplementierung */
void C(int i, CMC_COMM_DESC ort,
     MPI_Comm gcomm) {
 int me;   MPI_Comm ortcomm;
 /* Gruppenposition bestimmen */
 MPI_Comm_rank(gcomm, &me);
 /* orthogonalen Kommunikator erzeugen */
 MPI_Comm_split(ort.ecomm,me,0,&ortcomm);
 /* interner Broadcast */
 MPI_Bcast(&i,...,gcomm);
 /* orthogonaler Broadcast */
 MPI_Bcast(&i,...,ortcomm);
 MPI_Comm_free(&ortcomm);
}
```

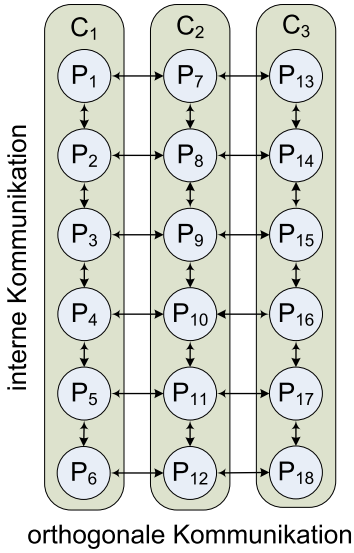


Abbildung 14: Beispiel für die Spezifikation und Implementierung eines orthogonalen Kommunikationsmusters (links) und Illustration der resultierenden Kommunikationsoperationen (rechts). Drei Basismodule C_1, C_2 und C_3 werden auf den Prozessorgruppen $\{P_1,\ldots,P_6\}$, $\{P_7,\ldots,P_{12}\}$ bzw. $\{P_{13},\ldots,P_{18}\}$ ausgeführt. Interne Kommunikation findet innerhalb der jeweiligen ausführenden Prozessorgruppe statt, während orthogonale Kommunikation zwischen Prozessoren mit derselben Position innerhalb der CM-tasks (z.B. in der Gruppe $\{P_1,P_7,P_{13}\}$) stattfindet. Die Spezifikation erfolgt analog zum Kommunikationsmuster V (s. Abbildung 13) durch einen `cparfor`-Konstruktor. Die Basismodulimplementierung erhält als Parameter eine Kommunikationsstruktur `ort` (s. Abschnitt 4.6 für den Aufbau), die einen externen Kommunikator `ort.ecomm` mit allen Prozessoren ($\{P_1,\ldots,P_{18}\}$) enthält, und einen Gruppenkommunikator `gcomm`, der die ausführenden Prozessoren des jeweiligen Basismoduls enthält. Anhand der Position im Gruppenkommunikator (`me`) wird ein Kommunikator `ortcomm` für orthogonale Kommunikationsoperationen erzeugt.

Approximationen $\mu^l_{(j)}$ dieser Stufenvektoren mit m Fixpunktiterationsschritten und sind durch folgendes Berechnungsschema gekennzeichnet:

$$\mu^l_{(0)} = f(x_k, y_k), \quad l = 1,\ldots,s \tag{1}$$

$$\mu^l_{(j)} = f\left(x_k + c_l h_k, y_k + h_k \sum_{i=1}^{s} a_{li}\mu^i_{(j-1)}\right), \quad l = 1,\ldots,s, j = 1,\ldots,m \tag{2}$$

$$y_{k+1} = y_k + h_k \sum_{l=1}^{s} b_l \mu^l_m \tag{3}$$

Dabei geben h_k die Schrittweite, x_k den Zeitindex und y_k den Näherungsvektor des ODE Systems im k-ten Zeitschritt an. Die Koeffizienten $A = (a_{li}) \in \mathbb{R}^{s\times s}$, $b = (b_1,\ldots,b_s) \in \mathbb{R}^s$

und $c = (c_1, \ldots, c_s) \in \mathbb{R}^s$ beschreiben das zugrundeliegende RK Verfahren. Die Funktion $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ definiert das ODE System der Dimension n. Für die Schrittweitenkontrolle, die nicht im Berechnungsschema aufgeführt ist, werden die Stufenvektorapproximationen $\mu^l_{(m)}$ und $\mu^l_{(m-1)}$ verwendet.

Eine parallele Implementierung von IRK Verfahren kann zwei Quellen potentieller Parallelität ausnutzen. Einerseits kann die Berechnung der n Komponenten des ODE Systems parallelisiert werden und andererseits kann die Berechnung der s Stufenvektoren eines Fixpunktiterationsschrittes parallel ausgeführt werden. Listing 4 zeigt ein Spezifikationsprogramm für IRK Verfahren, das beide Parallelitätsquellen ausnutzt. Im ersten Spezifikationsteil (Zeilen 1-3) werden Konstanten für die Anzahl der berechneten Stufenvektoren `s`, die Größe des ODE Systems `n` und die Anzahl der Fixpunktiterationsschritte `m` des Verfahrens definiert.

Datentypen werden in den Zeilen 4-6 für Vektoren der Länge `1` zur Darstellung des Zeitindexes und der Schrittweite (Typname: `scalar`), für Vektoren der Länge `n` zur Darstellung eines einzelnen Stufenvektors (Typname: `vector`) und für `s` × `n` Matrizen zur Repräsentation aller `s` Stufenvektoren (Typname: `svectors`) definiert. Der Datentyp `scalar` stellt analog zum Basisdatentyp `double` einzelne Gleitkommazahlen dar, kann aber im Gegensatz zu diesem Basisdatentyp als Ausgabeparameter eines Modulaufrufs verwendet werden.

Die Datenverteilungstypdefinitionen (Zeilen 7-9) umfassen die Spezifikation einer replizierten Verteilung für skalare Variablen (Typname: `sreplic`), einer blockweisen Verteilung für Vektoren (Typname: `vblock`) und einer spaltenblockweisen Verteilung für Stufenvektormatrizen (Typname: `sblock`). Die spaltenblockweise Verteilung ordnet jedem Prozessor $\lceil \frac{n}{p} \rceil$ aufeinanderfolgende Elemente jedes Stufenvektors zu.

Basismodule werden zur Initialisierung der Schrittweite (`init_step`), zur Berechnung eines Stufenvektors (`stage_vector`), zur Aktualisierung des Näherungsvektors eines Zeitschrittes (`compute_approx`) und zur Schrittweitenkontrolle (`step_control`) definiert.

Das Modul `init_step` (Zeile 10) initialisiert den Zeitindex `x`, berechnet eine erste Schrittweite `h` und legt eine Kopie des eingegebenen Näherungsvektors `y_k` im Hilfsvektor `y_k1` an. Die Kostenausdrücke im Spezifikationsprogramm, die für Basismodule nach dem Schlüsselwort `runtime` spezifiziert werden, basieren auf den Herleitungen symbolischer Laufzeitformeln für IRK Verfahren in [93] und werden in Abschnitt 4.2 näher betrachtet.

Das Modul `stage_vector` (Zeile 11) implementiert Gleichungen (1) und (2) zur Berechnung eines Stufenvektors μ^l, der durch den Eingabeparameter `l` bestimmt wird. Die Ausgabe besteht aus den Vektoren $\mu^l_{(m)}$ und $\mu^l_{(m-1)}$, die durch die Variablen `mu_l` bzw. `mu1_l` repräsentiert werden. Als Ausgabedatenverteilung der berechneten Vektoren wird eine blockweise Verteilung (`vblock`) festgelegt. Der Kommunikationsparameter `ort` erlaubt dem Modul, während der Ausführung Daten mit anderen Modulen auszutauschen.

Das Modul `compute_approx` (Zeile 12) beinhaltet Gleichung (3) zur Aktualisierung des Näherungsvektors `y_k` basierend auf allen `s` berechneten Stufenvektoren, die durch den Eingabeparameter `mu` zur Verfügung gestellt werden.

Die Schrittweitenkontrolle im Modul `step_control` (Zeile 13) entscheidet, ob der neu berechnete Näherungsvektor einer vorgegebenen Genauigkeit genügt, oder ob der Zeitschritt mit einer geringeren Schrittweite wiederholt werden muss. In ersterem Fall wird die Kopie des Näherungsvektors in `y_k1` aktualisiert, wohingegen in letzterem Fall die zuvor gespeicherte Kopie in `y_k` wiederhergestellt wird. In beiden Fällen wird eine neue Schrittweite `h` und ein neuer Zeitindex `x` ausgegeben.

Listing 4: Spezifikationsprogramm für Iterierte Runge-Kutta Verfahren.

```
   // Konstantendefinitionen
1  const s = 3;                    // Anzahl Stufenvektoren
2  const n = 1000;                 // Größe des ODE Systems
3  const m = 5;                    // Anzahl Fixpunktiterationsschritte

   // Datentypdefinitionen
4  type scalar = array[1] of double;          // Gleitkommazahl
5  type vector = array[n] of double;          // Vektor der Länge n
6  type svectors = array[s][n] of double;     // s×n Matrix

   // Datenverteilungstypdefinitionen
7  distrib scalar:sreplic = [replic on p];            // repliziert
8  distrib vector:vblock = [block on p];              // blockweise
9  distrib svectors:sblock = [block on 1][block on p];// spaltenblock

   // Basismoduldefinitionen
   // Initialisierung
10 cmtask init_step (x,h:scalar:out:sreplic, y_k:vector:in,
     y_k1:vector:out)
    runtime n/p*T_op+n/p*T_eval+T_ar(p,1)+T_ag(p,n/p);
   // Berechnung eines Stufenvektors l
11 cmtask stage_vector (l:int, x,h:scalar:in:sreplic, y_k:vector:in,
     mu1_l,mu_l:vector:out:vblock, ort:vector:comm) runtime
     m*n/p*T_eval+m*(s+1)*n/p*T_op+m*T_ag(p,n/p)+m*T_ag(s,n/p);
   // Berechnung des Näherungsvektors
12 cmtask compute_approx (h:scalar:in:sreplic, y_k:vector:inout,
     mu:svectors:in:sblock) runtime n*s/p*T_op;
   // Schrittweitenkontrolle
13 cmtask step_control(x,h:scalar:inout:sreplic,y_k,y_k1:vector:inout,
     mu,mu1:svectors:in:sblock) runtime n*s/p*T_op+T_ar(p,1);

   // Hauptmoduldefinition
14 cmmain irk (y_k:vector:inout) {
    // Deklaration lokaler Variablen
15  var x, h : scalar;        // Zeitindex und Schrittweite
16  var y_k1 : vector;        // Kopie des Näherungsvektors
17  var ort : vector;         // Vektor für orthogonale Kommunikation
18  var mu, mu1 : svectors;   // Stufenvektoren
19  var i : int;

    // Modulausdruck
20  seq {
21   init_step (x,h,y_k,y_k1);                    // Initialisierung
22   while (x[0]<X)#100 {          // Schleife mit 100 Zeitschritten
23    seq {
24     cparfor (i = 0:s-1) {  // Schleife zur Stufenvektorberechnung
25      stage_vector (i,x,h,y_k,mu1[i],mu[i], ort);
26     }
27     compute_approx (h,y_k,mu);    // Berechnung neue Approximation
28     step_control (x,h,y_k,y_k1,mu,mu1); // Schrittweitenkontrolle
29 }}}}
```

Die Definition des Hauptmoduls `irk` (Zeile 14) beinhaltet die Definition von lokalen Variablen zur Speicherung von Zwischenergebnissen und einen Modulausdruck, der die Abhängigkeiten der Modulaufrufe beschreibt. Die `while`-Schleife (Zeile 22) führt die Zeitschritte des IRK Verfahrens aus und wird abgebrochen, wenn ein vorgegebener Zeitindex `X` erreicht wird. Der Zugriff auf den aktuellen Zeitindex erfolgt durch `x[0]`, da zur Abspeicherung ein Vektor der Länge `1` verwendet wird. Die Größenordnung für die Anzahl der ausgeführten Schleifeniterationen wird mit `100` angegeben. Bei der tatsächlichen Ausführung der Anwendung können abhängig von `X` und dem Konvergenzverhalten des Verfahrens abweichende Iterationsanzahlen auftreten.

Der Schleifenrumpf der `while`-Schleife beinhaltet eine `cparfor`-Schleife (Zeile 24) zur Spezifikation von `s` Aufrufen des Basismoduls `stage_vector`, wobei jeder Aufruf einen Stufenvektor berechnet. Der gemeinsam verwendete Kommunikationsparameter `ort` beschreibt einen Datenaustausch zwischen diesen `s` Modulinstanzen. Ein Datenaustausch ist erforderlich, da jeder Fixpunktiterationsschritt von den Stufenvektoren des vorherigen Schrittes abhängig ist (vgl. Gleichung (2)). Die Realisierung dieses Datenaustauschs erfolgt durch orthogonale Kommunikation (Kommunikationsmuster VI, s. Abbildung 14 auf Seite 48).

Die nachfolgenden Aufrufe der Module `compute_approx` (Zeile 27) und `step_control` (Zeile 28) benötigen alle berechneten Stufenvektoren und müssen aufgrund vorhandener Datenabhängigkeiten nacheinander ausgeführt werden.

Die schrittweise Transformation des angegebenen Spezifikationsprogrammes in ein ausführbares Koordinationsprogramm wird in den Abschnitten 4.3, 4.4, 4.5 und 4.6 beschrieben. Kapitel 6 enthält Laufzeitmessungen auf verschiedenen Plattformen und einen Vergleich der durch den CM-task Compiler erzeugten Programmversion mit einer datenparallelen Implementierung, die alle Stufenvektoren nacheinander durch alle Prozessoren berechnet.

4.2. Kostenmodell und Plattformbeschreibung

Der Transformationsprozess des CM-task Compilers beruht auf einem Kostenmodell, durch das die Auswirkungen verschiedener Entscheidungen in der Schedulingphase auf die Laufzeit der Anwendung abgeschätzt wird. Kosteninformationen werden für die Ausführung von Basis- und Verbundmodulen und für die Kommunikationsoperationen zwischen diesen Modulen benötigt. Die Kosten von Basismodulen werden durch den Anwender spezifiziert, wohingegen die Ausführungskosten von Verbundmodulen und Datenumverteilungsoperationen im CM-task Compiler berechnet werden.

Die Definition der Basismodulkosten erfolgt in Form *symbolischer Laufzeitformeln* in geschlossener Form [60, 95]. Eine symbolische Laufzeitformel T_A für ein Basismodul A beschreibt die Ausführungszeit von A in Abhängigkeit von der Anzahl ausführender Prozessoren p. Der Aufbau von T_A spiegelt die interne Berechnungs- und Kommunikationsstruktur von A wider, d.h. T_A besitzt die Form

$$T_A(p) = \frac{ops(A)}{p} * T_{op} + T_{comm}(A, p),$$

wobei ops die Anzahl arithmetischer Operationen von A angibt, T_{op} die mittlere Ausführungszeit einer arithmetischen Operation auf der Zielplattform ist und T_{comm} die Summe der internen Kommunikationszeiten von A angibt.

Die Kommunikationszeit T_{comm} hängt von der Anzahl und der Art der ausgeführten Kommunikationsoperationen ab und wird basierend auf Kostenformeln für die einzelnen Kommunikationsprimitive beschrieben. Die durch die MPI Bibliothek [68] zur Verfügung gestellten Primitive umfassen Punkt-zu-Punkt Kommunikationsoperationen, deren Laufzeit von der zu übertragenden Datenmenge b abhängt, und kollektive Kommunikationsoperationen, deren Laufzeit zusätzlich von der Anzahl der beteiligten Prozessoren p abhängt. Beispielsweise kann die Ausführungszeit einer Broadcastoperation, die zur Kommunikation einen Binomialbaum der Tiefe $\log(p)$ verwendet, durch eine Funktion der Form

$$T_{bc}(p,b) = (\tau + t_c * \log(p)) * b$$

abgeschätzt werden. Dabei geben τ und t_c hardwareabhängige Parameter an, die durch Kurvenanpassung von Messwerten ermittelt werden können.

Eine geeignete Laufzeitformel für ein gegebenes Basismodul kann per Hand erstellt oder mit Hilfe eines Softwarewerkzeugs wie SCAPP [59] aus dem Programmquelltext extrahiert werden. Für die Berechnung eines Stufenvektors in einem s-stufigen iterierten Runge-Kutta(IRK) Verfahren mit m Fixpunktiterationsschritten wurde bspw. die Laufzeitformel

$$T_{stage\_vector}(p) = \frac{n*m}{p} * T_{eval} + \frac{m*(s+1)*n}{p} * T_{op} + m * T_{ag}(p, \frac{n}{p}) + m * T_{ag}(s, \frac{n}{p})$$

ermittelt [93]. Dabei entspricht T_{eval} der Auswertungszeit einer Komponente des ODE Systems der Größe n und $T_{ag}(p,b)$ beschreibt die Laufzeit einer Multibroadcast-Operation (MPI_Allgather()) in Abhängigkeit von der Prozessoranzahl p und der Datenmenge b.

Das konkrete Aussehen einer Laufzeitformel ist von der verwendeten Zielplattform unabhängig und wird daher in der Spezifikationssprache zusammen mit dem entsprechenden Basismodul definiert (s. Abschnitt 4.1.5 für ein Beispiel). Die hardwareabhängigen Parameter werden in einer separaten, weiter unten vorgestellten Plattformbeschreibung angegeben.

Die Ausführungszeit eines Verbundmoduls hängt von den enthaltenen Berechnungen und den benötigten Kommunikationsoperationen ab und wird durch hierarchisches Scheduling ermittelt (s. Abschnitt 5.2). Die Kosten von Datenumverteilungsoperationen hängen von der übertragenen Datenmenge und der Punkt-zu-Punkt Kommunikationsleistung der Zielplattform ab. Die Datenmenge wird aus den spezifizierten Felddatenverteilungstypen ermittelt, während die Kommunikationsleistung durch die im Folgenden besprochene Plattformbeschreibung definiert wird. Für Datenumverteilungsoperationen unter Beteiligung von Nutzerdatentypen und Nutzerdatenverteilungen erfolgt keine Kostenberechnung, da der Aufbau dieser Typen und Verteilungen für den CM-task Compiler nicht sichtbar ist.

Plattformbeschreibung

Die Plattformbeschreibung stellt Informationen über die Anzahl der verfügbaren Prozessoren und die Rechen- und Kommunikationsleistung eines parallelen Rechensystems zur Verfügung. Die Beschreibung erfolgt in Form von Konstanten- und Funktionsdefinitionen, wobei die Konstante `P` für die verfügbare Prozessorzahl vordefiniert ist und die Punkt-zu-Punkt Kommunikationsleistung durch eine Funktion der Form `T_p2p(b)` angegeben werden muss. Diese Funktion wird durch den CM-task Compiler zur Bestimmung der Datenumverteilungskosten für Felddatenverteilungen verwendet. Weitere Konstanten und Funktionen können frei gewählt

Listing 5: Beispiel für die Beschreibung einer Plattform mit `P=48` Prozessoren. Es werden die mittlere Ausführungszeit einer arithmetischen Operation `T_op`, die Punkt-zu-Punkt Kommunikationszeit `T_p2p` abhängig von der Datenmenge `b` und die Laufzeit `T_bc` einer Broadcastoperation abhängig von der Prozessorzahl `p` und der Datenmenge `b` spezifiziert.

```
machine {
  P = 48;
  T_op = 69e-9;
  T_p2p(b) = 2e-6+12e-10*b;
  T_bc(p,b) = (0.038+0.474/1000000*log(p))*b;
}
```

und danach in den Laufzeitformeln eines Spezifikationsprogrammes genutzt werden. Listing 5 zeigt ein Beispiel für die Beschreibung einer Plattform mit `P=48` Prozessoren.

4.3. Erweiterte Spezifikationssprache

Die erweiterte Spezifikationssprache definiert die Syntax von erweiterten Spezifikationsprogrammen, die als Ausgabe der Analysephase des CM-task Compilers erzeugt werden. Die Analysephase erkennt basierend auf den spezifizierten Variablenzugriffen und den verwendeten Konstruktoren die in einem gegebenen Spezifikationsprogramm implizit vorhandenen Daten- und Kommunikationsabhängigkeiten. Abbildung 15 stellt die in der Analysephase erkannten Abhängigkeiten des Spezifikationsprogrammes für IRK Verfahren graphisch dar.

Zur expliziten Darstellung der Abhängigkeiten wird in der erweiterten Spezifikationssprache die Syntax von Verbundmoduldefinitionen gegenüber der in Abschnitt 4.1 beschriebenen Spezifikationssprache verändert. Die erweiterte Spezifikationssprache definiert zusätzliche Annotationen für Identifikatoren, Kommunikations- und Datenabhängigkeiten, die im Folgenden vorgestellt werden. Die Syntax der erweiterten Spezifikationssprache wird anhand einer kontextfreien Grammatik im Anhang Abschnitt B.2 definiert.

Annotation von Identifikatoren In der erweiterten Spezifikationssprache wird jedem Basis- und Verbundmodulaufruf sowie jedem `for`-, `while`- und `if`-Konstrukt ein eindeutiger Identifikator zugeordnet. Diese Identifikatoren werden in den annotierten Daten- und Kommunikationsabhängigkeiten zur Spezifikation der beteiligten Konstrukte verwendet. Ein Identifikator ist ein im entsprechenden Verbundmodul eindeutiger Bezeichner, der am zugehörigen Konstrukt annotiert wird. Das folgende Beispiel verdeutlicht die Annotation eines Identifikators `n1` an einem Aufruf des Moduls `A` mit aktuellen Parametern `x`, `y` und `z`, wobei die gegenüber der Spezifikationssprache hinzugefügte Annotation grau hinterlegt ist.

```
n1: A(x, y, z);
```

Annotation von Kommunikationsabhängigkeiten Eine Kommunikationsabhängigkeit besteht zwischen n Basismodulaufrufen mit einem gemeinsamen Kommunikationsparameter,

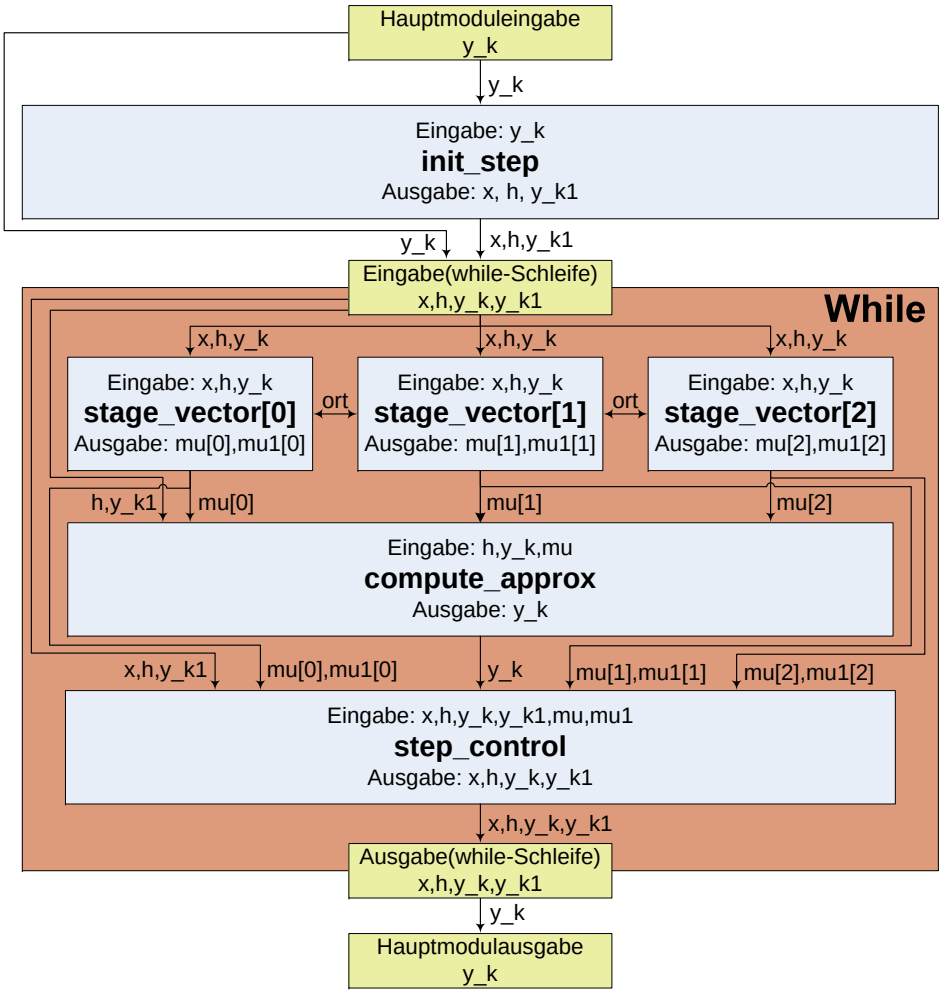


Abbildung 15: Illustration der Daten- und Kommunikationsabhängigkeiten im Hauptmodul `irk` des Spezifikationsprogrammes für Iterierte Runge-Kutta Verfahren (s. Listing 4 auf Seite 50). Die angegebenen Eingabe- und Ausgabeparameter des Hauptmoduls und der Modulaufrufe werden vom Anwendungsprogrammierer spezifiziert. Die Eingabe- und Ausgabeparameter der `while`-Schleife werden in der Analysephase des CM-task Compilers aus den Variablenzugriffen des Schleifenrumpfes bestimmt und werden intern verwendet. Diese Parameter sind für den Anwendungsprogrammierer nicht sichtbar.
Zwischen den drei Aufrufen des Moduls `stage_vector` besteht eine Kommunikationsabhängigkeit für die Variable `ort`, die durch zwei bidirektionale Kanten dargestellt ist.
Datenabhängigkeiten verlaufen entweder komplett außerhalb oder komplett innerhalb der `while`-Schleife und verbinden einen Schreibzugriff auf eine Variable mit einem nachfolgend auszuführenden Lesezugriff. Für den Lesezugriff auf die Variable `mu` durch das Modul `compute_approx` werden drei Datenabhängigkeiten (für `mu[0]`, `mu[1]` und `mu[2]`) eingefügt, da durch jeden Modulaufruf von `stage_vector` nur ein Teil der für den Lesezugriff benötigten Daten zur Verfügung gestellt wird.

d.h. einem aktuellen Parameter mit Zugriffstyp `comm`. Die Annotation von Kommunikationsabhängigkeiten erfolgt in der erweiterten Spezifikationssprache durch das `crels`-Konstrukt. Der Name des Konstrukts drückt aus, dass in der Schedulingphase des CM-task Compilers aus den annotierten Kommunikationsabhängigkeiten die C-Relationen des CM-task Programmes konstruiert werden. Ein `crels`-Konstrukt enthält eine Menge von Kommunikationsabhängigkeiten, wobei eine einzelne Abhängigkeit die Form

```
(id1--id2--id3--...--idn, commvar)
```

besitzt. Dabei geben `id1,...,idn` die Identifikatoren der *n* an der jeweiligen Kommunikationsabhängigkeit beteiligten Basismodulaufrufe und `commvar` den Namen des gemeinsamen Kommunikationsparameters an. Das Trennsymbol zwischen den einzelnen Identifikatoren (`--`) symbolisiert die Symmetrie Kommunikationsabhängigkeiten.

Annotation von Datenabhängigkeiten Eine Datenabhängigkeit verbindet einen Schreibzugriff auf eine Variable mit einem nachfolgend auszuführenden Lesezugriff auf dieselbe Variable. In der erweiterten Spezifikationssprache werden Datenabhängigkeiten innerhalb des `prels`-Konstrukt definiert. Der Name des Konstrukts drückt aus, dass die annotierten Datenabhängigkeiten die P-Relationen des CM-task Programmes beschreiben. Ein `prels`-Konstrukt enthält eine Menge von Datenabhängigkeiten, die jeweils die Form

```
(source->target, {var1,...,varm})
```

besitzen. Dabei gibt `source` den Identifikator des Schreibzugriffes und `target` den Identifikator des Lesezugriffes der Abhängigkeit an. Als Trennsymbol wird ein Pfeil (`->`) verwendet, der die Richtung der Abhängigkeit anzeigt. Die Menge von Variablen {`var1,...,varm`} gibt *m* Ausgabewerte des Schreibzugriffes an, die als Eingabe für den Lesezugriff der Abhängigkeit benötigt werden.

Beispiel: Iterierte Runge-Kutta Verfahren

Das erweiterte Spezifikationsprogramm für IRK Verfahren unterscheidet sich vom ursprünglichen Spezifikationsprogramm (s. Listing 4 auf Seite 50) nur in der Definition des Hauptmoduls `irk`. Listing 6 zeigt diese veränderte Hauptmoduldefinition. Die hinzugefügten Identifikatoren für die Modulaufrufe und die `while`-Schleife sind automatisch aus der hierarchischen Struktur des Modulausdrucks generierte Namen. Die annotierten Daten- und Kommunikationsabhängigkeiten sind in Abbildung 15 graphisch dargestellt.

Das erweiterte Spezifikationsprogramm enthält eine Kommunikationsabhängigkeit (Zeile 9), die zwischen den `s=3` Instanzen des Basismodulaufrufs `stage_vector` aufgrund des gemeinsamen Kommunikationsparameters `ort` besteht. Zur Auswahl der genauen Instanz des Modulaufrufs wird zusätzlich zum entsprechenden Identifikator die Schleifeniteration der umgebenden `cparfor`-Schleife angegeben. Beispielsweise beschreibt der Identifikator `n2_1[0]` den Basismodulaufruf in der Iteration `i=0`.

Datenabhängigkeiten werden an vier verschiedenen Stellen des Spezifikationsprogrammes eingefügt:

- Das erste `prels`-Konstrukt (Zeile 13) ist Bestandteil des `seq`-Konstruktors innerhalb der `while`-Schleife und enthält Datenabhängigkeiten, die zwischen Modulaufrufen derselben Iteration dieser Schleife bestehen.

Listing 6: Hauptmoduldefinition des erweiterten Spezifikationsprogrammes für Iterierte Runge-Kutta Verfahren. Die in der Analysephase hinzugefügten Annotationen sind grau hinterlegt und Auslassungen sind durch `[...]` gekennzeichnet.

```
// Hauptmoduldefinition
cmmain irk (y_k:vector:inout) {
 // Deklaration lokaler Variablen
 [...]

 // Modulausdruck
 seq {
  n1: init_step (x,h,y_k,y_k1);
  n2: while (x[0] < X)#100 {
   seq {
    cparfor (i = 0:s-1) {
     n2_1: stage_vector (i,x,h,y_k,mu1[i],mu[i], ort);
      /* Kommunikationsabhängigkeit zwischen den s=3 Instanzen
         des Modulaufrufs stage_vector. Die konkrete Iteration
         der cparfor-Schleife wird zusätzlich zum Identifikator
         in eckigen Klammern angegeben */
      crels { (n2_1[0]--n2_1[1]--n2_1[2],ort) }
     }
    n2_2: compute_approx (h,y_k,mu);
    n2_3: step_control (x,h,y_k,y_k1,mu,mu1);
   /* Datenabhängigkeiten zwischen Modulaufrufen innerhalb
      der while-Schleife */
    prels {
     (n2_1[0]->n2_2,{mu[0]}),(n2_1[1]->n2_2,{mu[1]}),
     (n2_1[2]->n2_2,{mu[2]}),(n2_1[0]->n2_3,{mu[0],mu1[0]}),
     (n2_1[1]->n2_3,{mu[1],mu1[1]}),
     (n2_2_1[2]->n2_3,{mu[2],mu1[2]}),(n2_2->n2_3,{y_k})
    }
   }
   // Datenabhängigkeiten für Ein- und Ausgabe der while-Schleife
   prels {
    (n2->n2_1[0],{x,h,y_k}), (n2->n2_1[1],{x,h,y_k}),
    (n2->n2_1[2],{x,h,y_k}), (n2->n2_2,{h,y_k}),
    (n2->n2_3,{x,h,y_k1}), (n2_3->n2,{x,h,y_k,y_k1})
   }
  }
  // Datenabhängigkeiten außerhalb der while-Schleife
  prels {(n1->n2,{x,h,y_k1})}
 }
 /* Datenabhängigkeiten für Ein- und Ausgabeparameter
    des Hauptmoduls irk */
 prels {(irk->n1,{y_k}), (irk->n2,{y_k}), (n2->irk,{y_k})}
}
```

- Das zweite `prels`-Konstrukt (Zeile 20) ist Bestandteil der `while`-Schleife und enthält Datenabhängigkeiten, die Eingabe- oder Ausgabeparameter der Schleife mit Modulaufrufen des Schleifenrumpfes verbinden. Die Eingabe- und Ausgabeparameter der Schleife werden durch die Analysephase des CM-task Compilers aus den Variablenzugriffen des Schleifenrumpfes ermittelt. Diese Parameter werden nicht im erweiterten Spezifikationsprogramm ausgegeben und sind somit nicht für den Anwendungsprogrammierer sichtbar. Im Beispiel sind die Variablen `x`, `h`, `y_k` und `y_k1` sowohl Eingabe- als auch Ausgabeparameter, da die in einer Iteration j geschriebenen Werte in Iteration $j+1$ gelesen werden.
- Das dritte `prels`-Konstrukt (Zeile 26) ist Bestandteil des `seq`-Konstruktors außerhalb der `while`-Schleife und enthält die Datenabhängigkeit zwischen dem Basismodulaufruf `init_step` und der Eingabe der `while`-Schleife.
- Das vierte `prels`-Konstrukt (Zeile 28) beinhaltet alle Datenabhängigkeiten, an denen Eingabe- oder Ausgabeparameter des Hauptmoduls beteiligt sind. In diesen Abhängigkeiten wird der Hauptmodulname als Identifikator für den Schreibzugriff (Hauptmoduleingabeparameter) bzw. den Lesezugriff (Hauptmodulausgabeparameter) verwendet.

4.4. Rahmenprogramm

Das Rahmenprogramm legt fest, wie eine CM-task Anwendung auf einer bestimmten Zielplattform ausgeführt werden soll. Dies beinhaltet sowohl die Festlegung der Ausführungsreihenfolge der spezifizierten Modulaufrufe als auch die Definition der Prozessorgruppen, die zur Ausführung der Modulaufrufe verwendet werden sollen. Zur Darstellung der Ausführungsreihenfolge und der verwendeten Prozessorgruppen wird die Syntax und die Semantik von Verbundmoduldefinitionen im Rahmenprogramm gegenüber dem erweiterten Spezifikationsprogramm verändert. Im Folgenden werden diese Veränderungen vorgestellt und anschließend das mit dem CM-task Compiler erzeugte Rahmenprogramm für IRK Verfahren beschrieben. Die Syntax des Rahmenprogrammes wird anhand einer kontextfreien Grammatik im Anhang Abschnitt B.3 definiert.

Festlegung der Ausführungsreihenfolge Die Ausführungsreihenfolge von Programmfragmenten wird durch die Konstruktoren des Rahmenprogrammes festgelegt. Diese Konstruktoren haben dieselbe Syntax wie die Konstruktoren der Spezifikationssprache, besitzen aber eine veränderte Semantik. Die Konstruktoren des Spezifikationsprogrammes definieren die potentielle Parallelität einer Anwendung, d.h. mögliche Ausführungsreihenfolgen der durch die Konstruktoren zusammengefassten Programmfragmente. Im Gegensatz dazu definieren die Konstruktoren des Rahmenprogrammes die tatsächlich ausgenutzte Parallelität, d.h. die tatsächliche Ausführungsreihenfolge bei Abarbeitung der Anwendung.

Dies bedeutet konkret, dass durch einen `seq`-Konstruktor kombinierte Programmfragmente nacheinander ausgeführt werden. Die `par`- und `cpar`-Konstruktoren beschreiben beide eine gleichzeitige Ausführung auf disjunkten Prozessorgruppen, wobei letztere zusätzlich eine Kommunikation zwischen den enthaltenen Basismodulaufrufen erlauben. Analoge Festlegungen gelten auch für die Schleifen, d.h. die Iterationen von `parfor`- und `cparfor`-Schleifen werden zeitgleich abgearbeitet, wohingegen `for`- und `while`-Schleifen eine Nacheinanderausführung der Schleifeniterationen festlegen.

Darstellung von Prozessorgruppen Eine Prozessorgruppe wird im Rahmenprogramm als eine Menge natürlicher Zahlen dargestellt. Diese Zahlen korrespondieren mit den eindeutigen Prozessornummern, die in einem MPI Programm vergeben werden. Im Unterschied zur MPI Bibliothek, die fortlaufende Prozessornummern beginnend mit 0 vergibt, beginnt die Nummerierung im Rahmenprogramm mit 1. Die Abkürzung von Teilfolgen mit direkt aufeinanderfolgenden Prozessornummern wird durch den Operator `..` unterstützt. Beispielsweise definiert

```
{1..4, 6..7, 10}
```

eine Prozessorgruppe bestehend aus den sieben Prozessoren `1,2,3,4,6,7,10`.

Annotation von Prozessorgruppen Ein Rahmenprogramm enthält gegenüber dem erweiterten Spezifikationsprogramm zusätzliche Annotationen von Prozessorgruppen, die an den folgenden Stellen innerhalb einer Verbundmoduldefinition auftreten.

(i) An der Definition des Verbundmoduls wird eine Prozessorgruppe der Form `{1..q}` annotiert, wobei `q` die Anzahl der zur Ausführung des jeweiligen Verbundmoduls benötigten Prozessoren angibt. Diese Prozessorgruppe stellt zugleich eine Obermenge jeder innerhalb der Verbundmoduldefinition annotierten Prozessorgruppe (Punkte (ii) und (iii)) dar. Das folgende Beispiel demonstriert die Annotation einer Prozessorgruppe an der Definition eines Verbundmoduls `A`, das `q=12` Prozessoren zur Ausführung benötigt.

```
cmgraph A([...]) on {1..12} { [...] }
```

(ii) Die `for`-, `while`- und `if`-Konstrukte des Rahmenprogrammes besitzen eine Prozessorgruppenannotation, die die ausführenden Prozessoren des Schleifenrumpfes bzw. der Bedingungszweige festlegt. Die Abbruchbedingung der Schleife bzw. die durch das `if`-Konstrukt spezifizierte Bedingung wird zur Laufzeit nur durch die annotierten Prozessoren ausgewertet. Zur Sicherstellung einer ordnungsgemäßen Abarbeitung des erzeugten Koordinationsprogrammes müssen alle in einem Schleifenrumpf oder einem Bedingungszweig annotierten Prozessorgruppen Teilgruppen der an der entsprechenden Schleife bzw. Bedingung annotierten Prozessorgruppe sein. Das folgende Beispiel demonstriert die Annotation einer Prozessorgruppe an einer `for`-Schleife mit Identifikator `n1`.

```
n1: for (i=1:n) on {5..10} { [...] }
```

(iii) Die Prozessorgruppenannotation an Basis- und Verbundmodulaufrufen legt fest, welche Prozessoren zur Laufzeit den spezifizierten Modulaufruf ausführen. Für Verbundmodulaufrufe entspricht die Anzahl der annotierten Prozessoren der Anzahl der an der Definition des aufgerufenen Verbundmoduls annotierten Prozessoren (s. Punkt (i)). Das folgende Beispiel demonstriert die Annotation einer Prozessorgruppe bestehend aus `12` Prozessoren am Aufruf eines Moduls `A`.

```
n2: A([...]) on {3..14};
```

Listing 7: Definition des Hauptmoduls `irk` im Rahmenprogramm für Iterierte Runge-Kutta Verfahren, das durch den CM-task Compiler für eine Zielplattform mit `48` Prozessoren erzeugt wurde. Die in der Schedulingphase hinzugefügten Annotationen sind grau hinterlegt und Auslassungen für die annotierten Daten- und Kommunikationsabhängigkeiten und die Deklaration lokaler Variablen durch `[...]` gekennzeichnet.

```
   // Hauptmoduldefinition
1  cmmain irk (y_k:vector:inout) on {1..48} {
     // Deklaration lokaler Variablen
2    [...]
     // Modulausdruck
3    seq {
       // Ausführung von init_step auf allen verfügbaren Prozessoren
4      n1: init_step (x,h,y_k,y_k1) on {1..48} ;
       // Ausführung der while-Schleife auf allen Prozessoren
5      n2: while (x[0] < X)#100 on {1..48} {
6        seq {
7          cparfor (i = 0:s-1) {
             /* Ausführung der drei Instanzen des Modulaufrufs
                stage_vector auf disjunkten Prozessorgruppen mit
                jeweils 16 Prozessoren */
8            n2_1: stage_vector (i,x,h,y_k,mu1[i],mu[i], ort)
9              on [{1..16},{17..32},{33..48}] ;
10           [...]
11         }
           // Ausführung von compute_approx auf allen Prozessoren
12         n2_2: compute_approx (h,y_k,mu) on {1..48} ;
           // Ausführung von step_control auf allen Prozessoren
13         n2_3: step_control (x,h,y_k,y_k1,mu,mu1) on {1..48} ;
14         [...] } [...] } [...] } [...]
15 }
```

Beispiel: Iterierte Runge-Kutta Verfahren

Listing 7 zeigt die Hauptmoduldefinition des Rahmenprogrammes für IRK Verfahren auf einer Plattform mit 48 Prozessoren, wie sie durch die Plattformbeschreibung in Listing 5 auf Seite 53 definiert wird. Zur besseren Lesbarkeit wurden die Annotationen für Daten- und Kommunikationsabhängigkeiten (`crels`- und `prels`-Konstrukte) des erweiterten Spezifikationsprogrammes ausgelassen und die hinzugefügten Prozessorgruppenannotationen grau hinterlegt.

Das Rahmenprogramm nutzt den maximalen Grad der im Spezifikationsprogramm definierten Taskparallelität aus. Die verwendeten Konstruktoren des Rahmenprogrammes legen eine zeitgleiche Ausführung der `s=3` Instanzen des Basismodulaufrufs `stage_vector` fest. Zur Ausführung dieser Instanzen werden drei disjunkte Prozessorgruppen mit jeweils 16 Prozessoren definiert, die in Form einer Liste an dem entsprechenden Modulaufruf annotiert sind. Die einzelnen Elemente dieser Liste beschreiben die Prozessorgruppen der einzelnen Instanzen des

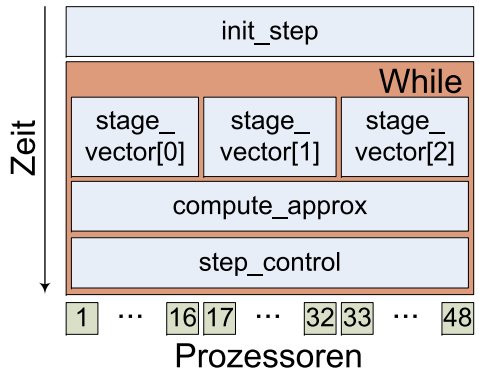


Abbildung 16: Illustration der Ausführungsreihenfolge und der verwendeten Prozessorgruppen im Rahmenprogramm für Iterierte Runge-Kutta Verfahren aus Listing 7. Das Modul `init_step` und die `while`-Schleife werden nacheinander auf allen Prozessoren abgearbeitet. Im Schleifenrumpf der `while`-Schleife werden drei gleichgroße Prozessorgruppen zur Ausführung der drei Instanzen des Basismodulaufrufs `stage_vector` genutzt. Anschließend werden `compute_approx` und `step_control` nacheinander auf allen Prozessoren abgearbeitet. Für alle ausgeführten Iterationen der `while`-Schleife werden dieselbe Ausführungsreihenfolge und dieselben Prozessorgruppen für die Modulaufrufe des Schleifenrumpfes verwendet.

Modulaufrufs. Eine graphische Darstellung der Ausführungsreihenfolge und der verwendeten Prozessorgruppen ist in Abbildung 16 angegeben.

4.5. Erweitertes Rahmenprogramm

Die Datenverteilungsphase des CM-task Compilers erzeugt ein erweitertes Rahmenprogramm, das gegenüber dem Rahmenprogramm zusätzliche Annotationen für Datenumverteilungsoperationen und (im semi-dynamischen Ansatz) für Lastausgleichsoperationen besitzt. Im Folgenden wird ein Überblick dieser Annotationen gegeben, die vollständige Syntax ist in Abschnitt B.4 des Anhangs enthalten.

Annotation von Datenumverteilungsoperationen Datenumverteilungsoperationen werden im erzeugten Koordinationsprogramm benötigt, um Eingabedaten von Modulaufrufen in der jeweils geforderten Datenverteilung auf der jeweiligen ausführenden Prozessorgruppe zur Verfügung zu stellen. Das erweiterte Rahmenprogramm legt durch Annotationen innerhalb von Verbundmoduldefinitionen fest, welche Datenumverteilungsoperationen zur Laufzeit ausgeführt werden müssen. Dazu wird die Syntax von Verbundmoduldefinitionen gegenüber dem Rahmenprogramm um ein zusätzliches `redistr`-Konstrukt erweitert, das eine Menge von Annotationen für Datenumverteilungsoperationen enthält. Eine Annotation hat die Form

```
(source:svar:sdistr on sgroup -> target:tvar:tdistr on tgroup),
```

wobei `source` den Identifikator des Quellkonstrukts der Umverteilung, `svar` den Namen der Quellvariable, `sdistr` den Quelldatenverteilungstyp und `sgroup` die der Quelldatenverteilung zugrundeliegende Prozessorgruppe angibt. Analog spezifizieren `target` den Identifikator

des Zielkonstrukts, `tvar` den Namen der Zielvariable, `tdistr` den Zieldatenverteilungstyp und `tgroup` die Prozessorgruppe der Zieldatenverteilung.

Annotation von Lastausgleichsoperationen Der semi-dynamische Compileransatz unterstützt Lastausgleichsoperationen im Schleifenrumpf von `for`- und `while`-Schleifen und in Bedingungszweigen von `if`-Konstrukten. Im erweiterten Rahmenprogramm werden `for`-, `while`- und `if`-Konstrukten, in denen zur Laufzeit ein Lastausgleich stattfinden soll, durch eine zusätzliche *Lastausgleichsannotation* gekennzeichnet.

Eine Lastausgleichsannotation besteht aus dem Schlüsselwort `rebalance` und einer optionalen *Lastausgleichsgranularität*, die die Häufigkeit des auszuführenden Lastausgleichs festlegt. Eine Lastausgleichsgranularität von `g` bewirkt einen periodischen Lastausgleich nach jeweils `g` ausgeführten Schleifeniterationen bzw. `g`-maliger Ausführung eines Bedingungszweiges der entsprechenden Schleife oder Bedingung. Wird keine Lastausgleichsgranularität spezifiziert, wird ein Lastausgleich in jeder ausgeführten Schleifeniteration bzw. bei jeder Abarbeitung eines Bedingungszweiges der entsprechend annotierten Schleife oder Bedingung durchgeführt, d.h. die Ausführung ist äquivalent zu der Angabe einer Granularität von `g=1`.

Daraus ergeben sich die folgenden drei Möglichkeiten der Spezifikation einer `for`-Schleife in einem Rahmenprogramm des semi-dynamischen Compileransatzes:

`for (i=1:n) on {...}{...}`	*Schleife ohne Lastausgleich*
`for (i=1:n) on {...} @rebalance {...}`	*Lastausgleich nach jeder Iteration*
`for (i=1:n) on {...} @rebalance (g) {...}`	*Ausgleich nach jeweils* `g` *Iterationen*

Die Annotation von Lastausgleichsoperationen an `while`-Schleifen und an `if`-Konstrukten erfolgt analog.

Mit Hilfe der Lastausgleichsannotationen und der darin enthaltenen Lastausgleichsgranularität kann die Anzahl der ausgeführten Lastausgleichsoperationen gesteuert werden. Eine derartige Operation ist stets mit zusätzlichen Bibliotheksaufrufen verbunden und kann daher bei zu häufiger Anwendung zu einem erheblichen Overhead zur Laufzeit der Anwendung führen. Auf der anderen Seite kann auch ein zu seltener Lastausgleich ungünstig sein, da eventuell vorhandene Ungleichgewichte erst spät erkannt und korrigiert werden können. Die Datenverteilungsphase des CM-task Compilers legt die zu annotierenden Konstrukte und die zugehörige Lastausgleichsgranularität mit Hilfe einer Heuristik fest, die in Abschnitt 5.3 skizziert wird. Der Anwendungsprogrammierer kann die getroffene Entscheidung durch entsprechende Anpassung des erweiterten Rahmenprogrammes verändern.

Beispiel: Iterierte Runge-Kutta Verfahren

Listing 8 zeigt die Hauptmoduldefinition des erweiterten Rahmenprogrammes für IRK Verfahren, das mit dem statischen Compileransatz erzeugt wurde. Nicht dargestellt sind die in der Analysephase erkannten Abhängigkeiten (s. Listing 6 auf Seite 56) und die deklarierten lokalen Variablen (s. Listing 4 auf Seite 50).

Das erweiterte Rahmenprogramm enthält Datenumverteilungsoperationen für die Stufenvektoren `mu` und `mu1` (Zeilen 16-21). Diese Vektoren werden in einer durch die `cparfor`-Schleife (Zeile 7) definierten Instanz des Modulaufrufs `stage_vector` (Identifikator: `n2_1`) mit dem Datenverteilungstyp `vblock` erzeugt. Die Verwendung der Stufenvektoren erfolgt durch den Modulaufruf `compute_approx` (Identifikator: `n2_2`) für `mu` bzw. durch den Modulaufruf

Listing 8: Hauptmoduldefinition des erweiterten Rahmenprogrammes für Iterierte Runge-Kutta Verfahren, wie sie durch den statischen Ansatz des CM-task Compilers für eine Plattform mit 48 Prozessoren erzeugt wird. Gegenüber dem Rahmenprogramm (s. Listing 7 auf Seite 59) sind zusätzliche Annotationen für Datenumverteilungsoperationen enthalten. Auslassungen sind durch [...] gekennzeichnet.

```
   // Hauptmodul
1  cmmain irk (y_k:vector:inout) on {1..48} {
    // Deklaration lokaler Variablen
2   [...]

    // Modulausdruck
3   seq {
4    n1: init_step (x,h,y_k,y_k1) on {1..48};
5    n2: while (x[0] < X)#100 on {1..48} {
6     seq {
7      cparfor (i = 0:s-1) {
8       n2_1: stage_vector (i,x,h,y_k,mu1[i],mu[i], ort)
9             on [{1..16},{17..32},{33..48}];
10      [...]
11     }
12     n2_2: compute_approx (h,y_k,mu) on {1..48};
13     n2_3: step_control (x,h,y_k,y_k1,mu,mu1) on {1..48};
14     [...]
       // Datenumverteilungen für die Stufenvektoren mu[i] und mu1[i]
15     redistr {
16      (n2_1[0]:mu[0]:vblock on{1..16}->n2_2:mu:sblock on{1..48}),
17      (n2_1[1]:mu[1]:vblock on{17..32}->n2_2:mu:sblock on{1..48}),
18      (n2_1[2]:mu[2]:vblock on{33..48}->n2_2:mu:sblock on{1..48}),
19      (n2_1[0]:mu1[0]:vblock on{1..16}->n2_3:mu1:sblock on{1..48}),
20      (n2_1[1]:mu1[1]:vblock on{17..32}->n2_3:mu1:sblock on{1..48}),
21      (n2_1[2]:mu1[2]:vblock on{33..48}->n2_3:mu1:sblock on{1..48})
22     }
23   } [...] } [...] } [...]
24 }
```

`step_control` (Identifikator: `n2_3`) für `mu1`. Beide Modulaufrufe werden auf allen `48` Prozessoren (Gruppe: `{1..48}`) ausgeführt und benötigen den Datenverteilungstyp `sblock`.

Zu beachten ist, dass als Quelle der Datenumverteilungsoperationen einzelne Vektoren (`mu[i]` bzw. `mu1[i]`) angegeben sind, während das Ziel der Umverteilungsoperationen die gesamte Stufenvektormatrix (`mu` bzw. `mu1`) ist. Diese Darstellung resultiert aus den Variablenzugriffen der entsprechenden Basismodulaufrufe und bewirkt, dass bei Ausführung der Datenumverteilungsoperation von der Quelle der Datenumverteilungsoperation einzelne Vektoren gesendet werden, die am Ziel der Datenumverteilungsoperation empfangen und an die korrekte Position der Matrix gespeichert werden.

Das im semi-dynamischen Compileransatz erzeugte erweiterte Rahmenprogramm enthält dieselben Datenumverteilungsoperationen und eine zusätzliche Lastausgleichsannotation für

die `while`-Schleife (Zeile 5). Dadurch wird in das generierte Koordinationsprogramm eine Lastausgleichsoperation in den Schleifenrumpf eingefügt, die zur Laufzeit einen Lastausgleich zwischen den zeitgleich ausgeführten Instanzen des Basismoduls `stage_vector` ausführt.

4.6. Koordinationsprogramm

Der letzte Transformationsschritt des CM-task Compilers ist die Codegenerierungsphase, die ein ausführbares Koordinationsprogramm in Form eines *C*-Quelltextes erzeugt. Das Koordinationsprogramm stellt eine parallele MPI Implementierung des im Spezifikationsprogramm definierten parallelen Algorithmus dar und beinhaltet alle Entscheidungen der Transformationsschritte des CM-task Compilers. Diese Entscheidungen umfassen die in der Schedulingphase ermittelten Prozessorgruppen und die Ausführungsreihenfolge der Modulaufrufe sowie die in der Datenverteilungsphase hinzugefügten Datenumverteilungs- und Lastausgleichsoperationen.

Das Koordinationsprogramm besteht aus drei Teilen:

- einer *C*-Headerdatei, die Deklarationen der verwendeten Datenstrukturen und Funktionen enthält;
- einem globalen Datenteil mit allen statisch initialisierten Daten, z.B. zur Beschreibung der Prozessorgruppen und der Datenumverteilungsoperationen, und
- einem Koordinationsteil, der eine *Koordinationsfunktion* für jedes im erweiterten Rahmenprogramm definierte Verbundmodul enthält.

Die Koordinationsfunktion eines Verbundmoduls ist eine *C*-Funktion mit einer aus der spezifizierten Verbundmodulschnittstelle generierten Parameterliste. Als letzter Parameter wird zusätzlich ein MPI Kommunikator angegeben, der zur Eingabe der ausführenden Prozessorgruppe des entsprechenden Verbundmoduls verwendet wird. Kernbestandteil des Rumpfes der Koordinationsfunktion ist der *Modulausführungsteil* , der die Übersetzung des Modulausdruckes des entsprechenden Verbundmoduls darstellt. Die konkrete Struktur des Modulausführungsteils hängt vom gewählten Compileransatz ab und wird in den Teilabschnitten 4.6.1 und 4.6.2 für den statischen bzw. den semi-dynamischen Compileransatz beschrieben. Im Folgenden wird zunächst die vom Compileransatz unabhängige Realisierung von Basis- und Verbundmodulaufrufen im Modulausführungsteil betrachtet.

Ein Basis- oder Verbundmodulaufruf des Spezifikationsprogrammes wird in der Koordinationsfunktion durch einen Funktionsaufruf der entsprechenden *C*-Funktion repräsentiert. Die Koordinationsfunktion stellt diesem Funktionsaufruf zwei Arten von Kommunikatoren zur Verfügung: den Gruppenkommunikator und externe Kommunikatoren. Der *Gruppenkommunikator* beschreibt die ausführende Prozessorgruppe des aufgerufenen Basis- oder Verbundmoduls und wird dem Funktionsaufruf zusätzlich zu den im Spezifikationsprogramm definierten aktuellen Parametern als letzter Parameter übergeben.

Externe Kommunikatoren werden zur Kommunikation zwischen zeitgleich ausgeführten Basismodulen benötigt. Die Koordinationsfunktion erzeugt für jede im erweiterten Rahmenprogramm definierte Kommunikationsabhängigkeit einen externen Kommunikator, der die ausführenden Prozessoren aller an der entsprechenden Kommunikationsabhängigkeit beteiligter Basismodule enthält. Die Bereitstellung eines externen Kommunikators erfolgt innerhalb einer *Kommunikationsstruktur* zusammen mit zusätzlichen Informationen, die die Zuordnung der Prozessoren innerhalb des Kommunikators zu den an der Kommunikationsabhängigkeit beteiligten

Basismodulen beschreiben. Diese Informationen können innerhalb der Basismodule verwendet werden, bspw. um die Prozessoranzahl der zeitgleich ausgeführten Module zu ermitteln oder um Daten zu einem spezifischen Prozessor eines bestimmten zeitgleich ausgeführten Basismoduls zu senden.

Die folgende *C*-Datentypdefinition beschreibt den konkreten Aufbau einer Kommunikationsstruktur:

```
typedef struct cmc_comm_desc {
   MPI_Comm ecomm;     // externer Kommunikator
   int numgroups;      // Anzahl beteiligte Module
   int *gsizes;        // Größen der beteiligten Prozessorgruppen
   int *goffsets;      // Positionen der Prozessorgruppen
   int groupid;        // lokale Gruppennummer
   void *data;         // Verweis auf Kommunikationsparameter
} CMC_COMM_DESC;
```

Der externe Kommunikator `ecomm` fasst die Prozessorgruppen aller an einer Kommunikationsabhängigkeit beteiligten Basismodule zusammen. Die Zuordnung der Prozesse des Kommunikators zu den einzelnen Basismodulen wird durch die Felder `gsizes` und `goffsets` beschrieben, die für jedes beteiligte Basismodul die Anzahl der ausführenden Prozessoren bzw. die Position des ersten Prozessors im Kommunikator angeben. Die lokale Gruppennummer `groupid` gibt an, welches der beteiligten Basismodule lokal ausgeführt wird, und ist vom konkreten Prozessor abhängig. Der Speicherbereich `data` verweist auf den der Kommunikationsabhängigkeit zugrundeliegenden Kommunikationsparameter. Die Größe dieses Speicherbereiches wird durch den im Spezifikationsprogramm definierten Datentyp des Kommunikationsparameters bestimmt.

4.6.1. Koordination im statischen Compileransatz

Das Rahmenprogramm des statischen Compileransatzes definiert sowohl eine feste Ausführungsreihenfolge als auch feste Prozessorgruppen für die auszuführenden Modulaufrufe. In einer statischen Koordinationsfunktion werden daher alle benötigten MPI Kommunikatoren bei Funktionseintritt angefordert und erst bei Verlassen der Funktion wieder freigegeben. Diese Herangehensweise bietet zwei Vorteile. Erstens können identische Prozessorgruppen, die bspw. für verschiedene Modulaufrufe benötigt werden, auf denselben MPI Kommunikator abgebildet werden. Dadurch wird der Ressourcenverbrauch der Koordinationsfunktion reduziert. Der zweite Vorteil liegt darin, dass jeder MPI Kommunikator nur einmal pro Ausführung des Verbundmoduls angefordert werden muss, wodurch die Anzahl der ausgeführten MPI Operationen und damit der Koordinationsoverhead minimiert wird.

Der Modulausführungsteil des statischen Compileransatzes ist analog zur Struktur des Modulausdrucks des Rahmenprogrammes hierarchisch aufgebaut und besteht aus den folgenden Codefragementen für die im Modulausdruck enthaltenen Basis- und Verbundmodulaufrufe und `for`-, `while`- und `if`-Konstrukte:

1. **Prozessorauswahl:** Der für die Prozessorauswahl eingefügte Programmcode stellt sicher, dass die nachfolgend aufgeführten Schritte nur von Prozessoren abgearbeitet werden, die

in der im Rahmenprogramm annotierten Prozessorgruppe des jeweiligen Konstrukts, also des Modulaufrufs, der Schleife oder der Bedingung, enthalten sind.

2. **Empfangsoperationen:** Für jede im erweiterten Rahmenprogramm definierte Datenumverteilungsoperation, deren Zielkonstrukt mit dem jeweiligen Basis- oder Verbundmodulaufruf bzw. der jeweiligen Schleife oder Bedingung übereinstimmt, wird eine entsprechende Empfangsoperation eingefügt. Die Empfangsoperation wird durch die Datenumverteilungsbibliothek bereitgestellt (s. Abschnitt 4.7) und ist blockierend realisiert, so dass die nachfolgenden Schritte erst nach Empfang aller benötigter Eingabedaten des jeweiligen Konstrukts ausgeführt werden.

3. **Bereitstellung der Kommunikationsstrukturen:** Die zur Kommunikation zwischen zeitgleich ausgeführten Basismodulen benötigten Kommunikationsstrukturen werden im statischen Compileransatz im globalen Datenteil des Koordinationsprogrammes abgelegt. An dieser Stelle werden die zur Laufzeit ermittelten Informationen, d.h. der externe Kommunikator (Strukturelement `ecomm`), der Verweis auf den Speicherplatz des Kommunikationsparameters (Strukturelement `data`) und die lokale Gruppennummer (Strukturelement `groupid`) ergänzt. Die restlichen Strukturelemente (`numgroups`, `gsizes` und `goffsets`) sind zur Compilezeit bekannt und werden daher statisch initialisiert.

4. **Ausführung:** An dieser Stelle wird für Basis- und Verbundmodulaufrufe ein Funktionsaufruf der entsprechenden *C*-Funtion eingefügt. Für Schleifen und Bedingungen wird ein Codefragment für den Schleifenrumpf bzw. die Bedingungszweige erzeugt und in ein entsprechendes Schleifen- bzw. Bedingungskonstrukt in *C* eingebettet.

5. **Sendeoperationen:** An dieser Stelle wird für jede im erweiterten Rahmenprogramm definierte Datenumverteilungsoperation, deren Quellkonstrukt mit dem jeweiligen Basis- oder Verbundmodulaufruf bzw. der jeweiligen Schleife oder Bedingung übereinstimmt, eine entsprechende Sendeoperation eingefügt. Sendeoperationen stellen die Eingabedaten für nachfolgend ausgeführte Berechnungen zur Verfügung und sind in der Datenumverteilungsbibliothek nichtblockierend realisiert. Dadurch können alle Sendeoperationen abgearbeitet werden, auch wenn die korrespondierenden Empfangsoperationen erst zu einem späteren Zeitpunkt ausgeführt werden.

Die restlichen Konstruktoren des Rahmenprogrammes (`par`, `cpar`, `parfor`, `cparfor` und `seq`) erscheinen nicht explizit im erzeugten Koordinationsprogramm. Diese Konstruktoren definieren im Rahmenprogramm die Ausführungsreihenfolge der Basis- und Verbundmodulaufrufe, Schleifen und Bedingungen und sind im Koordinationsprogramm implizit durch die Reihenfolge der eingefügten Codefragmente und der in der Prozessorauswahl (Punkt 1) verwendeten Prozessorgruppen vertreten.

Beispiel: Iterierte Runge-Kutta Verfahren

Die erzeugte Koordinationsfunktion für das Hauptmodul des Spezifikationsprogrammes für IRK Verfahren ist in Listing 9 dargestellt. Die generierte Funktionsschnittstelle (Zeile 1) besteht aus dem spezifizierten Ein- und Ausgabeparameter `y_k` und dem MPI Kommunikator `CMC_rootcomm`, der die ausführende Prozessorgruppe des Hauptmoduls beschreibt. Der

Listing 9: Statische Koordinationsfunktion des Hauptmoduls für Iterierte Runge-Kutta Verfahren. Auslassungen sind durch [...] gekennzeichnet.

```
1  void irk (double *y_k, MPI_Comm CMC_rootcomm) {
     // Deklaration der spezifizierten lokalen Variablen
2    double *x, *h, *mu, *mu1, *ort, *y_k1;
     // Deklaration der Koordinationsvariablen
3    int CMC_nump;
4    MPI_Group CMC_rootgroup, CMC_groups[4];
5    MPI_Comm CMC_comms[4];
     // Überprüfung der benötigten Prozessoranzahl
6    MPI_Comm_size (CMC_rootcomm, &CMC_nump);
7    if (CMC_nump < 48) {
8      fprintf (stderr, "Modul irk kann nicht ausgeführt werden.\n"
         "Benötigte Prozessoren: 48. Verfügbare Prozessoren: %d\n",
         CMC_nump);
9      return;
10   }
     // Speicheranforderung für spezifizierte lokalen Variablen
11   x = (double *) calloc (1, sizeof (double));
12   [...]
     // Initialisierung der MPI Kommunikatoren
13   MPI_Comm_group (CMC_rootcomm, &CMC_rootgroup);
14   MPI_Group_range_incl (CMC_rootgroup, 1, CMC_DAT_irk_group_0,
       &CMC_groups[0]);
15   MPI_Comm_create (CMC_rootcomm, CMC_groups[0], &CMC_comms[0]);
16   [...]
     // Modulausführungsteil (Zeilen 17–38) s. Listing 10
     // Freigabe der MPI Kommunikatoren
39   MPI_Group_free (&CMC_groups[0]);
40   MPI_Comm_free (&CMC_comms[0]);
41   [...]
     // Speicherfreigabe für spezifizierte lokale Variablen
42   free (x);
43   [...]
44 }
```

Präfix `CMC` zeigt an, dass es sich um eine vom CM-task Compiler zur Koordination verwendete Variable handelt.

Zeile 2 enthält die Deklaration der lokalen Variablen, die vom Anwendungsentwickler im Spezifikationsprogramm definiert wurden. Die zur Koordination benötigten lokalen Variablen werden in den Zeilen 3-5 deklariert. Diese *Koordinationsvariablen* beinhalten die Variable `CMC_nump` zur Speicherung der Prozessoranzahl im eingegebenen MPI Kommunikator `CMC_rootcomm` und vier MPI Kommunikatoren, die in der Variable `CMC_comms` gespeichert werden. Diese vier Kommunikatoren ergeben sich aus den vier unterschiedlichen Prozessorgruppenannotationen, die im Rahmenprogramm (s. Listing 7 auf Seite 59) inner-

halb der Hauptmoduldefinition auftreten. Der Kommunikator `CMC_comms[0]` repräsentiert die Prozessorgruppenannotation {`1..48`} und wird bspw. zur Ausführung des Basismoduls `init_step` benötigt. Die drei anderen Kommunikatoren repräsentieren die am Basismodulaufruf `stage_vector` annotierten Prozessorgruppen ({`1..16`}, {`17..32`} und {`33..48`}). Die deklarierten MPI Gruppen (`CMC_rootgroup` und `CMC_groups`) sind Hilfsvariablen, die zur Erzeugung der Kommunikatoren verwendet werden.

In den Zeilen 6-10 erfolgt die Überprüfung, ob der übergebene Kommunikator `CMC_root-comm` genügend Prozessoren zur Ausführung des Hauptmoduls enthält. Entsprechend der Prozessorgruppenannotation der Verbundmoduldefinition im Rahmenprogramm werden 48 Prozessoren benötigt. Stehen weniger Prozessoren zur Verfügung, wird die Ausführung des Hauptmoduls mit einer entsprechenden Fehlermeldung abgebrochen.

Zeile 11 zeigt die Initialisierung der vom Anwendungsprogrammierer spezifizierten lokalen Variablen am Beispiel der Variable `x`. Für Feldvariablen wird an dieser Stelle mit Hilfe der *C*-Funktion `calloc` Speicherplatz angefordert. Dieser Speicherplatz wird am Ende der Koordinationsfunktion wieder freigegeben (s. Zeile 42).

Der erzeugte Programmcode zur Initialisierung der vier Kommunikatoren ist in den Zeilen 13 bis 16 abgebildet. Zur Illustration der Vorgehensweise ist die Anforderung des Kommunikators `CMC_comms[0]` angegeben. Die Anforderung eines Kommunikators erfolgt in zwei Schritten. Zunächst wird durch die MPI Funktion `MPI_Group_range_incl` die MPI Gruppe `CMC_groups[0]` erzeugt, aus der anschließend durch `MPI_Comm_create` der entsprechende MPI Kommunikator erzeugt wird. Der Parameter `CMC_DAT_irk_group_0` definiert die Prozessoren der zu erzeugenden Gruppe und wird durch den CM-task Compiler aus der Prozessorgruppenannotation generiert und im globalen Datenteil des Koordinationsprogrammes gespeichert. Die Freigabe der erzeugten MPI Kommunikatoren und MPI Gruppen erfolgt am Ende der Koordinationsfunktion (Zeilen 39- 41).

Der Modulausführungsteil der Koordinationsfunktion (Zeilen 17- 38) ist separat in Listing 10 abgebildet. Entsprechend der durch das Rahmenprogramm definierten Ausführungsreihenfolge (s. Abbildung 16 auf Seite 60) erfolgt zunächst der Aufruf des Basismoduls `init_step`. Da dieses Basismodul weder an Datenumverteilungsoperationen noch an externen Kommunikationsoperationen beteiligt ist, wird nur Code für die Prozessorauswahl und die Ausführung des Modulaufrufs mit den spezifizierten Parametern erzeugt. Die Prozessorauswahl (Zeile 17) erfolgt durch den MPI Kommunikator `CMC_comms[0]`, der die im Rahmenprogramm annotierte Prozessorgruppe {`1..48`} repräsentiert.

Analoger Programmcode wird für die Prozessorauswahl der `while`-Schleife (Zeile 20) erzeugt. Der Programmcode zur Ausführung der `while`-Schleife besteht aus dem Schleifenkopf (Zeile 21), der direkt aus der Übersetzung der Spezifikation entsteht, und dem Schleifenrumpf, für den exemplarisch der erzeugte Code des Modulaufrufs `stage_vector[0]` angegeben ist. Dieser Modulaufruf erfolgt auf der Prozessorgruppe {`1..16`}, die im Koordinationsprogramm durch den MPI Kommunikator `CMC_comms[1]` repräsentiert wird.

Die durch den CM-task Compiler erzeugte Kommunikationsstruktur `CMC_DAT_irk_comm1` repräsentiert die Kommunikationsabhängigkeit zwischen den `s=3` Instanzen des Modulaufrufs `stage_vector` und ist im globalen Datenteil abgelegt. In den Zeilen 23 bis 25 werden der zur Laufzeit erzeugte externe Kommunikator (Strukturelement `ecomm`), die lokale Gruppenposition (Strukturelement `groupid`) und der Verweis auf den Kommunikationsparameter `ort` (Strukturelement `data`) ergänzt. Der externe Kommunikator beinhaltet die ausführenden Prozessoren aller drei Instanzen des Modulaufrufs und entspricht daher dem Kommunikator

Listing 10: Modulausführungsteil der statischen Koordinationsfunktion für Iterierte Runge-Kutta Verfahren. Auslassungen sind durch [...] gekennzeichnet.

```
    // Modulaufruf init_step auf Prozessorgruppe {1..48}
17  if (CMC_comms[0] != MPI_COMM_NULL) {          // Prozessorauswahl
18    init_step(x, h, y_k, y_k1, comms[0]);             // Ausführung
19  }
    // while-Schleife auf Prozessorgruppe {1..48}
20  if (CMC_comms[0] != MPI_COMM_NULL) {          // Prozessorauswahl
21    while (x[0]<X) { // Ausführung while-Schleife
        // Modulaufruf stage_vector[0] auf Prozessorgruppe {1..16}
22      if (CMC_comms[1] != MPI_COMM_NULL) {      // Prozessorauswahl
          // Bereitstellung der Kommunikationsstrukturen
23        CMC_DAT_irk_comm1.ecomm = CMC_comms[0];
24        CMC_DAT_irk_comm1.groupid = 0;
25        CMC_DAT_irk_comm1.data = ort;
          // Ausführung des Modulaufrufs
26        stage_vector (0, x, h, y_k, mu1+0*1000, mu+0*1000,
            CMC_DAT_irk_comm1, CMC_comms[1]);
          // Sendeoperationen
27        CMC_Red_send_static (mu, ..., 20000);
28        [...]
29      }
30      [...]
        // Modulaufruf compute_approx auf Prozessorgruppe {1..48}
31      if (CMC_comms[0] != MPI_COMM_NULL) {      // Prozessorauswahl
          // Empfangsoperationen
32        CMC_Red_recv_static (mu, ..., 20000);
33        [...]
34        compute_approx (h, y_k, mu, CMC_comms[0]);  // Ausführung
35      }
36      [...]
37    }
38  }
```

`CMC_comms[0]`.

Zeile 27 zeigt exemplarisch eine Sendeoperation für die durch den Modulaufruf geschriebene Variable `mu`. Diese Operation wird durch einen Aufruf der entsprechenden Funktion der Datenumverteilungsbibliothek (s. Abschnitt 4.7) realisiert. Der Parameter `20000` ist ein vom CM-task Compiler erzeugter, eindeutiger Identifikator (Tag) der Datenumverteilungsoperation, der auch von der korrespondierenden Empfangsoperation in Zeile 32 verwendet wird.

4.6.2. Koordination im semi-dynamischen Compileransatz

Im semi-dynamischen Compileransatz definiert das Rahmenprogramm eine feste Ausführungsreihenfolge, wohingegen die annotierten Prozessorgruppen nur die initiale Belegung vorgeben und zur Laufzeit der Anwendung verändert werden können. Daher enthalten semi-dynamische

Koordinationsprogramme wie ihre statischen Gegenstücke eine fest kodierte Ausführungsreihenfolge, verwenden aber im Unterschied zum statischen Ansatz eine flexible Darstellung der Prozessorgruppen. Die Verwaltung der Prozessorgruppen und das Erzeugen und Auflösen der benötigten MPI Kommunikatoren wird durch die Lastausgleichsbibliothek (s. Abschnitt 4.8 übernommen.

Semi-dynamische Koordinationsprogramme enthalten entsprechend den Annotationen des erweiterten Rahmenprogrammes Lastausgleichsoperationen, in denen Prozessoren zwischen den Prozessorgruppen zeitgleich auszuführender Module verschoben werden können. Die Lastausgleichsoperationen basieren auf dynamischen Leistungsdaten, die durch das Koordinationsprogramm gesammelt werden. Dazu werden im Koordinationsprogramm Modulaufrufe, Schleifen und Bedingungen in Profilingcode zur Messung der Ausführungszeit eingebettet. Für jeden Modulaufruf, jede Schleife und jede Bedingung wird die Ausführungsanzahl und die kumulierte Ausführungszeit abgespeichert. Innerhalb der Lastausgleichsoperationen können diese Zähler abgefragt und nach erfolgter Anpassung der Prozessorgruppen zurückgesetzt werden.

Der semi-dynamische Modulausführungsteil enthält Koordinationscode für Basis- und Verbundmodulaufrufe, `for`- und `while`-Schleifen und Bedingungen (`if`), der aus den folgenden Bestandteilen zusammengesetzt wird.

1. **Prozessorauswahl:** Der erzeugte Koordinationscode stellt sicher, dass die nachfolgend aufgeführten Schritte nur durch die aktuell dem jeweiligen Basis- oder Verbundmodulaufruf bzw. der jeweiligen Schleife oder Bedingung zugeordneten Prozessoren ausgeführt wird.

2. **Empfangsoperationen:** Entsprechend den im erweiterten Rahmenprogramm definierten Datenumverteilungsoperationen werden die Empfangsoperationen für die Eingabedaten des jeweiligen Basis- oder Verbundmodulaufrufs bzw. der jeweiligen Schleife oder Bedingung eingefügt. Analog zum statischen Ansatz sind diese Operationen blockierend realisiert und werden durch die Datenumverteilungsbibliothek bereitgestellt.

3. **Anforderung der Kommunikatoren:** Für Schleifen und Bedingungen wird ein Aufruf der Lastausgleichsbibliothek eingefügt, der die für die Modulaufrufe im zugehörigen Schleifenrumpf bzw. in den zugehörigen Bedingungszweigen benötigten Kommunikatoren entsprechend den aktuell zugeordneten Prozessorgruppen erzeugt. Zusätzlich initialisiert dieser Funktionsaufruf auch die im entsprechenden Schleifenrumpf bzw. die in den entsprechenden Bedingungszweigen verwendeten Kommunikationsstrukturen mit den entsprechenden externen Kommunikatoren.

4. **Bereitstellung der Kommunikationsstrukturen:** Für Modulaufrufe werden die zur Laufzeit ermittelten Strukturelemente der für den Aufruf benötigten Kommunikationsstrukturen mit den entsprechenden Werten belegt. Diese Strukturelemente sind die vom ausführenden Prozessor abhängige lokale Gruppennummer (Strukturelement `groupid`) und der Verweis auf den Speicherplatz des Kommunikationsparameters (Strukturelement `data`). Die restlichen Strukturelemente (`ecomm`, `numgroups`, `gsizes` und `goffsets`) werden bereits im vorherigen Schritt entsprechend der aktuell verwendeten Prozessorgruppen gesetzt.

5. **Start der Zeitmessung:** An dieser Stelle wird Programmcode zur Abfrage und Speicherung der aktuellen Systemzeit eingefügt.

6. **Ausführung:** Analog zum statischen Compileransatz wird an dieser Stelle ein entsprechender Funktionsaufruf (für Basis- und Verbundmodulaufrufe) oder der erzeugte Programmcode des Schleifenrumpfes bzw. der Bedingungszweige (für Schleifen und Bedingungen) eingefügt.

7. **Lastausgleich:** Für Schleifen und Bedingungen mit einer Lastausgleichsannotation im erweiterten Rahmenprogramm wird als letzte Anweisung des zugehörigen Schleifenrumpfes bzw. der zugehörigen Bedingungszweige ein entsprechender Aufruf der Lastausgleichsbibliothek eingefügt.

8. **Ende der Zeitmessung:** An dieser Stelle wird Programmcode zur Berechnung und Speicherung der Ausführungszeit des Basis- oder Verbundmoduls bzw. der kompletten Schleife oder Bedingung eingefügt.

9. **Kommunikatorfreigabe:** Für Schleifen und Bedingungen wird ein Aufruf der Lastausgleichsbibliothek eingefügt, der die in Punkt 3 angeforderten Kommunikatoren des zugehörigen Schleifenrumpfes bzw. der zugehörigen Bedingungszweige wieder freigibt.

10. **Sendeoperationen:** Entsprechend der im erweiterten Rahmenprogramm definierten Datenumverteilungsoperationen werden Sendeoperationen eingefügt. Diese Operationen sind analog zum statischen Ansatz nichtblockierend realisiert und werden durch die Datenumverteilungsbibliothek bereitgestellt.

Die restlichen Konstrukte des Rahmenprogrammes (`seq`, `par`, `cpar`, `parfor` und `cparfor`) erscheinen nicht explizit in der erzeugten Koordinationsfunktion. Im semi-dynamischen Compileransatz werden diese Konstrukte in eine Datenstruktur übersetzt, die die durch die Konstrukte definierte Ausführungsreihenfolge speichert und als Schnittstelle zwischen semi-dynamischem Koordinationsprogramm und Lastausgleichsbibliothek genutzt wird. Diese Datenstruktur wird auch als Schedulestruktur bezeichnet und in Abschnitt 5.4 näher vorgestellt.

4.7. Datenumverteilungsbibliothek

Datenumverteilungsoperationen werden in den vom CM-task Compiler erzeugten Koordinationsprogrammen benötigt, um die Art der Datenverteilung oder die zur Speicherung verwendete Prozessorgruppe von Variablen zu verändern, bspw. um die aktuellen Parameter eines Modulaufrufs in der vom aufgerufenen Modul geforderten Datenverteilung auf den das jeweilige Modul ausführenden Prozessoren zur Verfügung zu stellen. Die Realisierung dieser Datenumverteilungsoperationen erfolgt durch die Datenumverteilungsbibliothek des CM-task Compilerframeworks.

Eine Datenumverteilungsoperation gliedert sich in eine *Sendephase* und eine *Empfangsphase*. Die Sendephase wird von der Quellprozessorgruppe der Datenumverteilungsoperation abgearbeitet, d.h. von denjenigen Prozessoren, die die umzuverteilende Variable vor der Datenumverteilung speichern. Die Empfangsphase wird dagegen von der Zielprozessorgruppe der Datenumverteilungsoperation ausgeführt, d.h. von denjenigen Prozessoren, auf denen die Variable nach der Datenumverteilungsoperation gespeichert wird. Die Datenumverteilungsbibliothek

des CM-task Compilerframeworks unterstützt beliebige Quell- und Zielprozessorgruppen, d.h. die an einer Datenumverteilungsoperation beteiligten Gruppen müssen nicht identisch sein.

Die in einer Datenumverteilungsoperation ausgeführten Kommunikationsoperationen hängen neben der Quell- und der Zielprozessorgruppe auch vom Datenverteilungstyp der umzuverteilenden Variable vor und nach der Umverteilungsoperation ab. Im Allgemeinen müssen Daten von jedem Quellprozessor zu jedem Zielprozessor übertragen werden, wobei die erforderlichen Datentransfers durch nichtblockierende Punkt-zu-Punkt Kommunikationsoperationen der MPI Bibliothek realisiert sind.

Die Sendephase einer Datenumverteilungsoperation gliedert sich in die folgenden drei Arbeitsschritte:

(a) jeder Quellprozessor bestimmt für die lokal gespeicherten Elemente der umzuverteilenden Datenstruktur die entsprechenden Zielprozessoren und berechnet für jeden Zielprozessor, wieviele Datenelemente zu diesem Prozessor gesendet werden müssen;

(b) jeder Quellprozessor kopiert die lokal gespeicherten Datenelemente in einen Sendepuffer, so dass Elemente mit demselben Zielprozessor in aufeinanderfolgenden Positionen des Sendepuffers abgelegt werden;

(c) jeder Quellprozessor sendet die entsprechenden Abschnitte des Sendepuffers an den jeweiligen Zielprozessor. Durch die Anordnung der Elemente im Sendepuffer wird zu jedem Zielprozessor maximal eine Nachricht gesendet.

Abbildung 17 (links) zeigt die Kommunikationsoperationen zur Umverteilung eines zwölfelementigen Feldes von einer zyklischen Datenverteilung auf drei Quellprozessoren zu einer blockzyklischen Datenverteilung mit Blockgröße zwei auf vier Zielprozessoren. Rechts in dieser Abbildung sind die in der Sendephase ausgeführten Aktionen am Beispiel des Quellprozessors P_1 dargestellt.

Die Empfangsphase einer Datenumverteilungsoperation besteht aus den folgenden drei Arbeitsschritten:

(d) jeder Zielprozessor bestimmt für die lokal zu speichernden Elemente der umzuverteilenden Datenstruktur die entsprechenden Quellprozessoren und berechnet, wieviele Datenelemente von jedem Quellprozessor empfangen werden;

(e) jeder Zielprozessor empfängt die in Schritt (c) gesendeten Nachrichten von den jeweiligen Quellprozessoren und legt die empfangenen Daten geordnet nach Quellprozessor im Empfangspuffer ab;

(f) jeder Zielprozessor kopiert die Datenelemente aus dem Empfangspuffer an die korrekte Position innerhalb der umzuverteilenden Datenstruktur.

Abbildung 17 (rechts unten) illustriert die ausgeführten Aktionen der Empfangsphase am Beispiel des Zielprozessors Q_1.

Die Datenumverteilungsbibliothek des CM-task Compilerframeworks enthält zwei Arten von Umverteilungsoperationen für mehrdimensionale Felder: statische Datenumverteilungsoperationen und dynamische Datenumverteilungsoperationen. Im Folgenden wird ein Überblick dieser beiden Operationsarten gegeben.

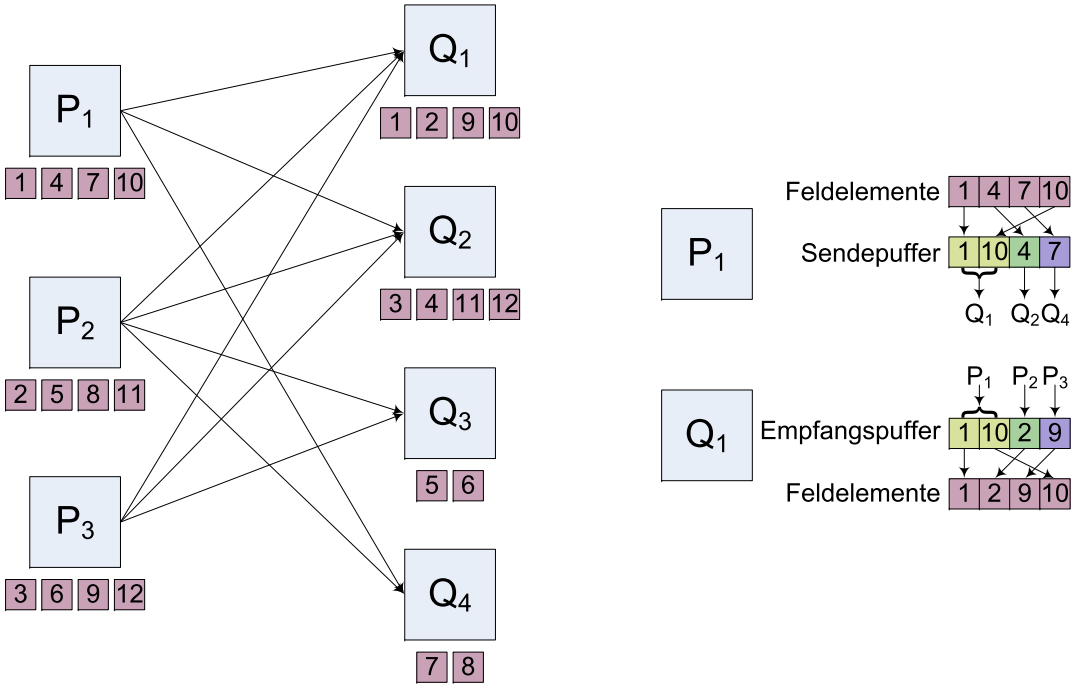


Abbildung 17: Beispiel für die Umverteilung eines eindimensionalen Feldes mit zwölf Elementen $\{1,\ldots,12\}$ von einer zyklischen Datenverteilung auf drei Quellprozessoren $\{P_1,P_2,P_3\}$ zu einer blockzyklischen Datenverteilung mit Blockgröße zwei auf vier Zielprozessoren $\{Q_1,\ldots,Q_4\}$. Links: Illustration der auszuführenden Kommunikationsoperationen zwischen den Quell- und Zielprozessoren. Rechts oben: Illustration der vom Quellprozessor P_1 auszuführenden Kopier- und Kommunikationsoperationen zum Aufbau des Sendepuffers bzw. zum Transfer der Pufferelemente zu den jeweiligen Zielprozessoren. Recht unten: Illustration der auszuführenden Empfangs- und Kopieroperationen für den Zielprozessor Q_1.

Statische Datenumverteilungsoperationen Die statischen Datenumverteilungsoperationen zielen auf eine Minimierung des entstehenden Datenumverteilungsoverheads zur Laufzeit ab. Dazu werden die Schritte (a) und (d) zur Berechnung der von den Quell- und Zielprozessoren auszuführenden Kopier- und Kommunikationsoperationen bereits zur Compilezeit ausgeführt. Diese Berechnung erfolgt in der Codegenerierungsphase des CM-task Compilers basierend auf den vom Anwendungsentwickler spezifizierten parametrisierten Datenverteilungsvektoren der Quell- und der Zieldatenverteilung. Da für diese Berechnung auch die an der Datenumverteilung beteiligten Prozessorgruppen bekannt sein müssen, können die statischen Datenumverteilungsoperationen nur im statischen Ansatz des CM-task Compilers verwendet werden.

Das Ergebnis der Berechnungen im CM-task Compiler wird in Form einer *Umverteilungsstruktur* für jeden Quell- und für jeden Zielprozessor in das erzeugte Koordinationsprogramm integriert. Eine Umverteilungsstruktur ist eine Datenstruktur, die für einen bestimmten Prozessor sowohl die auszuführenden lokalen Kopieroperationen als auch die auszuführenden Sendeoperationen (für Quellprozessoren einer Datenumverteilungsoperation) bzw. Empfangsoperationen (für Zielprozessoren einer Datenumverteilungsoperation) speichert.

```
// Datentyp und Datenverteilungstypspezifikation
type matrix = array [100][100] of double;
distrib matrix:rowblock = [block on p][block on 1];

// erzeugte Kodierungsmatrizen für P=10 Prozessoren
```

$$MSM^T = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$BSM^T = \begin{pmatrix} 100 & 50 & 34 & 25 & 20 & 17 & 15 & 13 & 12 & 10 \\ 100 & 100 & 100 & 100 & 100 & 100 & 100 & 100 & 100 & 100 \end{pmatrix}$$

Abbildung 18: Beispiel für die Spezifikation eines Datenverteilungstyps für ein zweidimensionales Feld `matrix` (oben) und Kodierung des Datenverteilungstyps durch zwei Matrizen MSM und BSM für $P = 10$ Prozessoren (unten).

Die Realisierung der statischen Datenumverteilungsoperationen erfolgt durch die Funktionen `CMC_Red_send_static` (Sendephase) und `CMC_Red_recv_static` (Empfangsphase) der Datenumverteilungsbibliothek. Eine Schnittstellenbeschreibung dieser Funktionen und der konkrete Aufbau der Umverteilungsstruktur ist im Anhang Abschnitt C.2.1 angegeben.

Dynamische Datenumverteilungsoperationen Die dynamischen Datenumverteilungsoperationen unterstützen flexible Quell- und Zielprozessorgruppen, die erst zur Laufzeit der Anwendung vor Abarbeitung der jeweiligen Umverteilungsoperation feststehen müssen. Damit sind die dynamischen Datenumverteilungsoperationen speziell für den semi-dynamischen Ansatz des CM-task Compilers geeignet.

Die Berechnung der auszuführenden Kopier- und Kommunikationsoperationen (Schritte (a) und (d)) erfolgt in den dynamischen Datenumverteilungsoperationen zur Laufzeit. Analog zu den statischen Umverteilungsoperationen basiert diese Berechnung auf den parametrisierten Datenverteilungsvektoren von Quell- und Zieldatenverteilung, die in kodierter Form in das Koordinationsprogramm eingebunden und den Umverteilungsoperationen zur Verfügung gestellt werden.

Die Kodierung eines parametrisierten Datenverteilungsvektors für ein d-dimensionales Feld erfolgt mit Hilfe von zwei $P \times d$ Matrizen MSM und BSM, wobei P die Anzahl der verfügbaren Prozessoren der Zielplattform angibt. Die Matrix $MSM = (msm_{pi})$ definiert das der Datenverteilung zugrundeliegende virtuelle d-dimensionale Prozessorgitter, d.h. Matrixelement msm_{pi} beschreibt die Anzahl der Prozessoren in Felddimension $i, i = 1, \ldots, d$, für $p, p = 1, \ldots, P$, zur Speicherung des entsprechenden Feldes verwendete Prozessoren. Die Matrix $BSM = (bsm_{pi})$ beschreibt die durch den parametrisierten Datenverteilungsvektor definierten Blockgrößen, d.h. Matrixelement bsm_{pi} definiert die Blockgröße in Felddimension $i, i = 1, \ldots, d$, für $p, p = 1, \ldots, P$, zur Speicherung des entsprechenden Feldes verwendete Prozessoren. Abbildung 18 zeigt ein Beispiel für die Kodierung eines definierten Datenverteilungstyps.

Die Realisierung der dynamischen Datenumverteilungsoperationen erfolgt durch die Funktionen `CMC_Red_send_dynamic` (Sendephase) und `CMC_Red_recv_dynamic` (Empfangs-

phase) der Datenumverteilungsbibliothek. Eine Schnittstellenbeschreibung dieser Funktionen ist im Anhang Abschnitt C.2.2 angegeben.

Datenumverteilungsoperationen für Nutzerdatentypen Zusätzlich definiert die Datenumverteilungsbibliothek eine Schnittstelle zur Einbindung von Datenumverteilungsoperationen für Nutzerdatentypen. Diese Schnittstelle ist vom Compileransatz unabhängig und besteht aus den Funktionen `CMC_Red_send_user` (Sendephase) und `CMC_Red_recv_user` (Empfangsphase). Diesen Funktionen wird zur Laufzeit der Anwendung der vom Anwendungsentwickler spezifizierte Datentypidentifikator (s. Abschnitt 4.1.1) der umzuverteilenden Datenstruktur, sowie die vom Anwendungsentwickler spezifizierten Identifikatoren für Quell- und Zieldatenverteilung (s. Abschnitt 4.1.2) übergeben. Die genaue Schnittstellenbeschreibung befindet sich im Anhang in Abschnitt C.2.3.

4.8. Lastausgleichsbibliothek

Die Lastausgleichsbibliothek stellt Funktionen und Datenstrukturen zur Unterstützung der Abarbeitung semi-dynamischer Koordinationsprogramme bereit. Zentrale Schnittstelle zwischen Koordinationsprogramm und Lastausgleichsbibliothek ist die *Schedulestruktur*. Diese Datenstruktur beschreibt den Schedule eines Programmblockes, d.h. eines Schleifenrumpfes einer `for`- oder `while`-Schleife, eines Bedingungszweiges oder eines gesamten Verbundmoduls.

Konkret beschreibt eine Schedulestruktur, welche Module des zugehörigen Programmblockes zeitgleich ausgeführt werden und damit für eine Lastausgleichsoperation in Frage kommen sowie welche Kommunikationsabhängigkeiten zwischen zeitgleich ausgeführten Basismodulaufrufen bestehen. Ein Modul kann dabei ein ausgeführtes Basis- oder Verbundmodul, eine komplette `for`- oder `while`-Schleife oder eine komplette Bedingung (`if`-Konstrukt) sein. Für jedes Modul speichert die Schedulestruktur die aktuell zugewiesene Prozessorgruppe mit dem entsprechenden MPI Kommunikator und die vom Koordinationsprogramm ermittelten Leistungsdaten, d.h. die Ausführungsanzahl und die kumulierte Ausführungszeit. Der konkrete Aufbau einer Schedulestruktur und die Erzeugung durch den CM-task Compiler werden in Abschnitt 5.4.2 beschrieben.

Die Funktionen der Lastausgleichsbibliothek sind in Abbildung 19 dargestellt. Die Allokationsfunktion `CMC_Lb_allocate` fordert die benötigten MPI Kommunikatoren eines Programmblockes an und speichert sie in der entsprechenden Schedulestruktur. Die angeforderten Kommunikatoren umfassen sowohl die Gruppenkommunikatoren zur Ausführung von Modulen als auch externe Kommunikatoren zur Kommunikation zwischen zeitgleich ausgeführten Basismodulen. In einem semi-dynamischen Koordinationsprogramm wird die Allokationsfunktion bei Eintritt in ein Verbundmodul oder vor der Ausführung einer Schleife oder Bedingung mit der entsprechenden Schedulestruktur aufgerufen.

Die Deallokationsfunktion `CMC_Lb_deallocate` gibt die in einer Schedulestruktur gespeicherten Kommunikatoren wieder frei. Diese Funktion wird in semi-dynamischen Koordinationsprogramm vor Verlassen eines Verbundmoduls bzw. nach kompletter Abarbeitung einer Schleife oder Bedingung ausgeführt.

Durch Lastausgleichsoperationen können in einem semi-dynamischen Koordinationsprogramm Prozessoren zwischen den Prozessorgruppen zeitgleich ausgeführter Module verschoben werden. Lastausgleichsoperationen werden am Ende eines Schleifenrumpfes oder eines

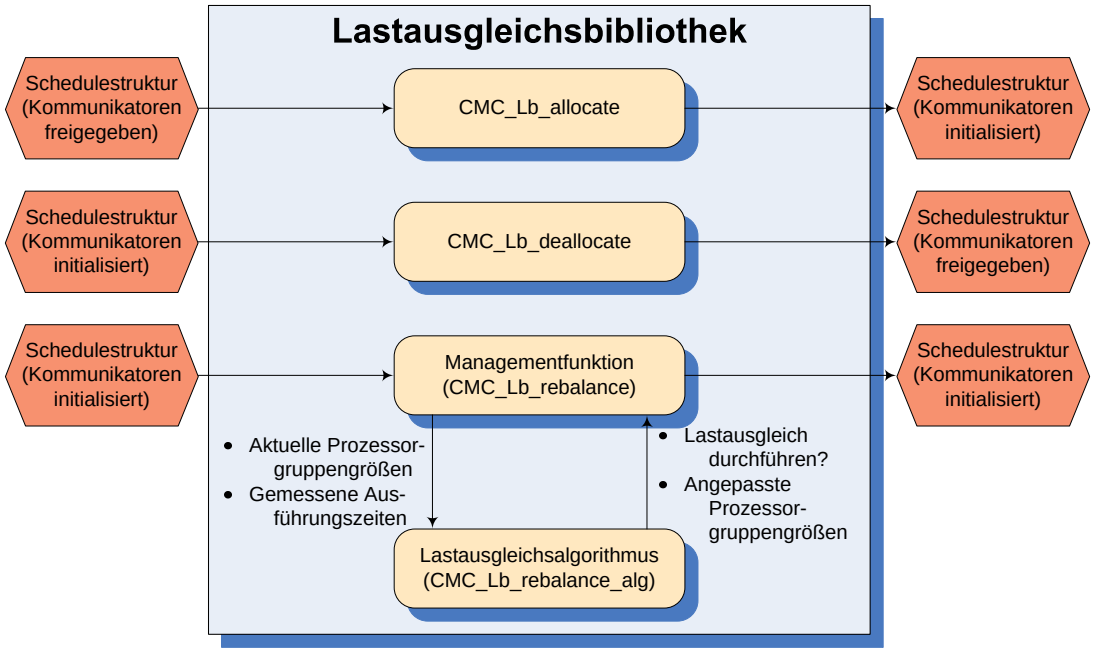


Abbildung 19: Illustration der Struktur der Lastausgleichsbibliothek. Die Funktionen `CMC_Lb_allocate` und `CMC_Lb_deallocate` fordern die benötigten Kommunikatoren einer Schedulestruktur an bzw. geben diese wieder frei. Die Managementfunktion `CMC_Lb_rebalance` extrahiert die aktuellen Prozessorgruppengrößen und die gemessenen Ausführungszeiten einer Menge zeitgleich ausgeführter Module aus der Schedulestruktur und stellt sie dem Lastausgleichsalgorithmus zur Verfügung. Dieser Algorithmus entscheidet, ob ein Lastausgleich notwendig ist und gibt gegebenenfalls angepasste Prozessorgruppengrößen aus. Falls ein Lastausgleich stattfinden soll, passt die Managementfunktion die Schedulestruktur entsprechend an und erzeugt neue Kommunikatoren basierend auf den angepassten Prozessorgruppengrößen.

Bedingungszweiges durch alle die entsprechende Schleife bzw. Bedingung abarbeitenden Prozessoren gemeinsam ausgeführt. Die Realisierung einer derartigen Operation erfolgt in der Lastausgleichsbibliothek durch zwei Funktionen: der Managementfunktion und dem Lastausgleichsalgorithmus.

Der *Lastausgleichsalgorithmus* (Bibliotheksfunktion `CMC_Lb_rebalance_alg`) entscheidet für eine Menge zeitgleich ausgeführter Module basierend auf den gemessenen Ausführungszeiten und den verwendeten Prozessorgruppengrößen, ob ein Lastausgleich stattfinden soll und gibt gegebenenfalls angepasste Prozessorgruppengrößen aus. Die Realisierung des Lastausgleichsalgorithmus ist unabhängig vom Aufbau der Schedulestruktur, wodurch eine einfache Erweiterbarkeit der Lastausgleichsbibliothek mit vom Anwender in Form von *C*-Funktionen bereitgestellten Algorithmen gewährleistet wird. Der im CM-task Compilerframework verwendete Lastausgleichsalgorithmus wird in Abschnitt 5.6 beschrieben.

Die *Managementfunktion* beinhaltet die für einen Lastausgleich notwendigen Operationen auf der Schedulestruktur eines Schleifenrumpfes oder Bedingungszweiges. Konkret führt die

Managementfunktion nacheinander die folgenden Arbeitsschritte aus.

(i) **Bereitstellung der gemessenen Ausführungszeiten:**
In diesem Arbeitsschritt werden die gemessenen Ausführungszeiten aus der Schedulestruktur extrahiert und allen beteiligten Prozessoren zur Verfügung gestellt.

(ii) **Ausführung des Lastausgleichsalgorithmus:**
Der Lastausgleichsalgorithmus wird für jede Menge zeitgleich ausgeführter Module des durch die Schedulestruktur beschriebenen Schleifenrumpfes oder Bedingungszweiges ausgeführt.

(iii) **Anpassung der Schedulestruktur:**
Falls der Lastausgleichsalgorithmus eine Anpassung der Prozessorgruppengrößen von zeitgleich ausgeführten Modulen vorgenommen hat, werden entsprechend angepasste MPI Kommunikatoren erzeugt und in der Schedulestruktur abgelegt.

(iv) **Rekursiver Lastausgleich:**
Falls durch eine Lastausgleichsoperation die Prozessoranzahl einer Schleife, einer Bedingung oder eines Verbundmodulaufrufs verändert wurde, erfolgt eine Anpassung der im Schleifenrumpf, Bedingungszweig bzw. aufgerufenem Verbundmodul verwendeten Prozessorgruppen durch einen rekursiven Aufruf der Managementfunktion mit der Schedulestruktur des jeweiligen Konstrukts.

4.9. Implementierung der Basismodule

Die Implementierung der Basismodule erfolgt durch den Anwendungsentwickler in Form von parallelen MPI+*C*-Funktionen[1]. Die Schnittstelle einer derartigen Funktion umfasst die im Spezifikationsprogramm definierten formalen Parameter der entsprechenden Basismoduldefinition sowie als letzten Parameter einen MPI Kommunikator. Dieser Kommunikator beschreibt die ausführende Prozessorgruppe des Basismoduls und wird zur Laufzeit durch das Koordinationsprogramm zur Verfügung gestellt.

Die Übergabe der im Spezifikationsprogramm definierten formalen Parameter erfolgt abhängig vom spezifizierten Datentyp und vom spezifizierten Zugriffstyp eines formalen Parameters. Eingabe- und Ausgabeparameter werden entweder als Wertparameter (Basisdatentypen) oder als Referenzparameter (Feld- und Nutzerdatentypen) übergeben. Kommunikationsparameter werden in Form einer Kommunikationsstruktur übergeben. Diese Datenstruktur beinhaltet neben einem Verweis auf den Speicherplatz des Kommunikationsparameters auch einen externen Kommunikator, der die Prozessoren aller Basismodule mit demselben Kommunikationsparameter beinhaltet. Der konkrete Aufbau der Kommunikationsstruktur ist auf Seite 64 angegeben.

Listing 11 zeigt ein Beispiel für die Verwendung externer Kommunikationsoperationen innerhalb einer Basismodulimplementierung. Als Beispiel wird ein Datenaustausch zwischen zwei zeitgleich ausgeführten Instanzen desselben Basismoduls betrachtet. Ein derartiger Datenaustausch entspricht dem Kommunikationsmuster I (s. Abbildung 9 auf Seite 44). Zeilen 1-4 des Listings zeigen den entsprechenden Ausschnitt aus dem Spezifikationsprogramm.

[1] Der Einsatz von *FORTRAN*-Routinen ist mit Hilfe von Wrapper-Funktionen möglich. Diese Funktionen müssen die durch das Koordinationsprogramm bereitgestellten MPI Kommunikatoren für *C*-Programme mit Hilfe der MPI Funktion `MPI_Comm_c2f()` in ein äquivalentes Konstrukt für *FORTRAN* übersetzen.

Listing 11: Beispiel für die Spezifikation einer Kommunikation zwischen zwei zeitgleich ausgeführten Instanzen desselben Basismoduls (Zeilen 1-4) und für die Verwendung externer Kommunikationsoperationen innerhalb der Basismodulimplementierung (Zeilen 5-17).

```
   // Spezifikation
1  cpar {
2    BM_example (comm1);
3    BM_example (comm1);
4  }

   // Basismodulimplementierung
5  void BM_example (COMM_DESC comm1, MPI_Comm groupcomm) {
6    int j, pid;
7    MPI_Status status;

     // Prozessor-ID innerhalb des Gruppenkommunikators bestimmen
8    MPI_Comm_rank (groupcomm, &pid);

     // Datenaustausch zwischen zwei kommunizierenden Modulen
9    if (comm1.groupid == 0) {
10     if (pid == 0) {
11       for (j = 0; j < comm1.gsizes[1]; j++)
         /* Prozessor 0 der Gruppe 0 sendet an alle Prozessoren
            der Gruppe 1 */
12         MPI_Send (comm1.data, 1, MPI_INT, comm1.goffsets[1]+j,
               0, comm1.ecomm);
13     }
14   } else if (comm1.groupid == 1) {
       // Gruppe 1 empfängt von Prozessor 0 der Gruppe 0
15     MPI_Recv (comm1.data, 1, MPI_INT, comm1.goffsets[0], 0,
           comm1.ecomm, &status);
16   }
17 }
```

Zeile 9 führt eine Fallunterscheidung anhand der lokalen Gruppennummer `comm1.groupid`, die innerhalb der Kommunikationsstruktur `comm1` bereitgestellt wird. Für den in Zeile 2 definierten Modulaufruf ist diese Gruppennummer 0, wohingegen der Modulaufruf in Zeile 3 eine Gruppennummer von 1 besitzt. Diese Nummern ergeben sich aus der Reihenfolge der Modulaufrufe im Spezifikationsprogramm.

Zeilen 10 bis 12 enthalten Programmcode für Sendeoperationen von einem ausgewählten Prozessor der Modulinstanz mit Gruppennummer 0 an alle ausführenden Prozessoren der Modulinstanz mit Gruppennummer 1. Die korrespondierenden Empfangsoperationen befinden sich in Zeile 15. Die Anzahl der ausführenden Prozessoren des Basismoduls mit Gruppennummer 1 wird in der Schedulingphase des CM-task Compilers festgelegt und über das Strukturelement `comm1.gsizes[1]` der Kommunikationsstruktur zur Laufzeit bereitgestellt. Die Kommunikation erfolgt über den externen Kommunikator `comm1.ecomm`, der die ausführenden Prozessoren beider Basismodulinstanzen enthält. Die Position des ersten ausführenden Prozessors der Modulinstanz `i` wird durch das Strukturelement `comm1.goffsets[i]` angegeben.

4.10. Zusammenfassung und verwandte Arbeiten

In diesem Kapitel wurden die im CM-task Compilerframework verwendeten Sprachen beschrieben und die Funktionalität der Datenumverteilungsbibliothek und Lastausgleichsbibliothek des Compilerframeworks vorgestellt. Die Sprachen werden durch den Anwendungsentwickler zur Spezifikation von parallelen Algorithmen und Plattformen und durch den CM-task Compiler zur Darstellung der Ergebnisse der Transformationsschritte verwendet.

Die plattformunabhängige Spezifikationssprache für parallele Algorithmen beinhaltet die Definition von Datentypen, Datenverteilungstypen und daten- und taskparallelen Programmteilen in Form von Basis- bzw. Verbundmodulen. Die Struktur von Verbundmodulen wird durch einen hierarchischen Modulausdruck beschrieben, der die Abhängigkeiten und Unabhängigkeiten zwischen Teilberechnungen definiert. Die zugrunde liegende Grammatik basiert auf der Spezifikationssprache des TwoL-Modells[92], enthält aber Erweiterungen zur Definition von Kommunikationsabhängigkeiten zwischen zeitgleich ausgeführten Modulen. Im Gegensatz zu parallelisierenden Compilern wie Paradigm [56], die aus einem sequentiellen Eingabeprogramm eine parallele Implementierung erzeugen, drückt das Spezifikationsprogramm des CM-task Compilers den maximal verfügbaren Grad an Taskparallelität aus.

Die Definition von Datenverteilungstypen in der Spezifikationssprache basiert auf parametrisierten Datenverteilungsvektoren. Diese Darstellungsform erlaubt die Spezifikation regelmäßiger Verteilungen auf einer variablen Prozessoranzahl. Die Beschreibung von Datenverteilungen ist Bestandteil vieler datenparalleler Sprachen, z.B. von Fortran D [47], HPF [46], High Performance C [114], HPJava [9] oder Vienna Fortran 90 [6], wobei häufig die Daten auf alle verfügbare Prozessoren verteilt werden. Die Definition regelmäßiger Datenverteilungen auf Prozessorgruppen durch Angabe einer Blockgröße und eines Skipfaktors wurde im Paradigm-Compiler [84] verwendet.

Die durch die Transformationsschritte des CM-task Compilers getroffenen Entscheidungen basieren auf Kosteninformationen für die im Spezifikationsprogramm definierten Basismodule. Die Spezifikation dieser Kosten erfolgt durch den Anwender in Form parametrisierter Laufzeitformeln, die aus einem Berechnungs- und einem Kommunikationsteil bestehen. Der Kommunikationsteil wird mit Hilfe von Kostenformeln für die verwendeten MPI Kommunikationsoperationen angegeben, die in einer separaten Plattformspezifikation zusammengefasst werden. Andere Ansätze zur Darstellung von Kosten paralleler Programme sind z.B. das Amdahlsche Gesetz [2], BSP [110], LogP [15] oder LogGP [1]. Aus diesen Modellen ergeben sich ebenfalls Kostenformeln in geschlossener Form, die in einem CM-task Spezifikationsprogramm verwendet werden können.

Das erweiterte Spezifikationsprogramm, das Rahmenprogramm und das erweiterte Rahmenprogramm repräsentieren Zwischenergebnisse des CM-task Compilers, die durch Hinzufügen zusätzlicher Annotationen entstehen. Annotationen werden bspw. für die vorhandenen Daten- und Kommunikationsabhängigkeiten zwischen Modulaufrufen, für die zur Ausführung zu verwendenden Prozessorgruppen oder für benötigte Datenumverteilungsoperationen verwendet. Das Konzept eines Rahmenprogrammes zur Darstellung von Designentscheidungen eines Transformationswerkzeuges wurde ebenfalls im TwoL-Modell verwendet [95, 37].

Die Datenumverteilungsbibliothek des CM-task Compilerframeworks unterstützt regelmäßige Datenverteilungen für mehrdimensionale Felder und erlaubt die Verwendung beliebiger Quell- und Zielprozessorgruppen. Die Realisierung der Umverteilungsoperationen erfolgt durch Einzeltransferoperationen, die im allgemeinen Fall Daten von jedem Quellprozessor zu jedem

Zielprozessor übertragen. Zur Reduzierung des zur Laufzeit entstehenden Kommunikationsoverheads unterstützt die Bibliothek vorberechnete Kommunikationsmuster, die durch den CM-task Compiler in ein erzeugtes Koordinationsprogramm eingebunden werden.

Datenumverteilungsoperationen werden für viele datenparallele Sprachen wie HPF [46] benötigt, z.B. zur Unterstützung von Zuweisungsoperationen $A = B$ für mehrdimensionale Felder A und B mit unterschiedlichen Datenverteilungstypen. DaRel[57] ist eine Datenumverteilungsbibliothek für HPF Programme, die wie die Datenumverteilungsbibliothek des CM-task Compilerframeworks die zur Laufzeit auszuführenden Kommunikationsoperationen zur Compilezeit vorberechnen kann. Die Datenumverteilungsoperationen sind in DaRel als atomare Operation realisiert, an der sowohl Quell- als auch Zielprozessoren beteiligt sind , und können wahlweise Punkt-zu-Punkt Kommunikationsoperationen oder kollektive Kommunikationsoperationen wie z.B. Broadcastoperationen verwenden.

DRDlib [99] ist eine Datenumverteilungsbibliothek für mehrdimensionale Felder, die kollektive MPI Kommunikationsoperationen zur Realisierung der Umverteilungen verwendet. Aus diesem Grund müssen Quell- und Zielprozessoren die Umverteilungsoperationen gemeinsam ausführen, während die Datenumverteilungsbibliothek des CM-task Compilerframeworks separate Sende- und Empfangsphasen für die Quell- bzw. Zielprozessoren verwendet. Eine weitere Bibliothek mit Unterstützung von Datenumverteilungen ist ScaLAPACK [81]. Im Unterschied zur Datenumverteilungsbibliothek des CM-task Compilerframeworks bieten diese Bibliotheken keine Compilerunterstützung zur Integration der Bibliotheksfunktionen in eine parallele Anwendung.

Die Lastausgleichsbibliothek des CM-task Compilerframeworks unterstützt die dynamische Anpassung der in einem Koordinationsprogramm verwendeten Prozessorgruppen an die zugrundeliegende parallele Plattform. Eine derartige Anpassung ist speziell für heterogene Zielsysteme sinnvoll, da die Größe der verwendeten Prozessorgruppen auf die Geschwindigkeit der enthaltenen Prozessoren abgestimmt werden kann.

Ein dynamischer Lastausgleich ist insbesondere auch für taskparallele Anwendungen mit irregulärem Berechnungen oder dynamisch erzeugten Tasks wichtig. Zu diesem Zweck wurden verschiedene Frameworks und Bibliotheken entwickelt, z.B. SAMBA [80], PREMA [4] oder Zoltan [18]. Diese Ansätze zielen auf eine gleichmäßige Verteilung einzelner Tasks auf die verfügbaren Prozessoren ab. Im Gegensatz dazu besteht die Aufgabe der Lastausgleichsbibliothek des CM-task Compilerframeworks in der Ermittlung einer geeigneten Prozessoranzahl für zeitgleich ausgeführte parallele Tasks.

Eine Klassifikation von Lastausgleichsverfahren wurde in [76] eingeführt. Nach dieser Klassifikation ist der Lastausgleichsansatz des CM-task Compilerframeworks

- periodisch, d.h. der Lastausgleich erfolgt an festen Programmpunkten;
- verteilt, d.h. Lastausgleichsoperationen werden durch mehrere Prozessoren ausgeführt;
- synchron, d.h. Lastausgleichsoperationen werden durch die beteiligten Prozessoren zeitgleich ausgeführt und
- global, d.h. alle an einer Lastausgleichsoperation beteiligten Prozessoren besitzen die benötigten Leistungsdaten.

5. Realisierung des CM-task Compilerframeworks

In diesem Kapitel wird die Realisierung der Komponenten des CM-task Compilerframeworks beschrieben. Abbildung 20 gibt anhand der in Abschnitt 2.2 vorgestellten Architektur des Compilerframeworks einen Überblick dieses Kapitels. Abschnitt 5.7 fasst dieses Kapitel zusammen und diskutiert verwandte Arbeiten.

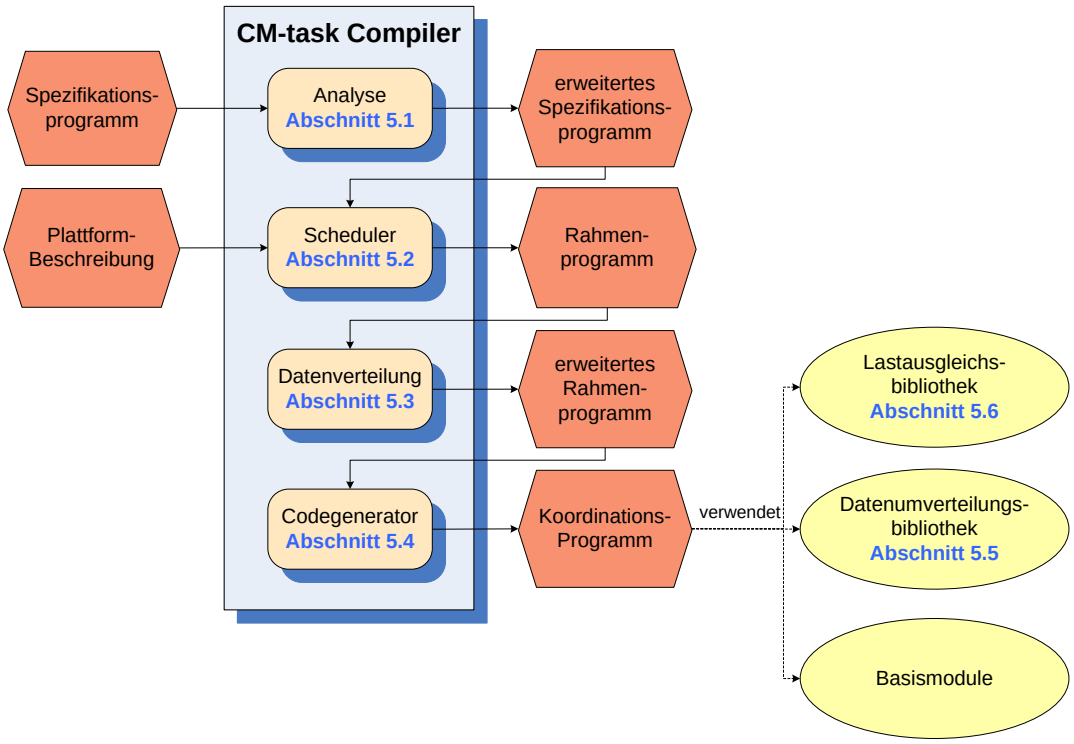


Abbildung 20: Architektur des CM-task Compilerframeworks mit Angabe der Abschnitte, die die Realisierung der Compilerkomponenten des CM-task Compilers bzw. der enthaltenen Bibliotheken beschreiben.

5.1. Analysephase des CM-task Compilers

Die Aufgabe der Analysephase besteht im Erkennen der in einem gegebenem CM-task Spezifikationsprogramm implizit über Variablennamen definierten Daten- und Kommunikationsabhängigkeiten und der Ausgabe der erkannten Abhängigkeiten in Form eines erweiterten Spezifikationsprogrammes. Als interne Datenstruktur der Analysephase werden die *abstrakten Syntaxbäume* der im Spezifikationsprogramm definierten Verbundmodule verwendet. Diese Syntaxbäume werden durch einen Parser mit gekoppeltem Übersetzungsschema aus dem Spezifikationsprogramm erzeugt.

Der abstrakte Syntaxbaum eines Verbundmoduls ist wie folgt aufgebaut. Der Wurzelknoten des Syntaxbaums speichert den Namen und die im Spezifikationsprogramm definierten Eingabe- und Ausgabeparameter des jeweiligen Verbundmoduls und besitzt einen Kindknoten. Die inneren Knoten des Syntaxbaums sind mit `seq`, `par`, `cpar`, `for`, `while`, `parfor`, `cparfor`

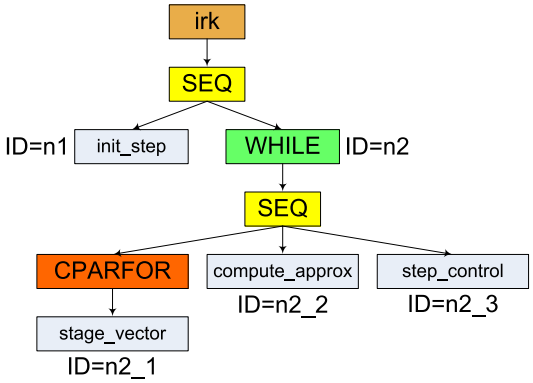


Abbildung 21: Abstrakter Syntaxbaum des Spezifikationsprogrammes für Iterierte Runge-Kutta Verfahren (s. Listing 4 auf Seite 50) mit annotierten Identifikatoren (ID). Beispielsweise wird dem Aufruf des Basismoduls `stage_vector` der Identifikator `n1` zugeordnet.

oder `if` bezeichnet und repräsentieren die im Modulausdruck des jeweiligen Verbundmoduls verwendeten Konstruktoren. Die Blattknoten des Syntaxbaums repräsentieren die im jeweiligen Verbundmodul definierten Basis- und Verbundmodulaufrufe. Jedes Konstrukt eines gegebenen Spezifikationsprogrammes wird durch genau einen Knoten im zugehörigen abstrakten Syntaxbaum dargestellt. Im Folgenden wird nicht mehr zwischen den Knoten des Syntaxbaums und dem jeweils zugehörigen Konstrukt des entsprechenden Spezifikationsprogrammes unterschieden. Abbildung 21 zeigt den abstrakten Syntaxbaum des Spezifikationsprogrammes für IRK Verfahren.

Die Analysephase besteht aus drei Arbeitsschritten, die für jeden abstrakten Syntaxbaum eines im Spezifikationsprogramm definierten Verbundmoduls nacheinander ausgeführt werden. Der erste Schritt annotiert eindeutige Identifikatoren an den Knoten eines gegebenen abstrakten Syntaxbaums. Im zweiten Schritt werden die Kommunikationsabhängigkeiten und im dritten Schritt die Datenabhängigkeiten in einem gegebenen abstrakten Syntaxbaum erkannt und an den Knoten des jeweiligen Syntaxbaums annotiert. Im Folgenden werden diese drei Schritte genauer beschrieben.

5.1.1. Festlegen eindeutiger Identifikatoren

Die Ausgabe der in den nachfolgenden Schritten ermittelten Daten- und Kommunikationsabhängigkeiten im erweiterten Spezifikationsprogramm erfordert Identifikatoren zur eindeutigen Bezeichnung der an der jeweiligen Abhängigkeit beteiligten Konstrukte. Dazu wird in diesem Schritt an den `for`-, `while`-, `if`- und Blattknoten eines gegebenen abstrakten Syntaxbaums jeweils ein eindeutiger Identifikator annotiert.

Die Annotation erfolgt durch eine top-down Traversierung des entsprechenden Syntaxbaums, wobei der an einem Knoten k annotierte Identifikator inkrementell anhand des Pfades von der Syntaxbaumwurzel zu k aufgebaut wird. Konkret wird wie folgt vorgegangen. Die Traversierung beginnt mit dem Identifikator „n“ in der Syntaxbaumwurzel. An jedem Syntaxbaumknoten k mit n, $n > 1$, Kindknoten wird der Identifikator für Kindknoten i, $i = 1,\ldots,n$, durch Anhängen von i an den Identifikator von k ermittelt. Besitzt ein Syntaxbaumknoten k nur einen Kindknoten,

Tabelle 1: Regeln zur Annotation der Kommunikationsmenge $k.COMM$ an einem Syntaxbaumknoten k.

Fall für Syntaxbaumknoten k	Konstruktionsregel für Kommunikationsmenge $k.COMM$
k ist Basismodulaufruf BM $(a_1,\ldots,a_n)$	$k.COMM = \{(a_i, k.ID) \mid$ Zugriffstyp des i-ten formalen Parameters von BM ist `comm`; $i = 1,\ldots,n\}$
k ist Verbundmodulaufruf VM $(a_1,\ldots,a_n)$	$k.COMM = \emptyset$
k ist **cpar**-Knoten mit n Kindknoten $c_1,\ldots,c_n$	$k.COMM = \cup_{i=1,\ldots,n} c_i.COMM$

so wird der Identifikator von k an den Kindknoten weitergegeben.

Falls der an einen `for`-, `while`-, `if`- oder Blattknoten zu annotierende Identifikator bereits vergeben ist (z.B. falls ein Blattknoten einziges Kind eines `for`-Knotens ist), wird diesem Identifikator der String *„_1“* angehängt. Diese Konstruktionsvorschrift erzeugt Identifikatoren, die den Aufbau des abstrakten Syntaxbaums widerspiegeln und eine möglichst geringe Länge besitzen. Abbildung 21 zeigt die am Syntaxbaum für IRK Verfahren annotierten Identifikatoren.

5.1.2. Bestimmung der Kommunikationsabhängigkeiten

Eine Kommunikationsabhängigkeit in einem CM-task Spezifikationsprogramm besteht zwischen zwei oder mehr Basismodulaufrufen, die (a) durch einen `cpar`- oder `cparfor`-Konstruktor zusammengefasst werden und die (b) einen gemeinsamen Kommunikationsparameter besitzen. Als Kommunikationsparameter wird ein aktueller Parameter eines Basismodulaufrufs im Spezifikationsprogramm bezeichnet, dessen Zugriffstyp in der Schnittstellendefinition des aufgerufenen Basismoduls als `comm` spezifiziert ist. Eine Kommunikationsabhängigkeit zeigt an, dass die entsprechenden Basismodule während ihrer Ausführung miteinander kommunizieren.

Die Bestimmung der Kommunikationsabhängigkeiten für einen gegebenen abstrakten Syntaxbaum eines Verbundmoduls erfolgt in zwei Teilschritten. Im ersten Schritt wird an den Knoten des Syntaxbaums eine *Kommunikationsmenge* annotiert. Eine Kommunikationsmenge besteht aus Paaren (a, id), wobei *id* der Identifikator eines Basismodulaufrufs und a ein durch diesen Basismodulaufruf verwendeter Kommunikationsparameter ist. Die Kommunikationsmenge eines Syntaxbaumknotens k gibt die im Teilbaum des abstrakten Syntaxbaums mit Wurzel k verwendeten Kommunikationsparameter mit den zugehörigen Basismodulaufrufen an. Im zweiten Schritt werden aus den berechneten Kommunikationsmengen die zu annotierenden Kommunikationsabhängigkeiten bestimmt.

Annotation der Kommunikationsmengen. Die Kommunikationsmenge eines Syntaxbaumknotens k wird aus den Kommunikationsmengen der Kindknoten von k berechnet, d.h. es findet eine bottom-up Attributierung des abstrakten Syntaxbaums statt. Tabelle 1 zeigt die Regeln zur Berechnung der Kommunikationsmenge $k.COMM$ eines Knotens k, wobei $k.ID$ den zuvor an k annotierten Identifikator bezeichnet. Die Regel für Verbundmodulaufrufe berücksichtigt, dass

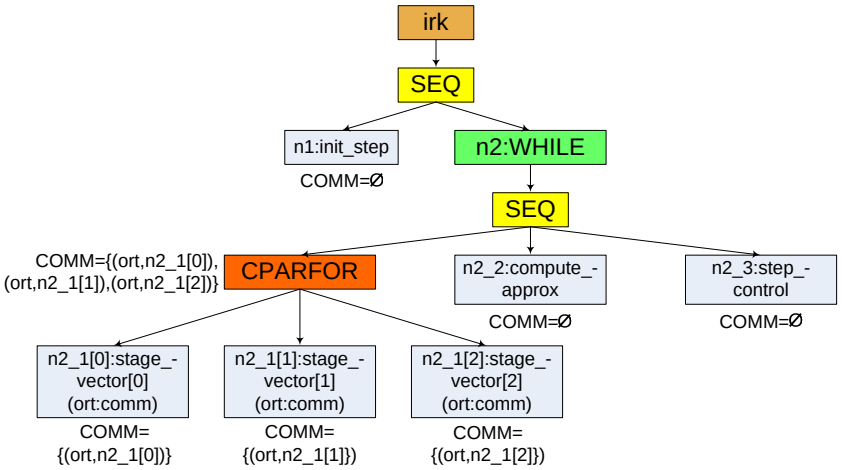


Abbildung 22: Annotation der Kommunikationsmengen (`COMM`) am abstrakten Syntaxbaum für Iterierte Runge-Kutta Verfahren. In den Knoten sind die zugeordneten Identifikatoren (z.B. `n1`) und die verwendeten Kommunikationsparameter (z.B. `ort`) angegeben. Die Iterationen der `cparfor`-Schleife werden durch separate Teilbäume dargestellt, in denen den Identifikatoren die entsprechende Schleifeniteration in eckigen Klammern angehängt ist.

in der CM-task Spezifikationssprache keine Kommunikationsparameter für Verbundmodule definiert werden können. An den nicht in der Tabelle aufgeführten `seq`-, `par`-, `for`-, `while`- und `if`-Knoten wird keine Kommunikationsmenge annotiert, da zwischen den durch diese Konstruktoren zusammengefassten Programmteilen keine Kommunikationsabhängigkeiten bestehen. Für `parfor`- und `cparfor`-Knoten werden spezielle Regeln verwendet, die im Folgenden erläutert werden.

Behandlung von `parfor`- und `cparfor`-Schleifen. Die Behandlung von `parfor`- und `cparfor`-Schleifen erfolgt in drei Schritten. Im ersten Schritt werden die Schleifen entrollt, d.h. im Syntaxbaum wird der Teilbaum unterhalb des entsprechenden `parfor`- bzw. `cparfor`-Knotens entsprechend des Iterationsraumes der Schleife repliziert. Die so entstehenden Teilbäume werden unterhalb des entsprechenden `parfor`- bzw. `cparfor`-Knotens in den Syntaxbaum eingefügt und den annotierten Identifikatoren in diesen Teilbäumen wird die jeweilige Schleifeniteration angehängt. Abbildung 22 zeigt den Syntaxbaum für IRK Verfahren mit entrollter `cparfor`-Schleife.

Im zweiten Schritt erfolgt die Bestimmung der Kommunikationsabhängigkeiten für den modifizierten Syntaxbaum gemäß den Regeln aus Tabelle 1, wobei die `cparfor`-Knoten analog zu den `cpar`-Knoten behandelt werden, d.h. die Kommunikationsmenge eines `cparfor`-Knotens ergibt sich aus der Vereinigung der Kommunikationsmengen der einzelnen Schleifeniterationen. Auf diese Weise können im nachfolgenden Schritt auch Kommunikationsabhängigkeiten zwischen verschiedenen Iterationen einer `cparfor`-Schleife erkannt werden.

Im dritten Schritt erfolgt die Rücktransformantion des Syntaxbaums in die ursprüngliche Darstellung. Dabei werden die im ersten Schritt replizierten Syntaxbaumknoten auf den entsprechenden ursprünglichen Syntaxbaumknoten abgebildet, wobei die Kommunikationsmenge

des ursprünglichen Syntaxbaumknotens aus der Vereinigung der Kommunikationsmengen der auf diesen Knoten abgebildeten replizierten Knoten berechnet wird.

Annotation der Kommunikationsabhängigkeiten. Die Annotation der Kommunikationsabhängigkeiten erfolgt durch einen Tiefensuchlauf über einen gegebenen abstrakten Syntaxbaum mit annotierten Kommunikationsmengen. Bei dieser Tiefensuche werden nur diejenigen `cpar`- und `cparfor`-Knoten betrachtet, die nicht Kindknoten eines anderen `cpar`- oder `cparfor`-Knotens sind. Damit wird gewährleistet, dass alle Zugriffe von zeitgleich auszuführenden Basismodulen auf einen bestimmten Kommunikationsparameter berücksichtigt werden.

Die Annotation der Kommunikationsabhängigkeiten an einem derartigen Syntaxbaumknoten k mit annotierter Kommunikationsmenge $k.COMM$ wird wie folgt durchgeführt. Zunächst wird $k.COMM$ in disjunkte Teilmengen partitioniert, so dass Paare einer Teilmenge denselben Kommunikationsparameter und Paare verschiedener Teilmengen unterschiedliche Kommunikationsparameter besitzen. Für jede derart konstruierte Teilmenge wird eine Kommunikationsabhängigkeit an k annotiert. Die an der einzufügenden Kommunikationsabhängigkeit beteiligten Basismodulaufrufe sind durch die in den Paaren der entsprechenden Teilmenge der Kommunikationsmenge gespeicherten Identifikatoren gegeben.

Beispiel: Iterierte Runge-Kutta Verfahren

Abbildung 22 zeigt den abstrakten Syntaxbaum des Spezifikationsprogrammes für IRK Verfahren, in dem die `cparfor`-Schleife entrollt dargestellt ist. Die Kommunikationsmengen der Basismodulaufrufe ergeben sich aus den verwendeten Kommunikationsparametern, die in den entsprechenden Syntaxbaumknoten angegeben sind. Die Kommunikationsmenge der `cparfor`-Schleife ergibt sich aus der Vereinigung der Kommunikationsmengen der im Schleifenrumpf enthaltenen Aufrufe des Basismoduls `stage_vector`.

Die Annotation der Kommunikationsabhängigkeiten erfolgt am `cparfor`-Knoten des Syntaxbaums. Dabei wird eine Kommunikationsabhängigkeit für den Kommunikationsparameter `ort` eingefügt. An dieser Kommunikationsabhängigkeit sind alle drei Aufrufe des Basismoduls `stage_vector` beteiligt.

5.1.3. Bestimmung der Datenabhängigkeiten

Eine Datenabhängigkeit in einer CM-task Anwendungsspezifikation besteht zwischen einem schreibenden Zugriff auf eine Variable a und einem nachfolgend auszuführenden Lesezugriff auf a, falls zwischen diesem Schreibzugriff und diesem Lesezugriff kein anderer Schreibzugriff auf a ausgeführt wird. Eine Datenabhängigkeit zeigt an, dass vom Schreibzugriff Daten für den Lesezugriff bereitgestellt werden und dass abhängig von den jeweiligen Datenverteilungstypen für die Variable a und den jeweiligen ausführenden Prozessorgruppen eine Datenumverteilungsoperation zwischen Schreib- und Lesezugriff benötigt wird.

Zur Bestimmung der Datenabhängigkeiten für einen gegebenen abstrakten Syntaxbaum eines Verbundmoduls wird zunächst an jedem Knoten dieses Syntaxbaums eine *Lesemenge* und eine *Schreibmenge* annotiert. Die Lese- und Schreibmengen enthalten Paare (a, id) bestehend aus einer Variable a und einem Identifikator id, wobei ein derartiges Paar einen Lese- bzw. Schreibzugriff auf a durch das mit id bezeichnete Konstrukt repräsentiert.

Die Lesemenge eines Syntaxbaumknotens k enthält alle Lesezugriffe innerhalb des Teilbaums des abstrakten Syntaxbaums mit Wurzel k, die ihre Werte von einem Schreibzugriff außerhalb dieses Teilbaums erhalten. Die Schreibmenge eines Knotens k enthält alle im Teilbaum des abstrakten Syntaxbaums mit Wurzel k ausgeführten Schreibzugriffe, deren geschriebene Werte nach kompletter Abarbeitung der durch diesen Teilbaum repräsentierten Berechnungen erhalten bleiben, d.h. nicht von einem anderen sich in diesem Teilbaum befindenden Schreibzugriff überschrieben werden.

Aus den berechneten Lese- und Schreibmengen werden in einem nachfolgenden Schritt die zu annotierenden Datenabhängigkeiten bestimmt. Im Folgenden wird die Konstruktion der Lese- und Schreibmengen und die Annotation der Datenabhängigkeiten beschrieben.

Berechnung der Lese- und Schreibmengen

Die Berechnung der Lese- und der Schreibmenge eines Syntaxbaumknotens k erfolgt mit Hilfe der Lese- und Schreibmengen der Kindknoten von k, d.h. es findet eine bottom-up Attributierung des abstrakten Syntaxbaums statt. Tabelle 2 zeigt die Konstruktionsregeln zur Bestimmung der Lesemenge $k.READ$ und der Schreibmenge $k.WRITE$ eines Syntaxbaumknotens k. Dabei bezeichnet $k.ID$ den zuvor an k annotierten Identifikator.

Die Lese- und Schreibmenge eines Basismodulaufrufs wird direkt aus den aktuellen Parametern des Modulaufrufs und den Zugriffstypen der korrespondierenden formalen Parameter bestimmt. Diese Zugriffstypen werden vom Anwendungsentwickler in der Schnittstellenbeschreibung des aufgerufenen Moduls spezifiziert. Die Berechnung der Lese- und Schreibmengen von Verbundmodulaufrufen erfolgt analog zu den Basismodulaufrufen und ist daher nicht in der Tabelle angegeben.

Die Konstruktionsregeln für `par`- und `cpar`-Knoten berücksichtigen die durch die zugehörigen Konstrukte definierte Datenunabhängigkeit zwischen den durch die entsprechenden Kindknoten repräsentierten Berechnungen. Für `seq`-Knoten ist die Reihenfolge der Kindknoten entscheidend, die sich aus der Reihenfolge der Definition der zugehörigen Berechnungen im Spezifikationsprogramm ergibt. Dies bedeutet, dass bspw. ein Schreibzugriff auf eine Variable a in einem Kindknoten i eines `seq`-Knotens mit n Kindknoten den in einem Kindknoten j, $1 \leq j < i \leq n$, dieses `seq`-Knotens geschriebenen Wert von a überschreibt. Die `parfor`- und `cparfor`-Knoten werden analog zu der Bestimmung der Kommunikationsabhängigkeiten (s. Abschnitt 5.1.2) behandelt, d.h. die entsprechenden Schleifen werden entrollt und die Lese- und Schreibmengen für jede Schleifeniteration separat berechnet.

Schleifen. Für `for`-Schleifen wird in diesem Schritt eine *virtuelle Parameterliste* konstruiert. Diese Parameterliste wird intern verwendet und ist für den Anwendungsentwickler nicht sichtbar. Virtuelle Eingabeparameter der Schleife sind alle Variablen, die in mindestens einem Paar der Lesemenge des zugehörigen Schleifenrumpfes enthalten sind. Diese Variablen erhalten ihren Wert (zumindest in der ersten Iteration der Schleife) von einem Schreibzugriff, der vor der entsprechenden Schleife ausgeführt wird. Virtuelle Ausgabeparameter der Schleife sind alle Variablen, die in mindestens einem Paar der Schreibmenge des zugehörigen Schleifenrumpfes enthalten sind. Die Werte dieser Variablen stehen nach Ablauf einer Schleifeniteration für Lesezugriffe in der darauffolgenden Iteration sowie für nach der entsprechenden Schleife ausgeführte Lesezugriffe zur Verfügung.

Tabelle 2: Konstruktionsregeln für die Lesemenge $k.READ$ und die Schreibmenge $k.WRITE$ eines Knotens k des abstrakten Syntaxbaums.

Fall für Syntaxbaumknoten k	Konstruktionsregeln für Lese- und Schreibmenge von k
k ist Basismodulaufruf BM $(a_1,\ldots,a_n)$	$k.READ = \{(a_i, k.ID) \mid$ Zugriffstyp des i-ten formalen Parameters von BM ist `in` oder `inout`; $i = 1,\ldots,n\}$ $k.WRITE = \{(a_i, k.ID) \mid$ Zugriffstyp des i-ten formalen Parameters von BM ist `out` oder `inout`; $i = 1,\ldots,n\}$
k ist **cpar**-Knoten mit n Kindknoten $c_1,\ldots,c_n$	$k.READ = \cup_{i=1,\ldots,n} c_i.READ$ $k.WRITE = \cup_{i=1,\ldots,n} c_i.WRITE$
k ist **par**-Knoten mit n Kindknoten $c_1,\ldots,c_n$	$k.READ = \cup_{i=1,\ldots,n} c_i.READ$ $k.WRITE = \cup_{i=1,\ldots,n} c_i.WRITE$
k ist **seq**-Knoten mit n Kindknoten $c_1,\ldots,c_n$	$k.READ = \cup_{i=1,\ldots,n}\{(a,id) \mid (a,id) \in c_i.READ$ und es gibt kein $(a,id') \in c_j.WRITE$ für $1 \le j < i\}$ $k.WRITE = \cup_{i=1,\ldots,n}\{(a,id) \mid (a,id) \in c_i.WRITE$ und es gibt kein $(a,id') \in c_j.WRITE$ für $i < j \le n\}$
k ist **for**-Knoten mit Kindknoten c	$k.READ = \{(a,k.ID) \mid (a,id') \in c.READ\}$ $k.WRITE = \{(a,k.ID) \mid (a,id') \in c.WRITE\}$
k ist **if**-Knoten mit einem Kindknoten c	$k.READ = \{(a,k.ID) \mid (a,id') \in c.READ \cup c.WRITE\}$ $k.WRITE = \{(a,k.ID) \mid (a,id') \in c.WRITE\}$
k ist **if**-Knoten mit zwei Kindknoten c_1 und c_2	$k.READ = \{(a,k.ID) \mid (a,id') \in c_1.READ \cup c_2.READ\} \cup$ $\{(a,k.ID) \mid (a,id') \in c_1.WRITE$ und es gibt kein $(a,id'') \in c_2.WRITE\} \cup$ $\{(a,k.ID) \mid (a,id') \in c_2.WRITE$ und es gibt kein $(a,id'') \in c_1.WRITE\}$ $k.WRITE = \{(a,k.ID) \mid (a,id') \in c_1.WRITE \cup c_2.WRITE\}$

Die Lese- und Schreibmengen eines `for`-Knotens werden wie für einen Modulaufruf konstruiert, der die entsprechenden virtuellen Eingabeparameter benötigt und die entsprechenden virtuellen Ausgabeparameter erzeugt. Die Behandlung von `while`-Knoten erfolgt analog zu den `for`-Knoten und ist nicht in Tabelle 2 aufgeführt.

Bedingungen. Für Bedingungen wird ebenfalls eine virtuelle Parameterliste konstruiert, wobei unterschiedliche Regeln für einseitige Bedingungen (d.h. Bedingungen ohne else-Zweig) und zweiseitige Bedingungen verwendet werden. Zunächst werden einseitige Bedingungen betrachtet, die im abstrakten Syntaxbaum durch einen `if`-Knoten mit einem Kindknoten dargestellt werden. Die virtuellen Eingabeparameter eines derartigen Konstrukts umfassen alle Variablen, die in mindestens einem Paar der Lesemenge oder der Schreibmenge des if-Zweigs enthalten sind. Die im if-Zweig geschriebenen Variablen zählen zu den Eingabeparametern der entsprechenden Bedingung, da erst zur Laufzeit der Anwendung entschieden wird, ob die Schreibzugriffe des if-Zweigs tatsächlich ausgeführt werden. Wird der if-Zweig nicht

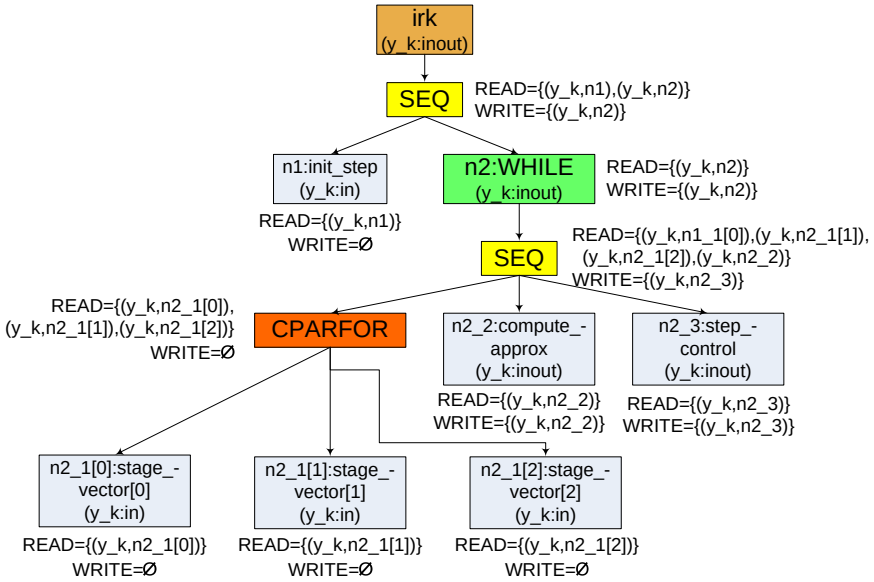


Abbildung 23: Syntaxbaum für das Hauptmodul des Spezifikationsprogrammes für Iterierte Runge-Kutta Verfahren mit berechneten Lese- und Schreibmengen (*READ* bzw. *WRITE*). Für die Berechnung dieser Mengen werden nur die Zugriffe auf die Variable `y_k` betrachtet, die in den entsprechenden Knoten in Klammern angegeben sind. Die drei Iterationen der `cparfor`-Schleife sind durch separate Teilbäume dargestellt, für die jeweils eigene Lese- und Schreibmengen berechnet werden.

ausgeführt, entsprechen die Werte der Ausgabeparameter der Bedingung den Werten der jeweiligen Variablen vor Abarbeitung der Bedingung.

Analoge Überlegungen liegen den Konstruktionsregeln für zweiseitige Bedingungen zugrunde. Virtuelle Eingabeparameter eines derartigen Konstrukts sind alle Variablen, die in einem Paar der Lesemenge eines Bedingungszweigs enthalten sind, sowie Variablen, die in genau einem Bedingungszweig geschrieben werden. Variablen, die in beiden Bedingungszweigen geschrieben werden, sind keine Eingabeparameter, da unabhängig vom tatsächlich ausgeführten Bedingungszweig ein neuer Wert durch das Bedingungskonstrukt erzeugt wird.

Beispiel: Iterierte Runge-Kutta Verfahren

Abbildung 23 zeigt den Syntaxbaum des Hauptmoduls des Spezifikationsprogrammes für IRK Verfahren (s. Listing 4 auf Seite 50) mit entrollter `cparfor`-Schleife und annotierten Lese- und Schreibmengen. Vereinfachend wird nur die Variable `y_k` berücksichtigt und auf die Betrachtung der Zugriffe auf die Variablen `x`, `h`, `y_k1`, `mu` und `mu1` verzichtet. Die Zugriffe auf `y_k` sind in den entsprechenden Syntaxbaumknoten angegeben. In der virtuellen Parameterliste der `while`-Schleife tritt `y_k` als Eingabe- und als Ausgabeparameter auf, da sowohl in der Lese- als auch in der Schreibmenge des Schleifenrumpfes ein Paar mit `y_k` enthalten ist.

In der Lesemenge des `seq`-Knotens im Schleifenrumpf der `while`-Schleife sind die Lesezugriffe durch die Modulaufrufe von `stage_vector` und `compute_approx` enthalten. Der

Tabelle 3: Konstruktionsregeln zur Annotation der Datenabhängigkeiten an einem Syntaxbaumknoten k.

Fall für Syntaxbaumknoten k	Regel für die Annotation der Datenabhängigkeiten

k ist **seq**-Knoten mit n Kindknoten $c_1,\ldots,c_n$	**for** $(i = 1,\ldots,n)$ **do**
	  foreach $((a,id) \in c_i.READ \setminus k.READ)$ **do** $\{$
	  finde $(a,id') \in c_j.WRITE$ mit $1 \leq j < i$ und j maximal;
	  annotiere Datenabhängigkeit mit Schreibzugriff id',
	  Lesezugriff id, Variable a an k;
	  $\}$

k ist **for**-Knoten mit Kindknoten c	**foreach** $((a,id) \in c.READ)$ **do**
	  annotiere Datenabhängigkeit mit Schreibzugriff $k.ID$,
	  Lesezugriff id, Variable a an k;
	foreach $((a,id) \in c.WRITE)$ **do**
	  annotiere Datenabhängigkeit mit Schreibzugriff id,
	  Lesezugriff $k.ID$, Variable a an k;

k ist Wurzelknoten mit Kindknoten c	**foreach** $((a,id) \in c.READ)$ **do** $\{$
	  if (a ist Eingabeparameter des durch k repräsentierten Verbundmoduls) **then**
	  annotiere Datenabhängigkeit mit Schreibzugriff k,
	  Lesezugriff id, Variable a an k;
	$\}$
	foreach (Ausgabeparameter a des durch k repräsentierten Verbundmoduls) **do** $\{$
	  finde $(a,id) \in c.WRITE$;
	  annotiere Datenabhängigkeit mit Schreibzugriff id,
	  Lesezugriff k, Variable a an k;
	$\}$

Lesezugriff durch den Basismodulaufruf `step_control` erfolgt auf den durch den Schreibzugriff des Modulaufrufs `compute_approx` geschriebenen Wert und ist daher nicht in dieser Lesemenge enthalten.

Annotation der Datenabhängigkeiten

Die Annotation der Datenabhängigkeiten erfolgt durch einen Tiefensuchlauf über einen gegebenen abstrakten Syntaxbaum mit annotierten Lese- und Schreibmengen. Die Konstruktionsregeln für einen Syntaxbaumknoten k sind in Tabelle 3 zusammengefasst.

An einem `seq`-Knoten werden diejenigen Datenabhängigkeiten annotiert, die zwischen einem Schreibzugriff und einem Lesezugriff in jeweils verschiedenen Teilbäumen unterhalb des jeweiligen `seq`-Knotens bestehen. An `for`-, `while`- und `if`-Knoten werden Datenabhängigkeiten eingefügt, an denen die virtuellen Eingabe- und Ausgabeparameter des entsprechenden Konstrukts beteiligt sind. Tabelle 3 zeigt dies exemplarisch für die `for`-Knoten. Am Wur-

zelknoten des abstrakten Syntaxbaums werden Datenabhängigkeiten für Lesezugriffe auf die Eingabeparameter des jeweiligen Verbundmoduls und für Schreibzugriffe innerhalb des Verbundmoduls eingefügt, die die Werte für die Ausgabeparameter des jeweiligen Verbundmoduls bereitstellen.

Beispiel: Iterierte Runge-Kutta Verfahren

Im Beispiel für IRK Verfahren (s. Abbildung 23 für die berechneten Lese- und Schreibmengen und Abbildung 15 auf Seite 54 für eine graphische Darstellung der Abhängigkeiten) werden eine Datenabhängigkeit am `seq`-Knoten innerhalb der `while`-Schleife, fünf Datenabhängigkeiten am `while`-Knoten und drei Datenabhängigkeiten am Wurzelknoten annotiert. Die an diesem `seq`-Knoten annotierte Abhängigkeit besteht zwischen dem Schreibzugriff durch `compute_approx` und dem Lesezugriff durch `step_control`. Die am Wurzelknoten und am `while`-Knoten annotierten Abhängigkeiten ergeben sich aus den Paaren in der Lese- und Schreibmenge des jeweiligen Kindknotens.

5.2. Schedulingphase des CM-task Compilers

Die Schedulingphase des CM-task Compilers erzeugt für eine gegebene Anwendungsspezifikation mit annotierten Daten- und Kommunikationsabhängigkeiten einen globalen, hierarchischen Schedule, der die Ausführungsreihenfolge und die ausführenden Prozessorgruppen der in der Anwendungsspezifikation definierten Modulaufrufe festlegt. Die Schedulingphase besteht aus sieben nacheinander ausgeführten Arbeitsschritten die in den nachfolgenden Unterabschnitten beschrieben und anhand eines erklärenden Beispiels (running example, s. Abbildung 24) verdeutlicht werden. Der letzte Arbeitsschritt erzeugt das Rahmenprogramm, das alle von der Schedulingphase getroffenen Entscheidungen zusammenfasst.

5.2.1. Konstruktion einer Hierarchie von CM-task Graphen

Der erste Arbeitsschritt der Schedulingphase erzeugt für jedes im gegebenen erweiterten Spezifikationsprogramm definierte Verbundmodul eine Menge von CM-task Graphen. Dabei wird ein CM-task Graph für jeden *Programmblock* des jeweiligen Verbundmoduls erzeugt. Ein Programmblock eines Verbundmoduls ist das Verbundmodul selbst sowie jeder im entsprechenden Verbundmodul definierte Bedingungszweig und jeder im entsprechenden Verbundmodul definierte Schleifenrumpf einer `for`- oder `while`-Schleife.

Die Knoten des erzeugten CM-task Graph eines Programmblockes repräsentieren die im jeweiligen Programmblock definierten Basis- und Verbundmodulaufrufe, Schleifen (`for` und `while`) und Bedingungen. Jeder dieser Graphknoten speichert einen Verweis auf den zugehörigen Knoten im abstrakten Syntaxbaum des erweiterten Spezifikationsprogrammes. Schleifen und Bedingungen repräsentierende Graphknoten speichern zusätzlich einen Verweis auf den CM-task Graph des zugehörigen Schleifenrumpfes bzw. auf die CM-task Graphen der zugehörigen Bedingungszweige, so dass eine Hierarchie der CM-task Graphen eines Verbundmoduls entsteht.

Zusätzlich zu den bereits erwähnten Graphknoten enthält jeder konstruierte CM-task Graph einen eindeutigen *Startknoten* und einen eindeutigen *Endknoten*. Diese Knoten repräsentieren die Eingabe bzw. die Ausgabe des zugehörigen Programmblockes und sind nicht mit Berechnungen assoziiert.

```
cmmain C() {
 var x:vektor;
 par {
  n1: B();
  n2: while ([...])#5 {
   parfor (i=1:3) {
    n2_1: A(x[i]);
   }
   prels {
    (n2->n2_1[1],{x[1]}),
    (n2->n2_1[2],{x[2]}),
    (n2->n2_1[3],{x[3]})
}}}}
```

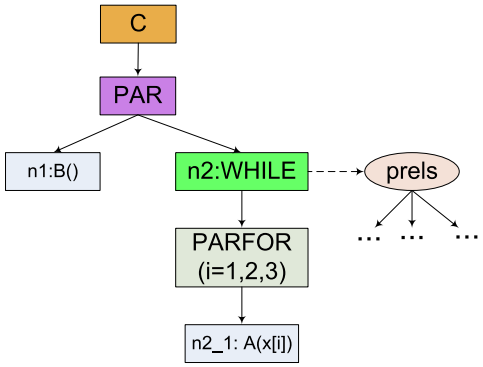


Abbildung 24: Erklärendes Beispiel für den Ablauf der Schedulingphase. Links: Hauptmodulde-finition im erweiterten Spezifikationsprogramm. Rechts: Abstrakter Syntaxbaum des Hauptmoduls. Die drei im `prels`-Konstrukt gespeicherten Datenabhängigkeiten sind durch drei abgehende Pfeile illustriert.

Die Konstruktion eines CM-task Graph für einen gegebenen abstrakten Syntaxbaum mit annotierten Daten- und Kommunikationsabhängigkeiten ist in Algorithmus 4 skizziert. Der Algorithmus führt eine bottom-up Attributierung des Syntaxbaums durch. An jedem Knoten k des Syntaxbaums wird der zugehörige Teilgraph, der den Teilbaum des Syntaxbaums mit Wurzel k repräsentiert, in Form der Knotenmenge $k.V$ und der Kantenmenge $k.E$ annotiert. Am Wurzelknoten des Syntaxbaums werden dem für den entsprechenden Kindknoten konstruierten Teilgraphen der Startknoten und der Endknoten hinzugefügt, so dass der vollständige CM-task Graph entsteht.

Die angegebenen Konstruktionsregeln für `par`-, `cpar`- und `seq`-Knoten kombinieren die für die jeweiligen Kindknoten ermittelten Teilgraphen gemäß den entsprechenden Kompositionsregeln für erweiterte SP-Graphen (s. Definition 5 auf Seite 42).

An `cpar`-Knoten werden zusätzliche bidirektionale Kanten für die an diesen Knoten annotierten Kommunikationsabhängigkeiten eingefügt und an `seq`-Knoten sowie am Wurzelknoten werden zusätzliche gerichtete Kanten für die an diesen Knoten annotierten Datenabhängigkeiten eingefügt. Diese Kanten sind mit den zugrundeliegenden Variablen der entsprechenden Abhängigkeiten annotiert und werden im nachfolgenden Schritt zur Bestimmung der Kommunikationskosten zwischen den jeweils verbundenen Graphknoten genutzt.

Die `parfor`- und `cparfor`-Schleifen werden vor Ausführung von Algorithmus 4 entrollt (vgl. Abschnitt 5.1.2). Dies bedeutet, dass bspw. für Modulaufrufe in einem Schleifenrumpf einer derartigen Schleife entsprechend des Iterationsraumes der jeweiligen Schleife mehrere Graphknoten erzeugt werden, wobei jeder Graphknoten den Modulaufruf in einer der Schleifeniterationen repräsentiert.

Die Konstruktionsvorschriften für die `for`-, `while`- und `if`-Knoten des abstrakten Syntaxbaums sind nicht in Algorithmus 4 enthalten. Für einen derartigen Knoten k wird zunächst analog zu dem Wurzelknoten des Syntaxbaums dem für den entsprechenden Kindknoten ermittelten Teilgraph ein Start- und ein Endknoten sowie entsprechende gerichtete Kanten hinzugefügt. Der resultierende CM-task Graph wird an einem neu erzeugten einzelnen Graphknoten v annotiert. Analog zu den Blattknoten wird dem Syntaxbaumknoten k eine Knotenmenge

Algorithmus 4 : Konstruktion des CM-task Graph für ein Verbundmodul. An jedem Knoten k des zugehörigen abstrakten Syntaxbaums wird bottom-up die zugehörige Knotenmenge $k.V$ und die zugehörige Kantenmenge $k.E$ annotiert.

```
1  procedure construct_CMtask_graph (Syntaxbaumknoten k)
2  begin
3      foreach (Kindknoten c von k) do construct_CMtask_graph (c) ; // bottom-up
4      if (k ist Blattknoten) then
5          erzeuge neuen Graphknoten v, der auf k verweist;
6          k.V = {v}; k.E = ∅;
7      else if (k ist par-Knoten mit n Kindknoten c_1,...,c_n) then
8          k.V = ∪_{i=1,...,n} c_i.V; k.E = ∪_{i=1,...,n} c_i.E;
9      else if (k ist cpar-Knoten mit n Kindknoten c_1,...,c_n) then
10         k.V = ∪_{i=1,...,n} c_i.V; k.E = ∪_{i=1,...,n} c_i.E;
11         for (i = 1,...,n-1) do
12             wähle Graphknoten v_i ∈ c_i.V und v_{i+1} ∈ c_{i+1}.V;
13             füge bidirektionale Kante (v_i, v_{i+1}) zu k.E hinzu;
14         foreach (an k annotierte Kommunikationsabhängigkeit cdep) do
15             seien v_1,...,v_m die Graphknoten der m an cdep beteiligten Modulaufrufe;
16             for (j = 1,...,m-1) do
17                 füge bidirektionale Kante (v_j, v_{j+1}) zu k.E annotiert mit dem cdep
                       zugrundeliegenden Kommunikationsparameter hinzu;
18     else if (k ist seq-Knoten mit n Kindknoten c_1,...,c_n) then
19         k.V = ∪_{i=1,...,n} c_i.V; k.E = ∪_{i=1,...,n} c_i.E;
20         for (i = 1,...,n-1) do
21             foreach (Knoten v_i ∈ c_i.V ohne ausgehende gerichtete Kante in c_i.E) do
22                 foreach (Knoten v_{i+1} ∈ c_{i+1}.V ohne eingehende gerichtete
23                 Kante in c_{i+1}.E) do
24                     füge gerichtete Kante (v_i, v_{i+1}) zu k.E hinzu;
25         foreach (an k annotierte Datenabhängigkeit ddep) do
26             seien v_i und v_j die Graphknoten des Schreib- bzw. Lesezugriffes von ddep;
27             füge gerichtete Kante (v_i, v_j) annotiert mit der ddep zugrundeliegenden
                   Variablen zu k.E hinzu;
28     else if (k ist Wurzelknoten mit Kindknoten c) then
29         erzeuge neuen Startknoten v_s und neuen Endknoten v_e;
30         k.V = c.V ∪ {v_s, v_e}; k.E = c.E;
31         foreach (Knoten v ∈ c.V ohne eingehende gerichtete Kante in c.E) do
32             füge gerichtete Kante (v_s, v) zu k.E hinzu;
33         foreach (Knoten v ∈ c.V ohne ausgehende gerichtete Kante in c.E) do
34             füge gerichtete Kante (v, v_e) zu k.E hinzu;
35         füge je eine gerichtete Kanten für jede an k annotierte Datenabhängigkeit
               zu k.E hinzu;
36 end
```

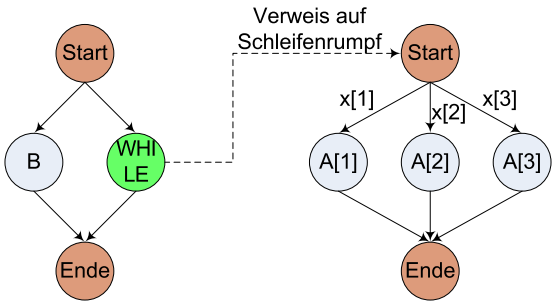


Abbildung 25: Konstruierte CM-task Graphen für das erklärende Beispiel aus Abbildung 24. Der linke CM-task Graph repräsentiert das Hauptmodul, wobei die `while`-Schleife durch einen Knoten repräsentiert wird, der auf den CM-task Graph des Schleifenrumpfes verweist. Im CM-task Graph des Schleifenrumpfes werden drei Knoten für die durch die `parfor`-Schleife definierten Instanzen des Modulaufrufs `A` eingefügt. Die Annotationen an den Kanten resultieren aus den im `prels`-Konstrukt des erweiterten Spezifikationsprogrammes definierten Datenabhängigkeiten.

bestehend aus diesem einzelnen Graphknoten (d.h. $k.V = \{v\}$) und einer leeren Kantenmenge (d.h. $k.E = \emptyset$) annotiert.

Beispiel 5.2.1 *Abbildung 24 zeigt die Hauptmoduldefinition des erweiterten Spezifikationsprogrammes und den zugehörigen abstrakten Syntaxbaum für das erklärende Beispiel der Schedulingphase. Die für dieses Beispiel erzeugten CM-task Graphen sind in Abbildung 25 dargestellt. Durch die bottom-up Konstruktionsvorschrift aus Algorithmus 4 wird zunächst der rechts dargestellte CM-task Graph des Schleifenrumpfes der `while`-Schleife und anschließend der links dargestellte CM-task Graph für das Hauptmodul erzeugt.*

5.2.2. Hierarchisches Scheduling der CM-task Anwendung

Das Scheduling eines gegebenen CM-task Graph erfordert Kostenwerte für die im Graph enthaltenen Knoten. Die Kostenwerte eines Knotens geben die Ausführungszeit des durch den Knoten repräsentierten Programmfragments abhängig von der Anzahl ausführender Prozessoren an. Die Ausführungszeit einer kompletten Schleife oder eines gesamten Verbundmoduls hängt davon ab, in welcher Reihenfolge und auf welchen Prozessorgruppen die im entsprechenden Schleifenrumpf bzw. im jeweiligen Verbundmodul definierten Modulaufrufe ausgeführt werden. Daher muss zur Kostenbestimmung eines einen Verbundmodulaufruf oder eine Schleife repräsentierenden Knotens eines CM-task Graphen der Schedule des entsprechenden Verbundmoduls bzw. des entsprechenden Schleifenrumpfes bekannt sein. Daraus ergibt sich ein hierarchisches Vorgehen, bei dem Kostenberechnung und Scheduling miteinander verschränkt sind.

Algorithmus 5 skizziert das hierarchische Scheduling für eine CM-task Anwendung. Die Prozedur `schedule_hierarchical` bestimmt die Kostenwerte für die im erweiterten Spezifikationsprogramm definierten Verbundmodule durch Scheduling des zugehörigen CM-task Graph des jeweiligen Verbundmoduls. Für jedes Verbundmodul (mit Ausnahme des Hauptmoduls) wird für jede Prozessoranzahl p, $p = 1,\ldots,P$, ein Schedule berechnet, wobei P die

Algorithmus 5 : Hierarchisches Scheduling einer CM-task Anwendung.

```
1  procedure schedule_hierarchical (verfügbare Prozessoranzahl P)
2  begin
3      foreach (Verbundmoduldefinition VM des erweiterten Spezifikationsprogrammes
               in der Reihenfolge der Definition) do
4          sei G = (V,E) der an VM annotierte CM-task Graph;
5          compute_costs(G, P);
6          if (VM ist Hauptmodul) then p_min = P; else p_min = 1;
7          for (p = p_min,...,P) do
8              S = schedule (G,p);
9              setze Kosten von VM für p Prozessoren auf T_max(S);
10 end
11 procedure compute_costs (CM-task Graph G = (V,E), Prozessorzahl P)
12 begin
13     foreach (Knoten v ∈ V) do
14         if (v repräsentiert Aufruf des Basismoduls BM) then
15             setze Kosten von v für p Prozessoren auf Ergebnis der Auswertung der
                   für BM spezifizierten symbolischen Laufzeitformel (p = 1,...,P);
16         else if (v repräsentiert Aufruf des Verbundmoduls VM) then
17             setze Kosten von v auf den an VM annotierten Kostenwert für p
                   (p = 1,...,P);
18         else if (v repräsentiert for- oder while-Schleife) then
19             sei G_1 = (V_1,E_1) der an v gespeicherte CM-task Graph des
                   Schleifenrumpfes;
20             compute_costs (G_1,P);
21             sei it die Anzahl der Iterationen der durch v repräsentierten Schleife;
22             for (p = 1,...,P) do
23                 S = schedule (G_1,p);
24                 setze Kosten von v für p Prozessoren auf T_max(S) * it;
25         else if (v repräsentiert Bedingung) then
26             seien G_1 = (V_1,E_1) und G_2 = (V_2,E_2) die CM-task Graphen der
                       Bedingungszweige;
27             compute_costs (G_1,P);   compute_costs (G_2,P);
28             for (p = 1,...,P) do
29                 S_1 = schedule (G_1,p);   S_2 = schedule (G_2,p);
30                 setze Kosten von v für p Prozessoren auf max{T_max(S_1),T_max(S_2)};
31 end
32 function schedule (CM-task Graph G = (V,E), Prozessoranzahl p)
33 begin
34     führe CM-task Schedulingalgorithmus mit CM-task Graph G und verfügbarer
           Prozessoranzahl p aus und liefere einen CM-task Schedule S mit Gesamt-
           ausführungszeit T_max(S) in Form eines Syntaxbaumfragmentes zurück;
35 end
```

Prozessoranzahl der Zielplattform ist. Die genaue Prozessoranzahl für ein Verbundmodul wird durch die dem entsprechenden Verbundmodulaufruf zugeordnete Prozessoranzahl festgelegt. Für das Hauptmodul wird nur ein Schedule auf allen verfügbaren Prozessoren berechnet, da die Spezifikationssprache keine Rekursion unterstützt und somit keine Aufrufe dieses Moduls vorhanden sind.

Die rekursive Prozedur `compute_costs` annotiert Kostenwerte an den Knoten eines gegebenen CM-task Graph G. Die Kosten eines Knotens v werden abhängig vom von v repräsentierten Konstrukt der Spezifikationssprache bestimmt. Für Schleifen wird die vom Anwendungsprogrammierer spezifizierte Iterationsanzahl berücksichtigt. Die Kosten einer Bedingung entsprechen dem Maximum aus den ermittelten Ausführungszeiten der Bedingungszweige. Damit wird gewährleistet, dass im berechneten Schedule unabhängig vom tatsächlich ausgeführten Bedingungszweig genügend Zeit zur Abarbeitung der Bedingung vorhanden ist.

Die Funktion `schedule` enthält den Schedulingalgorithmus für einen gegebenen CM-task Graph G mit annotierten Kostenwerten und eine gegebene Prozessoranzahl p. Die Ausgabe dieser Funktion ist ein CM-task Schedule S, der in Form eines Syntaxbaumfragments des Rahmenprogrammes ausgegeben wird. Dieses Syntaxbaumfragment enthält je einen Blattknoten für jeden Graphknoten von G (mit Ausnahme des Startknotens und des Endknotens). Jeder dieser Blattknoten besitzt eine Annotation für die zugeordnete Prozessorgruppe und verweist auf den zugehörigen Knoten im Syntaxbaum des erweiterten Spezifikationsprogrammes. Dieser Verweis wird aus den entsprechenden Knoten des CM-task Graph G übernommen.

Die inneren Knoten des erzeugten Syntaxbaumfragments sind mit `seq`, `par` oder `cpar` bezeichnet und legen die Ausführungsreihenfolge der durch die Blattknoten repräsentierten Programmteile fest. Entsprechend der Semantik der Konstruktoren des Rahmenprogrammes (vgl. Abschnitt 4.4) definieren diese Syntaxbaumknoten eine Nacheinanderausführung (`seq`), eine gleichzeitige Ausführung ohne Kommunikationsabhängigkeiten (`par`) bzw. eine gleichzeitige Ausführung mit Kommunikationsabhängigkeiten (`cpar`) der durch die Kindknoten repräsentierten Programmfragmente. Der CM-task Schedulingalgorithmus aus Kapitel 3 erzeugt Schedules mit einer Struktur, die sich direkt in ein entsprechendes Syntaxbaumfragment übersetzen lässt.

Algorithmus 5 berechnet für jedes Verbundmodul (außer dem Hauptmodul), jeden Schleifenrumpf und jeden Bedingungszweig P Schedules, wobei P die Prozessoranzahl der Zielplattform ist. Daraus kann eine hohe Anzahl berechneter Schedules für eine gegebene CM-task Anwendung resultieren. Die Programmlaufzeit dieses Arbeitsschrittes des CM-task Compilers ist für die in Kapitel 6 betrachteten Anwendungen auch bei einer hohen Prozessoranzahl P gering, da die erzeugten CM-task Graphen dieser Anwendungen aufgrund der hierarchischen Anordnung nur wenige Knoten besitzen und der genutzte Schedulingalgorithmus eine sehr geringe Laufzeit besitzt (vgl. Abschnitt 3.3).

Beispiel 5.2.2 *Für das hierarchische Scheduling des erklärenden Beispiels wird eine Zielplattform mit $P = 64$ Prozessoren verwendet. Durch die Prozedur `compute_costs` aus Algorithmus 5 werden zunächst 64 CM-task Schedules für den CM-task Graph des Schleifenrumpfes der `while`-Schleife berechnet (s. Abbildung 26). Aus den Gesamtausführungszeiten dieser Schedules werden die Kosten des `while`-Knotens im CM-task Graph des Hauptmoduls bestimmt. Anschließend erfolgt die Berechnung eines CM-task Schedules für den CM-task Graph des Hauptmoduls mit $p = 64$ Prozessoren. Der entsprechende CM-task Schedule mit dem zugehörigen Syntaxbaumfragment des Rahmenprogrammes ist in Abbildung 27 dargestellt.*

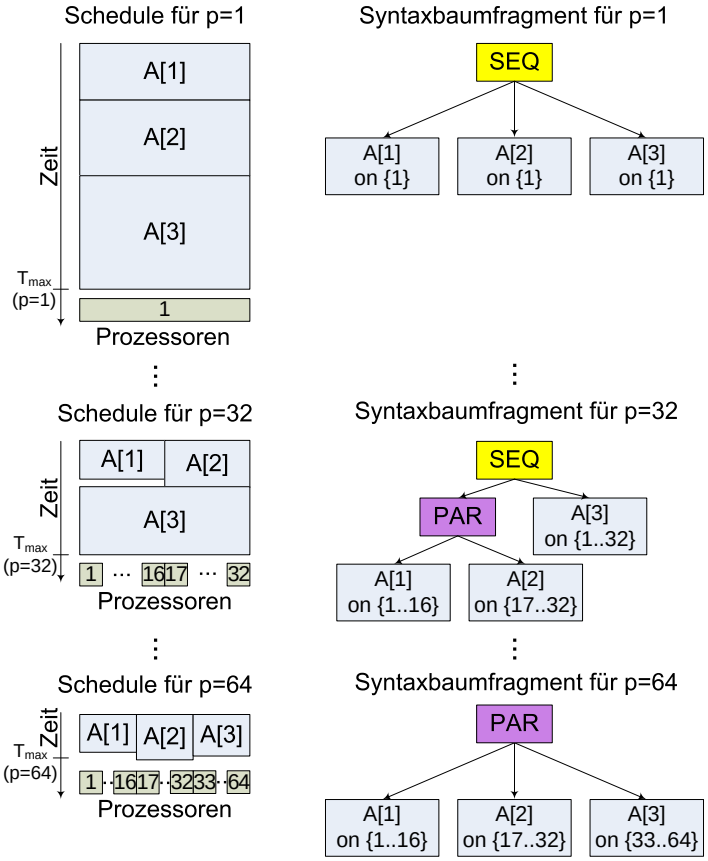


Abbildung 26: Illustration der erzeugten CM-task Schedules (links) und der zugehörigen Syntaxbaumfragmente des Rahmenprogrammes (rechts) für den CM-task Graph des Schleifenrumpfes der `while`-Schleife des erklärenden Beispiels (s. Abbildung 25 (rechts)). Für eine Zielplattform mit $P = 64$ Prozessoren werden insgesamt 64 CM-task Schedules erzeugt. Die Abbildung zeigt exemplarisch die Resultate für $p \in \{1, 32, 64\}$ Prozessoren. In den erzeugten Syntaxbaumfragmenten werden die verwendeten Prozessorgruppen in den Blattknoten gespeichert. Die inneren Knoten sind `par`-, `seq`- und `cpar`-Knoten, die die Ausführungsreihenfolge der Blattknoten definieren.

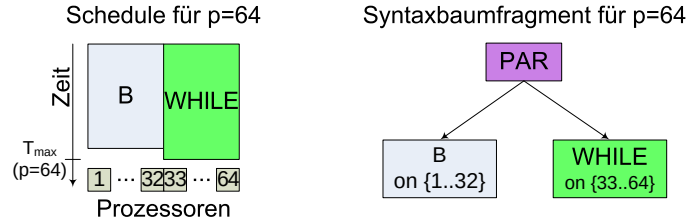


Abbildung 27: Illustration des erzeugten CM-task Schedules (links) und des zugehörigen Syntaxbaumfragments des Rahmenprogrammes (rechts) für den CM-task Graph des Hauptmoduls des erklärenden Beispiels (s. Abbildung 25 (links)).

Algorithmus 6 : Konstruktion der Syntaxbäume des Rahmenprogrammes

```
1  begin
2      initialisiere X = {(HM,P)} mit Hauptmodul HM des erweiterten Spezifikations-
           programmes und Prozessoranzahl P der Zielplattform;
3      while (X ≠ ∅) do
4          wähle Paar (VM,p) ∈ X;
5          X = X \ (VM,p);
6          erzeuge Wurzelknoten w des Syntaxbaums des Rahmenprogrammes für VM;
7          annotiere Prozessorgruppe {1..p} an w;
8          setze Kindknoten von w auf Syntaxbaumfragment für VM mit p Prozessoren;
9          starte Tiefensuche im Syntaxbaum des Rahmenprogrammes ausgehend von w;
10         foreach (in der Tiefensuche angetroffenen Syntaxbaumknoten k) do
11             if (k repräsentiert Aufruf des Verbundmoduls VM_1) then
12                 sei q die Größe der an k annotierten Prozessorgruppe;
13                 if (VM_1 hat keine Markierung für q Prozessoren) then
14                     füge Markierung für q Prozessoren zu VM_1 hinzu;
15                     X = X ∪ {(VM_1,q)};
16             else if (k symbolisiert eine Schleife oder Bedingung) then
17                 sei q die Größe der an k annotierten Prozessorgruppe;
18                 setze Kindknoten von k auf das für q Prozessoren erzeugte Syntaxbaum-
                       fragment des entsprechenden Schleifenrumpfes bzw. der
                       entsprechenden Bedingungszweige;
19                 setze Tiefensuche mit diesen Kindknoten von k fort;
20         füge w zur Menge erzeugter Syntaxbäume des Rahmenprogrammes hinzu;
21 end
```

5.2.3. Konstruktion der Syntaxbäume des Rahmenprogrammes

Dieser Schritt kombiniert die vom Schedulingalgorithmus erzeugten Syntaxbaumfragmente zu einer Menge von Syntaxbäumen, die die Verbundmoduldefinitionen des Rahmenprogrammes darstellen. Die erzeugten Syntaxbäume definieren die Ausführungsreihenfolge und die ausführenden Prozessorgruppen für die Modulaufrufe, `for`- und `while`-Schleifen und die Bedingungen des Rahmenprogrammes, enthalten aber noch nicht alle für die Erzeugung des Rahmenprogrammes benötigten Informationen wie z.B. die Parameterlisten von Modulaufrufen oder die Annotationen für Daten- und Kommunikationsabhängigkeiten. Die fehlenden Informationen werden sukzessive in den nachfolgenden Arbeitsschritten hinzugefügt.

Algorithmus 6 skizziert das Vorgehen dieses Arbeitsschrittes. Der Algorithmus verwendet eine Menge X, die die noch zu betrachtenden Paare (VM, p) bestehend aus einem Verbundmodul VM des erweiterten Spezifikationsprogrammes und die VM ausführende Prozessoranzahl p speichert. In jedem Schritt wählt der Algorithmus ein Paar (VM, p) aus und konstruiert den abstrakten Syntaxbaum für VM im Rahmenprogramm. Dazu wird zunächst die entsprechende Syntaxbaumwurzel erzeugt und das vom Schedulingalgorithmus für den CM-task Graph für VM mit p Prozessoren erzeugte Syntaxbaumfragment als Kindknoten eingesetzt. Durch

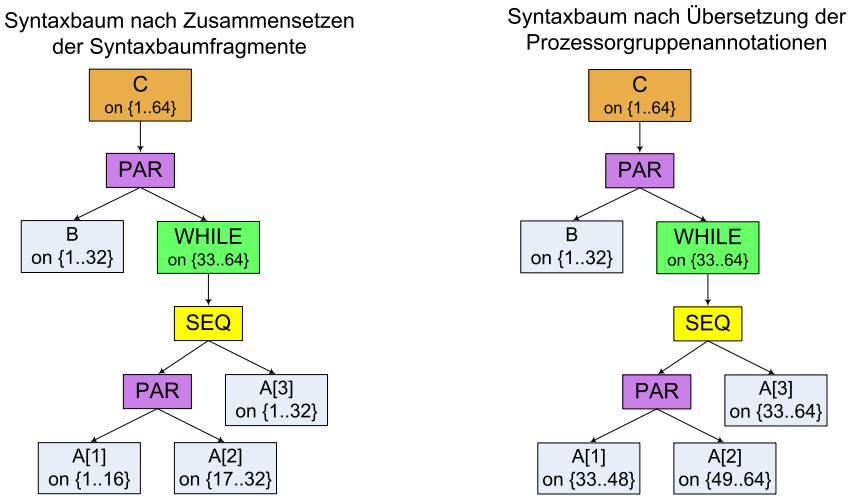


Abbildung 28: Erzeugter Syntaxbaum des Rahmenprogrammes für das Hauptmodul des erklärenden Beispiels (links) und Syntaxbaum des erklärenden Beispiels nach Übersetzung der Prozessorgruppenannotation (rechts).

einen Tiefensuchlauf werden sukzessive die entsprechenden Syntaxbaumfragmente für die Schleifenrümpfe und Bedingungszweige dieses Syntaxbaumfragments eingefügt.

Für im zu konstruierenden Syntaxbaum auftretende Verbundmodulaufrufe wird gegebenfalls ein neues Paar bestehend aus dem aufgerufenen Verbundmodul VM_1 und der für den Aufruf verwendeten Prozessoranzahl q zur Menge X hinzugefügt. Durch eine Markierung der Verbundmodule wird sichergestellt, dass jedes derartige Paar maximal einmal zu X hinzugefügt wird. Ein bestimmtes Verbundmodul kann jedoch mit verschiedenen Prozessorzahlen zu X hinzugefügt werden, so dass aus einer Verbundmoduldefinition des erweiterten Spezifikationsprogrammes mehrere Verbundmoduldefinitionen im Rahmenprogramm entstehen. Dieser Fall tritt ein, wenn für ein Verbundmodul VM Aufrufe mit unterschiedlicher vom Schedulingalgorithmus festgelegter Prozessoranzahl existieren. Im ausgegebenen Rahmenprogramm werden die verschiedenen Versionen von VM anhand eines eindeutigen Namenssuffixes unterschieden.

Beispiel 5.2.3 *Abbildung 28 (links) zeigt den konstruierten Syntaxbaum für das Hauptmodul des erklärenden Beispiels. Dieser Syntaxbaum entsteht aus dem neu erzeugten Wurzelknoten mit annotierter Prozessorgruppe* $\{1..64\}$ *und der im hierarchischen Schedulingschritt erzeugten Syntaxbaumfragmente für den CM-task Graph des Hauptmoduls mit* $p = 64$ *Prozessoren (s. Abbildung 28 (rechts)) und für den Rumpf der* `while`*-Schleife mit* $p = 32$ *Prozessoren (s. Abbildung 27 (mitte rechts)).*

5.2.4. Übersetzung der Prozessorgruppenannotationen

Die im vorherigen Schritt erzeugten Syntaxbäume des Rahmenprogrammes bestehen jeweils aus einer Menge von Syntaxbaumfragmenten, wobei jedes Syntaxbaumfragment den Schedule eines Programmblockes wie bspw. eines Schleifenrumpfes repräsentiert. Die in den Syntaxbaumfragmenten eines Syntaxbaums annotierten Prozessorgruppen verwenden lokale Prozessornummern innerhalb des jeweiligen Syntaxbaumfragments. Beispielsweise sind in einem für q Prozessoren

erzeugten Syntaxbaumfragment alle annotierten Prozessorgruppen Teilmengen von $\{1,\ldots,q\}$. In diesem Schritt werden alle in einem gegebenen abstrakten Syntaxbaum annotierten Prozessornummern in globale Prozessornummern des jeweiligen Verbundmoduls übersetzt. Damit wird eine bessere Verständlichkeit des ausgegebenen Rahmenprogrammes angestrebt, da nach diesem Schritt für jeden Modulaufruf die ausführenden Prozessoren direkt ersichtlich sind.

Die Realisierung dieses Arbeitsschrittes erfolgt durch eine top-down Traversierung eines gegebenen abstrakten Syntaxbaums. Die neue Prozessorgruppenannotation eines Knotens k wird bei der Traversierung aus der zuvor an k annotierten Prozessorgruppe und der neuen Prozessorgruppe des Syntaxbaumknotens bestimmt, an dem das k enthaltende Syntaxbaumfragment im vorherigen Schritt eingehängt wurde, also dem letzten `for`-, `while`-, `if`- oder Wurzelknoten auf dem Pfad von der Syntaxbaumwurzel zu k.

Beispiel 5.2.4 *Abbildung 28 (rechts) zeigt den abstrakten Syntaxbaum des erklärenden Beispiels nach Übersetzung der Prozessorgruppenannotationen. Für den Modulaufruf* `B` *und den* `while`*-Knoten entspricht die neue Prozessorgruppenannotation der jeweiligen alten Annotation. Für die im Teilbaum unterhalb des* `while`*-Knotens annotierten Prozessornummern erfolgt eine Abbildung von der lokalen Prozessormenge* $\{1,\ldots,32\}$ *auf die globale Prozessormenge* $\{33,\ldots,64\}$*, die sich aus der am* `while`*-Knoten annotierten Prozessorgruppe ergibt.*

5.2.5. Schleifenrekonstruktion

Das Ziel dieses Arbeitsschrittes ist die Erhöhung der Lesbarkeit des Rahmenprogrammes, indem `parfor`- und `cparfor`-Schleifen des erweiterten Spezifikationsprogrammes im Rahmenprogramm rekonstruiert werden. Eine derartige Rekonstruktion wird ausgeführt, wenn für die Iterationen einer `parfor`- oder `cparfor`-Schleife des erweiterten Spezifikationsprogrammes durch entsprechende `par`- bzw. `cpar`-Knoten im erzeugten Syntaxbaum des Rahmenprogrammes eine zeitgleiche Ausführung festgelegt ist. In diesem Arbeitsschritt werden die bereits erzeugten Syntaxbäume des Rahmenprogrammes transformiert. Dabei können sowohl zusätzliche `parfor`- und `cparfor`-Knoten eingefügt werden, als auch bestehende `par`- und `cpar`-Knoten durch `parfor`- bzw. `cparfor`-Knoten ersetzt werden.

Da eine `parfor`-Schleifen im erweiterten Spezifikationsprogramm die Unabhängigkeit der enthaltenen Schleifeniterationen definiert, können durch den Schedulingalgorithmus verschiedene Ausführungsreihenfolgen für die Iterationen einer derartigen Schleife festgelegt werden. Beispielsweise kann ein Teil der Iterationen zeitgleich ausgeführt werden, wohingegen andere Iterationen nacheinander ausgeführt werden. In diesem Fall wird für jede zeitgleich ausgeführte Teilmenge von Iterationen eine eigene Kopie der ursprünglichen `parfor`-Schleife mit einem entsprechend eingeschränkten Iterationsraum eingefügt. Dadurch können aus einer `parfor`-Schleife des erweiterten Spezifikationsprogrammes mehrere `parfor`-Schleifen im Rahmenprogramm entstehen, wobei die Schleifen des Rahmenprogrammes paarweise disjunkte Teilmengen der Iterationen der ursprünglichen Schleife enthalten.

Algorithmus 7 skizziert die Schleifenrekonstruktion für ein gegebenes Verbundmodul des Rahmenprogrammes. In jedem Schritt des Algorithmus wird eine der Schleifen des erweiterten Spezifikationsprogrammes betrachtet. Die Reihenfolge der Betrachtung ist derart gewählt, dass die Rekonstruktion innerer Schleifen vor der Rekonstruktion äußerer Schleifen ausgeführt wird. Dadurch ist auch die Rekonstruktion ineinander verschachtelter Schleifen möglich.

Für eine bestimmte zu rekonstruierende Schleife l werden durch einen Tiefensuchlauf über

Algorithmus 7 : Rekonstruktion der `parfor`- und `cparfor`-Schleifen.

```
1  begin
2      sei W der abstrakte Syntaxbaum eines Verbundmoduls VM im Rahmenprogramm;
3      sei W' der abstrakte Syntaxbaum von VM im erweiterten Spezifikationsprogramm;
4      bestimme Liste L aller in W' enthaltener parfor- und cparfor-Knoten durch
           eine postorder Traversierung von W';
5      foreach (parfor- oder cparfor-Knoten l ∈ L in Reihenfolge der Liste) do
6          starte Tiefensuche in W;
7          foreach (in der Tiefensuche angetroffenen par- bzw. cpar-Knoten k) do
8              bilde Liste Iter mit Kindknoten von k, die auf Kindknoten von l verweisen;
9              if (Iter enthält mindestens zwei Elemente, die zu direkt aufeinanderfolgen-
                   den Iterationen von l gehören) then
10                 erzeuge neuen parfor- bzw. cparfor-Knoten l' mit Verweis auf l;
11                 bestimme neuen Iterationsbereich für l';
12                 if (es gibt schon eine rekonstruierte Schleife für l) then
13                     füge Kopieannotation zu l' hinzu;
14                 bestimme Liste PG aller Prozessorgruppen, die in den Teilbäumen aus
                       Iter annotiert sind;
15                 setze Schleifenrumpf von l' auf erstes Element von Iter;
16                 setze Prozessorgruppenannotation des einzigen Modulaufrufs im
                       neuen Schleifenrumpf von l' auf PG;
17                 if (Iter beinhaltet alle Kindknoten von k) then
18                     ersetze k durch l' im abstrakten Syntaxbaum W;
19                 else
20                     entferne alle Kindknoten von k, die in Iter enthalten sind;
21                     füge l' als Kindknoten zu k hinzu;
22 end
```

den abstrakten Syntaxbaum des Rahmenprogrammes direkt aufeinanderfolgende Schleifeniterationen von l gesucht, für die durch einen `par`- oder `cpar`-Knoten eine zeitgleiche Ausführung festgelegt ist. Zum Auffinden der Schleifeniterationen von l im Syntaxbaum des Rahmenprogrammes werden die in den entsprechenden Syntaxbaumknoten gespeicherten Verweise auf die zugehörigen Knoten im Syntaxbaum des erweiterten Spezifikationsprogramm genutzt.

Dadurch werden nur diejenigen Schleifen rekonstruiert, deren Schleifenrumpf entweder aus einem einzelnen Modulaufruf oder aus einer anderen bereits rekonstruierten `parfor`- oder `cparfor`-Schleife besteht. Dies bedeutet, dass jede rekonstruierte Schleife genau einen Modulaufruf enthält. Schleifen des erweiterten Spezifikationsprogrammes mit enthaltenen `par`, `seq`- oder `cpar`-Konstrukten werden durch Algorithmus 7 nicht rekonstruiert, d.h. diese Schleifen sind im Rahmenprogramm unabhängig von der vom Schedulingalgorithmus festgelegten Ausführungsreihenfolge der Iterationen in entrollter Form enthalten.

Werden für die Schleife l des erweiterten Spezifikationsprogrammes mehrere Schleifen in das Rahmenprogramm eingefügt, wird den eingefügten Schleifen eine entsprechende Kopiean-

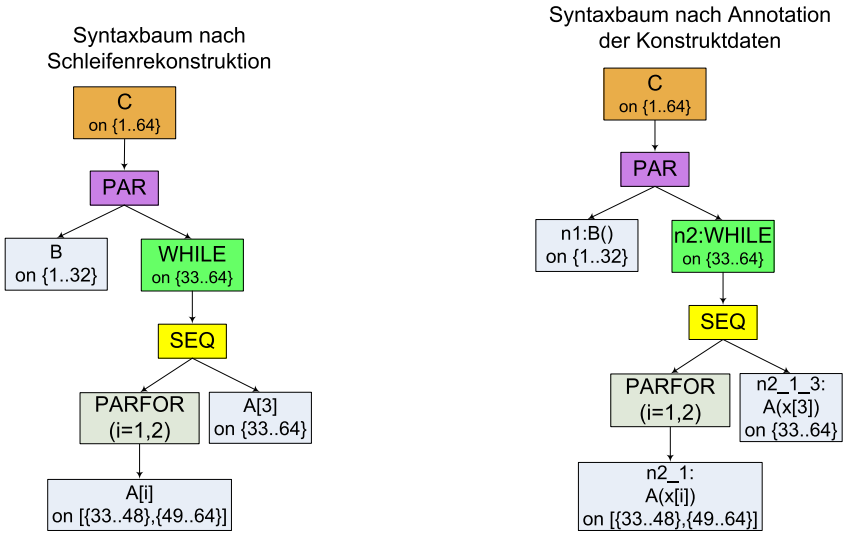


Abbildung 29: Links: Abstrakter Syntaxbaum des Rahmenprogrammes für das Hauptmoduls des erklärenden Beispiels nach der Schleifenrekonstruktion. Dieser Schritt ersetzt den `par`-Knoten, der eine zeitgleiche Ausführung für zwei Iterationen (`A[1]` und `A[2]`) der `parfor`-Schleife des erweiterten Spezifikationsprogrammes definiert, durch einen entsprechenden `parfor`-Knoten. Rechts: Abstrakter Syntaxbaum des Hauptmoduls des erklärenden Beispiels nach der Annotation der Konstruktdaten. Die annotierten Daten umfassen die Parameterlisten und Identifikatoren der Modulaufrufe und der `while`-Schleife. Am Modulaufruf `A(x[3])` wird ein gegenüber dem erweiterten Spezifikationsprogramm veränderter Identifikator (`n2_1_3`) annotiert, da dieser Aufruf nicht mehr Bestandteil der ursprünglichen `parfor`-Schleife ist.

notation hinzugefügt. Diese Annotation wird im nächsten Arbeitsschritt zur Bestimmung der Identifikatoren verwendet und nicht im Rahmenprogramm ausgegeben.

An dem in der rekonstruierten Schleife l' befindenden Modulaufruf wird eine Liste von Prozessorgruppen *PG* annotiert. Diese Liste gibt die ausführenden Prozessoren für jede durch l' definierte Instanz des Modulaufrufs an.

Beispiel 5.2.5 *Das erweiterte Spezifikationsprogramm des erklärenden Beispiels enthält eine `parfor`-Schleife (vgl. Abbildung 24). Zur Rekonstruktion dieser Schleife führt Algorithmus 7 einen Tiefensuchlauf über den abstrakten Syntaxbaum des Rahmenprogrammes durch. Bei diesem Tiefensuchlauf wird festgestellt, dass für die Iterationen `A[1]` und `A[2]` der `parfor`-Schleife des erweiterten Spezifikationsprogrammes eine zeitgleich Ausführung durch einen `par`-Knoten festgelegt wird. Daher wird für diese Iterationen eine neue `parfor`-Schleife mit zwei Iterationen erzeugt und der zugehörige `parfor`-Knoten anstelle des `par`-Knotens in den abstrakten Syntaxbaum eingefügt. Als Kindknoten des neu erzeugten `parfor`-Knotens wird der Modulaufruf `A[1]` verwendet, dem zusätzlich die Prozessorgruppenannotation des Modulaufrufs `A[2]` hinzugefügt wird. Abbildung 29 (links) zeigt das Ergebnis der Schleifenrekonstruktion für das erklärende Beispiel.*

5.2.6. Annotation der Konstruktdaten

In diesem Arbeitsschritt werden die in den Syntaxbäumen des erweiterten Spezifikationsprogrammes gespeicherten Informationen in die entsprechenden Syntaxbäume des Rahmenprogrammes übertragen. Diese Informationen umfassen die in der Analysephase festgelegten Identifikatoren sowie die vom Anwendungsprogrammierer spezifizierten Parameterlisten von Basis- und Verbundmodulaufrufen, Iterationsgrenzen von `for`-Schleifen und logischen Ausdrücke von `while`- und `if`-Konstrukten.

Da sich die Programmstruktur des Rahmenprogrammes von der des erweiterten Spezifikationsprogrammes unterscheidet, sind Anpassungen an den übertragenen Informationen erforderlich. Diese Anpassungen betreffen die Identifikatoren, für die die Eindeutigkeit im Rahmenprogramm sichergestellt wird, und die Parameterlisten und Schleifengrenzen, in denen die Iterationsvariable einer Schleife des erweiterten Spezifikationsprogrammes auftritt, die im vorherigen Schritt nicht rekonstruiert werden konnte. Verweise auf derartige Iterationsvariablen werden in diesem Schritt durch entsprechende Konstanten ersetzt.

Konkret werden für ein gegebenes Verbundmodul des Rahmenprogrammes die folgenden drei Schritte nacheinander ausgeführt.

(i) **Bestimmung der umgebenden Schleifen im Rahmenprogramm:**
Durch einen Tiefensuchlauf über den abstrakten Syntaxbaum des gegebenen Verbundmoduls wird an jedem Knoten k des Syntaxbaums eine Liste bestehend aus allen `parfor`- und `cparfor`-Knoten auf dem Pfad von der Syntaxbaumwurzel zu k gespeichert.

(ii) **Bestimmung der umgebenden Schleifen im erweiterten Spezifikationsprogramm:**
Analog zu Schritt (i) wird durch einen Tiefensuchlauf eine Liste der umgebenden Schleifen an jedem Knoten des abstrakten Syntaxbaums des entsprechenden Verbundmoduls im erweiterten Spezifikationsprogramm annotiert.

(iii) **Übertragung und Anpassung der Daten:**
Dieser Schritt ist durch einen Tiefensuchlauf über den gegebenen abstrakten Syntaxbaum eines Verbundmoduls im Rahmenprogrammes realisiert. Dabei werden für die `for`-, `while`-, `if`- und Blattknoten zwei nachfolgend beschriebene Teilschritte ausgeführt. Die übrigen Syntaxbaumknoten besitzen entweder keine zugeordneten Knoten im Syntaxbaum des erweiterten Spezifikationsprogrammes (`seq`, `par` und `cpar`) oder die Informationen wurden bereits bei der Schleifenrekonstruktion übertragen (`parfor` und `cparfor`).

Im Folgenden wird ein Knoten k des abstrakten Syntaxbaums des Rahmenprogrammes betrachtet. Der zu k korrespondierende Knoten k' im abstrakten Syntaxbaum des erweiterten Spezifikationsprogrammes wird durch den in k gespeicherten Verweis (vgl. Abschnitt 5.2.2) bestimmt. Im ersten Teilschritt wird ein Identifikator für k derart bestimmt, dass nach Möglichkeit die Identifikatoren des erweiterten Spezifikationsprogrammes erhalten bleiben, aber kein Identifikator mehrmals im Rahmenprogramm auftritt.

Dazu werden die in Schritt (i) und Schritt (ii) bestimmten Schleifenumgebungen von k bzw. k' betrachtet. Besitzen k und k' eine identische Schleifenumgebung, d.h. die Anzahl der umgebenden Schleifen ist gleich und keine der k umgebenden Schleifen besitzt eine Kopieannotation, wird k der Identifikator von k' annotiert. Andernfalls wird ein neuer

Identifikator für k erzeugt, indem dem Identifikator von k' die konkreten an k gespeicherten Iterationen der nicht rekonstruierten Schleifen angehängt werden. Die Zuordnung von Identifikatoren des erweiterten Spezifikationsprogrammes zu Identifikatoren des Rahmenprogrammes wird in einer Tabelle gespeichert, die im nachfolgend zur Annotation der Daten- und Kommunikationsabhängigkeiten verwendet wird.

Der zweite für k ausgeführte Teilschritt überträgt die Parameterliste (Blattknoten), die zugeordnete Bedingung (`while`- und `if`-Knoten), die zugeordnete Iterationsanzahl (`while`-Knoten) oder die zugeordneten Iterationsgrenzen (`for`-Knoten) von k' zu k. Eine Anpassung dieser Daten ist erforderlich, wenn sich die Anzahl umgebender Schleifen von k und k' unterscheiden, d.h. nicht alle k' umgebenden Schleifen rekonstruiert wurden. Anhand der an k und k' annotierten Liste der umgebenden Schleifen werden die nicht rekonstruierten Schleifen identifiziert und alle Verwendungen der zugehörigen Iterationsvariable dieser Schleifen durch entsprechende in k gespeicherten Konstanten ersetzt.

Beispiel 5.2.6 *Abbildung 29 (rechts) zeigt den abstrakten Syntaxbaum des erklärenden Beispiels nach Annotation der Konstruktdaten. Die übertragenen Informationen umfassen die Parameterlisten der Modulaufrufe sowie die Identifikatoren der Modulaufrufe und der* `while`*-Schleife. Für den Modulaufruf* `A(x[3])` *wird eine vom erweiterten Spezifikationsprogramm abweichende Schleifenumgebung festgestellt. Daher wird ein neuer Identifikator für diesen Knoten erzeugt (*`n2_1_3`*) und die Iterationsvariable* `i` *in der Parameterliste durch die entsprechende Konstante (*`3`*) ersetzt.*

5.2.7. Annotation der Daten- und Kommunikationsabhängigkeiten

In diesem Arbeitsschritt werden die in der Analysephase des CM-task Compilers bestimmten Daten- und Kommunikationsabhängigkeiten vom erweiterten Spezifikationsprogramm in das Rahmenprogramm übertragen. Die Realisierung erfolgt für einen gegebenen abstrakten Syntaxbaum W des Rahmenprogrammes durch die folgenden drei Teilschritte.

(i) **Bestimmung der zu annotierenden Daten- und Kommunikationsabhängigkeiten:**
Durch einen Tiefensuchlauf über den zu W gehörigen abstrakten Syntaxbaum des erweiterten Spezifikationsprogrammes werden alle in der Analysephase des CM-task Compilers annotierten Daten- und Kommunikationsabhängigkeiten des von W repräsentierten Verbundmoduls bestimmt.

(ii) **Anpassung der Identifikatoren:**
In diesem Teilschritt werden die in den in Schritt (i) bestimmten Daten- und Kommunikationsabhängigkeiten gespeicherten Identifikatoren des erweiterten Spezifikationsprogrammes in die entsprechenden Identifikatoren des Rahmenprogrammes übersetzt. Identifikatoren werden für den Schreibe- und Lesezugriff (Datenabhängigkeiten) bzw. für die beteiligte Basismodulaufrufe (Kommunikationsabhängigkeiten) gespeichert. Dieser Schritt ist mit Hilfe einer Tabelle realisiert, die bei der Annotation der Konstruktdaten aufgebaut wird und zu jedem Identifikator des erweiterten Spezifikationsprogrammes den entsprechenden Identifikator des Rahmenprogrammes speichert.

(iii) **Annotation der Daten- und Kommunikationsabhängigkeiten:**
Die Annotation einer Daten- oder Kommunikationsabhängigkeit erfolgt an der Wurzel

des kleinsten Teilbaumes von W, der alle an der entsprechenden Abhängigkeit beteiligten Konstrukte enthält. Die Realisierung erfolgt durch einen Tiefensuchlauf über W, wobei für einen Syntaxbaumknoten k wie folgt vorgegangen wird. Zunächst wird die Tiefensuche mit den Kindknoten von k fortgesetzt, bei der die Menge $IDS(k)$ der im Teilbaum des Syntaxbaums mit Wurzel k annotierten Identifikatoren bestimmt wird. Anschließend werden an k diejenigen Daten- und Kommunikationsabhängigkeiten annotiert, die (a) an keinem zuvor betrachteten Syntaxbaumknoten annotiert sind und für die (b) die Menge der gespeicherten Identifikatoren eine Teilmenge von $IDS(k)$ ist.

Nach diesem Arbeitsschritt sind alle Informationen aus dem erweiterten Spezifikationsprogramm in den erzeugten Syntaxbäumen des Rahmenprogrammes enthalten. Die Ausgabe des Rahmenprogrammes erfolgt durch einen top-down Lauf über den entsprechenden abstrakten Syntaxbaum.

Beispiel 5.2.7 *In Teilschritt (i) werden für das erklärende Beispiel durch einen Tiefensuchlauf über den abstrakten Syntaxbaum des erweiterten Spezifikationsprogrammes die folgenden drei zu annotierenden Datenabhängigkeiten ermittelt (vgl. Abbildung 24):*

```
(n2->n2_1[1],{x[1]}), (n2->n2_1[2],{x[2]}), (n2->n2_1[3],{x[3]})
```

In Teilschritt (ii) wird der Identifikator `n2_1[3]`*, der den Modulaufruf* `A(x[3])` *im erweiterten Spezifikationsprogramm bezeichnet, in den entsprechenden Identifikator* `n2_1_3` *des Rahmenprogrammes übersetzt, d.h. es entstehen die folgenden drei zu annotierende Datenabhängigkeiten:*

```
(n2->n2_1[1],{x[1]}), (n2->n2_1[2],{x[2]}), (n2->n2_1_3,{x[3]})
```

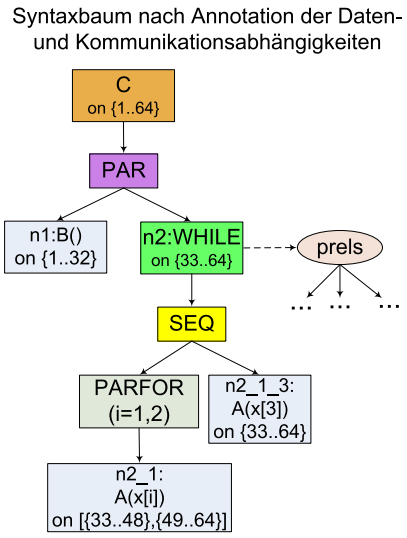
In Teilschritt (iii) erfolgt ein Tiefensuchlauf über den abstrakten Syntaxbaum des Rahmenprogrammes, bei dem alle drei Datenabhängigkeiten am `while`*-Knoten annotiert werden. Abbildung 30 zeigt den resultierenden abstrakten Syntaxbaum und die daraus erzeugte Hauptmoduldefinition des Rahmenprogrammes.*

5.3. Datenverteilungsphase des CM-task Compilers

Die Datenverteilungsphase gliedert sich in die folgenden drei Arbeitsschritte, die für jedes im Rahmenprogramm definierte Verbundmodul nacheinander ausgeführt werden.

(i) **Bestimmung der Datenverteilungen für Schleifen und Bedingungen.**
Dieser Arbeitsschritt legt für jeden virtuellen Eingabeparameter einer Schleife (`for` oder `while`) oder Bedingung des betrachteten Verbundmoduls eine *Eingabedatenverteilung* fest, die angibt in welcher Datenverteilung und auf welcher Prozessorgruppe der jeweilige Eingabeparameter von der entsprechenden Schleife oder Bedingung erwartet wird. Für die virtuellen Ausgabeparameter von `for`-, `while`- und `if`-Konstrukten wird eine *Ausgabedatenverteilung* festgelegt, die angibt, in welcher Datenverteilung und auf welcher Prozessorgruppe die Ausgabeparameter vom entsprechenden Konstrukt produziert werden.

(ii) **Einfügen der Datenumverteilungsoperationen.**
Dieser Arbeitsschritt bestimmt die für eine korrekte Abarbeitung des zu erzeugenden




```
cmmain C() {
 var x : vektor;
 par {
  n1:B() on {1..32};
  n2:while([...])#5 on {33..64} {
   seq {
    parfor (i=1:2) {
     n2_1:A(x[i]) on
       [{33..48},{49..64}];
    }
    n2_1_3:A(x[3]) on {33..64};
   }
   prels {
    (n2->n2_1[1],{x[1]}),
    (n2->n2_1[2],{x[2]}),
    (n2->n2_1_3,{x[3]})
}}}}
```

Abbildung 30: Links: Abstrakter Syntaxbaum des Rahmenprogrammes des erklärenden Beispiels nach Annotation der Daten und Kommunikationsabhängigkeiten. Am `while`-Knoten wird ein `prels`-Konstrukt annotiert, das drei Datenabhängigkeiten enthält, die durch drei abgehende Pfeile illustriert sind.
Rechts: Aus dem abstrakten Syntaxbaum des erklärenden Beispiels (links) erzeugte Hauptmoduldefinition des Rahmenprogrammes.

Koordinationsprogrammes benötigten Datenumverteilungsoperationen. Dazu werden die in der Analysephase des CM-task Compilers eingefügten Datenabhängigkeiten, die in der Schedulingphase des CM-task Compilers festgelegte Ausführungsreihenfolge und ausführenden Prozessorgruppen der Modulaufrufe, die in Schritt (i) festgelegten Datenverteilungen für Schleifen und Bedingungen sowie die vom Anwendungsentwickler spezifizierten Datenverteilungen für Basis- und Verbundmodule verwendet.

(iii) **Annotation von Lastausgleichsoperationen.**
Im semi-dynamischen Compileransatz werden zusätzlich die Programmpunkte festgelegt, an denen zur Laufzeit eine Lastausgleichsoperation durchgeführt werden soll.

Im Folgenden werden diese drei Arbeitsschritte beschrieben.

5.3.1. Bestimmung der Datenverteilungen für Schleifen und Bedingungen

Die Eingabe- und Ausgabedatenverteilungen von Schleifen und Bedingungen werden mit dem Ziel festgelegt, die Anzahl der benötigten Datenumverteilungsoperationen im nächsten Schritt zu reduzieren. Im Folgenden werden die für Schleifen und Bedingungen verwendeten Regeln erläutert und im Anschluss die Realisierung im CM-task Compiler vorgestellt.

Bedingungen. Die Eingabedatenverteilung eines virtuellen Eingabeparameters a einer Bedingung besteht aus der Datenverteilung und der Prozessorgruppe des ersten Lesezugriffs auf a im if-Zweig der jeweiligen Bedingung. Existiert kein Lesezugriff auf a im if-Zweig,

werden Datenverteilung und Prozessorgruppe des ersten Lesezugriffs auf *a* im else-Zweig der Bedingung verwendet. Durch diese Festlegung kann eine Datenumverteilungsoperation von der Eingabe des `if`-Konstrukts zum entsprechenden Lesezugriff vermieden werden.

Analog wird die Ausgabedatenverteilung eines virtuellen Ausgabeparameters einer Bedingung durch den letzten Schreibzugriff im if-Zweig bzw. (falls dieser nicht existiert) im else-Zeig festgelegt, so dass eine Datenumverteilung von diesem Schreibzugriff zur Ausgabe des `if`-Konstrukts vermieden werden kann. Durch diese Festlegungen können für einen Parameter, der sowohl virtueller Eingabe- als auch virtueller Ausgabeparameter des `if`-Konstrukts ist, unterschiedliche Eingabe- und Ausgabedatenverteilungen festgelegt werden.

Schleifen. Die Eingabedatenverteilungen für die virtuellen Eingabeparameter und die Ausgabedatenverteilungen für die virtuellen Ausgabeparameter einer `for`- oder `while`-Schleife werden analog zu den Bedingungen festgelegt, d.h. die entsprechenden Verteilungen verwenden die Datenverteilung und die Prozessorgruppe des ersten Lesezugriffs bzw. des letzten Schreibzugriffs im Schleifenrumpf.

Für Parameter, die sowohl virtueller Eingabe- als auch virtueller Ausgabeparameter einer Schleife sind, werden von den Bedingungen abweichende Regeln verwendet. Für diese Parameter wird sowohl die Eingabedatenverteilung als auch die Ausgabedatenverteilung durch die Datenverteilung und die Prozessorgruppe der letzten Schreibzugriffs im entsprechenden Schleifenrumpf bestimmt. Dadurch sind Eingabedatenverteilung und Ausgabedatenverteilung dieser Parameter identisch und zusätzliche Datenumverteilungsoperationen zwischen den Iterationen der jeweiligen Schleife werden vermieden.

Realisierung im CM-task Compiler. Die Realisierung dieses Arbeitsschrittes der Datenverteilungsphase erfolgt im CM-task Compiler durch einen Tiefensuchlauf über den gegebenen abstrakten Syntaxbaum eines Verbundmoduls im Rahmenprogramm. An den Knoten dieses Syntaxbaums sind die in der Analysephase des CM-task Compilers bestimmten Datenabhängigkeiten und die in der Schedulingphase festgelegten Prozessorgruppen annotiert. An den Blattknoten sind zusätzlich die Datenverteilungen der aktuellen Parameter des zugehörigen Modulaufrufs entsprechend der vom Anwendungsentwickler definierten Schnittstellenbeschreibung des aufgerufenen Moduls annotiert.

Bei dieser Tiefensuche wird für einen bestimmten `for`- oder `while`-Knoten *k* wie folgt vorgegangen. Zunächst wird die Tiefensuche mit dem Kindknoten von *k* fortgesetzt und bei dieser Tiefensuche eine geordnete Liste $CL(k)$ mit den sich im Teilbaum des Syntaxbaums mit Wurzel *k* befindenden Syntaxbaumknoten in der Reihenfolge der Tiefensuche erstellt.

Anschließend werden die virtuellen Eingabeparameter und die virtuellen Ausgabeparameter der von *k* repräsentierten Schleife ermittelt. Diese Parameter werden von der Analysephase des CM-task Compilers nicht explizit im ausgegebenen erweiterten Spezifikationsprogramm gespeichert, sondern sind implizit durch die eingefügten Datenabhängigkeiten für den entsprechenden Schleifenrumpf gegeben. Virtuelle Eingabeparameter (bzw. Ausgabeparameter) der von *k* repräsentierten Schleife sind alle Variablen, für die an *k* eine Datenabhängigkeit mit Schreibzugriff (bzw. Lesezugriff) *k* annotiert ist.

Die Eingabedatenverteilung bzw. die Ausgabedatenverteilung für einen bestimmten virtuellen Parameter *a* der von *k* repräsentierten Schleife wird anhand der Reihenfolge der Lese- bzw. Schreibzugriffe auf *a* durch die sich in der Liste $CL(k)$ befindenden Konstrukte gemäß den

oben aufgeführten Regeln für Schleifen bestimmt.

Für `if`-Knoten k werden zwei Listen $CL_1(k)$ und $CL_2(k)$ gebildet, die die bei der Tiefensuche betrachteten Syntaxbaumknoten im Teilbaum des if-Zweigs bzw. des else-Zweigs enthalten. Anschließend wird analog wie bei den Schleifen vorgegangen, d.h. es werden die virtuellen Eingabe- und Ausgabeparameter des `if`-Konstrukts bestimmt und zunächst anhand der Liste $CL_1(k)$ und (falls kein Lese- bzw. Schreibzugriff auf einen bestimmten Parameter gefunden wurde) anhand der Liste $CL_2(k)$ die entsprechenden Eingabe- und Ausgabedatenverteilungen gemäß den obigen Regeln festgelegt.

5.3.2. Einfügen der Datenumverteilungsoperationen

In diesem Abschnitt wird zunächst beschrieben, wie für eine gegebene Datenabhängigkeit die zugehörige Datenumverteilungsoperation bestimmt wird. Anschließend wird die Umsetzung im CM-task Compiler für einen gegebenen abstrakten Syntaxbaum des Rahmenprogrammes mit annotierten Datenabhängigkeiten und Prozessorgruppen vorgestellt.

Bestimmung einer Datenumverteilungsoperation. Im Folgenden wird eine bestimmte Datenabhängigkeit mit Schreibzugriff s, Lesezugriff t und Variable a betrachtet. Die zugehörige Datenumverteilungsoperation findet zwischen einem Quellkonstrukt q und einem Zielkonstrukt r statt. Zur Laufzeit der Anwendung führt das Quellkonstrukt die Sendephase der Datenumverteilungsoperation aus, wohingegen das Zielkonstrukt für die Ausführung der korrespondierenden Empfangsphase verantwortlich ist.

Das Zielkonstrukt der Datenumverteilungsoperation ist immer identisch mit dem Lesezugriff der entsprechenden Datenabhängigkeit, d.h es gilt $t = r$. Dagegen können sich Schreibzugriff der Datenabhängigkeit und Quellkonstrukt der zugehörigen Datenumverteilungsoperation unterscheiden, falls zwischen der Ausführung von s und t weitere Zugriffe auf a erfolgen. Konkret wird das Quellkonstrukt q derart festgelegt, dass die folgenden drei Bedingungen erfüllt sind:

(i) Das Quellkonstrukt q besitzt den durch den Schreibzugriff s geschriebenen Wert der Variable a;

(ii) Das Quellkonstrukt q wird vor dem Zielkonstrukt r ausgeführt und

(iii) zwischen der Ausführung von Quellkonstrukt q und Zielkonstrukt r erfolgt durch keinen q oder r ausführenden Prozessor ein Zugriff auf die Variable a.

Bedingung (i) wird durch den Schreibzugriff s und alle Lesezugriffe auf den durch s geschriebenen Wert von a erfüllt. Bedingung (iii) sorgt dafür, dass die Datenverteilung der Variable a zwischen der Ausführung von q und r nicht verändert wird.

Diese Auswahlkriterien für q bieten zwei Vorteile. Zum einen wird der zeitliche Abstand zwischen der Ausführung von Quell- und Zielkonstrukt möglichst gering gehalten. Damit werden zur Laufzeit der Anwendung die durch die Kommunikation zwischen Quell- und Zielkonstrukt belegten Ressourcen nur kurze Zeit benötigt. Zum anderen kann auf die Ausführung der Datenumverteilungsoperation verzichtet werden, falls q und r identische Prozessorgruppen und identische Datenverteilungen zur Speicherung der Variable a verwenden.

Realisierung im CM-task Compiler. Die Datenverteilungsphase des CM-task Compilers führt zur Umsetzung der beschriebenen Vorgehensweise einen Tiefensuchlauf über einen gegebenen abstrakten Syntaxbaum des Rahmenprogrammes aus. An den Knoten dieses Syntaxbaums sind die in der Analysephase bestimmten Datenabhängigkeiten und die in der Schedulingphase des CM-task Compilers bestimmten Prozessorgruppen annotiert. Desweiteren sind an den `for`-, `while`- und `if`-Knoten die im vorherigen Schritt der Datenverteilungsphase festgelegten Eingabe- und Ausgabedatenverteilungen für die zugehörigen virtuellen Eingabe- und Ausgabeparameter annotiert. An den Blattknoten des Syntaxbaums sind die Datenverteilungen für die Eingabe- und Ausgabeparameter des zugehörigen Modulaufrufs entsprechend der vom Anwender definierten Schnittstelle des jeweiligen aufgerufenen Moduls annotiert.

Für einen in der Tiefensuche angetroffenen Syntaxbaumknoten k wird wie folgt vorgegangen. Zunächst wird die Tiefensuche mit den Kindknoten von k fortgesetzt, wobei eine geordnete Liste $CL(k)$ erzeugt wird, die alle Syntaxbaumknoten im Teilbaum des Syntaxbaums mit Wurzel k in der Reihenfolge der Tiefensuche enthält.

Anschließend werden alle an k annotierten Datenabhängigkeiten nacheinander betrachtet. Für eine Datenabhängigkeit mit Schreibzugriff s und Lesezugriff t werden alle Kandidaten für das Quellkonstrukt q der zugehörigen Datenumverteilungsoperation bestimmt, die Bedingung (i) erfüllen. Diese Kandidaten sind der Schreibzugriff s und alle Lesezugriffe auf a, für die eine Datenabhängigkeit mit Schreibzugriff s besteht. Eine derartige Datenabhängigkeit ist entweder an k oder einem Knoten aus der Liste $CL(k)$ annotiert.

Das Quellkonstrukt q wird als der Kandidat mit der maximalen Position in der Liste $CL(k)$ bestimmt, so dass sich (a) q in der Liste $CL(k)$ vor dem Lesezugriff t befindet und dass (b) die an q und t annotierten Prozessorgruppen nicht disjunkt sind. Bedingung (b) wird benötigt, um sicherzustellen, dass q vor t ausgeführt wird, also Bedingung (ii) erfüllt ist. Existiert kein derartiges Konstrukt, etwa weil die Prozessorgruppe jedes Kandidaten mit der an t annotierten Prozessorgruppe disjunkt ist, wird der Schreibzugriff s als Quellkonstrukt festgelegt. Aufgrund der Datenabhängigkeit zwischen s und t wird s stets vor t ausgeführt.

Durch das Quellkonstrukt q werden sowohl die Quelldatenverteilung als auch die Quellprozessorgruppe für die Variable a festgelegt. Die Zieldatenverteilung und die Zielprozessorgruppe für a ergeben sich aus den entsprechenden Annotationen des Lesezugriffs t. Im statischen Compileransatz wird nur dann eine Datenumverteilungsoperation erzeugt, wenn sich Quell- und Zieldatenverteilung oder Quell- und Zielprozessorgruppe unterscheiden. Im semi-dynamischen Compileransatz wird in jedem Fall eine Datenumverteilungsoperation erzeugt und erst zur Laufzeit der Anwendung abhängig von der aktuell dem Quellkonstrukt und dem Zielkonstrukt zugewiesenen Prozessorgruppen über die Ausführung der entsprechenden Datenumverteilungsoperation entschieden. Die Annotation der erzeugten Datenumverteilungsoperation erfolgt sowohl im statischen als auch im semi-dynamischen Compileransatz am Syntaxbaumknoten k.

Ein Beispiel für das Einfügen von Datenumverteilungsoperationen ist in Abbildung 31 angegeben. In diesem Beispiel besteht eine Datenabhängigkeit vom Schreibzugriff A (Identifikator: `n1`) zum Lesezugriff D (Identifikator: `n4`) für die Variable a. Als Kandidaten für das Quellkonstrukt der zugehörigen Datenumverteilungsoperation werden der Schreibzugriff A sowie die Lesezugriffe B und C bestimmt. Aufgrund der Reihenfolge der Tiefensuche, die sich aus der Reihenfolge der Definition im Rahmenprogramm ergibt, wird C als Quellkonstrukt der Datenumverteilungsoperation ausgewählt. Für die restlichen drei Datenabhängigkeiten (von A zu B, von A zu C und von B zu D) existiert jeweils nur ein Kandidat für das Quellkonstrukt, der in jedem dieser Fälle mit dem Schreibzugriff der jeweiligen Datenabhängigkeit identisch ist.

```
seq {
 n1:A(a/* out */) on {1..64};
 par {
  n2:B(a/* in */, b/*out */)
    on {1..16};
  n3:C(a/* in */) on {17..64};}
 n4:D(a/* in */, b/* in */)
   on {1..64};
 prels {
  (n1->n2,{a}),(n1->n3,{a}),
  (n1->n4,{a}),(n2->n4,{b}) }}
```

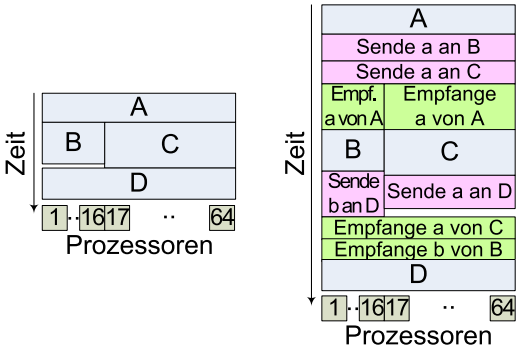


Abbildung 31: Beispiel für das Einfügen von Datenumverteilungsoperationen. Links: Modulausdruck des Rahmenprogrammes mit annotierten Datenabhängigkeiten (`prels`-Konstrukt) und Prozessorgruppen. Mitte: Illustration der vom Rahmenprogramm definierten Ausführungsreihenfolge der Modulaufrufe. Rechts: Illustration der resultierenden Abarbeitung nach Einfügen der Datenumverteilungsoperationen.

5.3.3. Annotation von Lastausgleichsoperationen

Der semi-dynamische Compileransatz bestimmt in diesem Arbeitsschritt, in welchen Schleifenrümpfen und welchen Bedingungszweigen eines gegebenen Verbundmoduls zur Laufzeit der Anwendung ein Lastausgleich stattfinden soll. Für jeden Lastausgleich wird zusätzlich die Häufigkeit der Ausführung festgelegt.

Der CM-task Compiler verwendet zu diesem Zweck eine Heuristik, die festlegt, dass

(a) Lastausgleichsoperationen in jedem Schleifenrumpf einer `for`- oder `while`-Schleife sowie in jedem Bedingungszweig mit mindestens einer umgebenden `for`- oder `while`-Schleife stattfinden und dass

(b) jeder Lastausgleich einmal pro Schleifeniteration der jeweils äußersten Schleife (`for` oder `while`) erfolgt.

Durch Festlegung (a) wird gewährleistet, dass Lastausgleichsoperationen nur für mehrfach abgearbeitete Berechnungen ausgeführt werden. Festlegung (b) wurde gewählt, um einen zu häufigen Lastausgleich in inneren Schleifen zu vermeiden und damit den Anteil der durch die Lastausgleichsoperationen entstehenden Koordinationsoverhead an der Gesamtausführungszeit der jeweiligen Anwendung gering zu halten.

Die Realisierung dieses Arbeitsschrittes erfolgt durch einen Tiefensuchlauf über den abstrakten Syntaxbaum eines gegebenen Verbundmoduls im Rahmenprogramm. Bei diesem Tiefensuchlauf wird für jeden Syntaxbaumknoten k eine Liste der k umgebenden `for`- und `while`-Schleifen erstellt, d.h. die Liste enthält alle `for`- und `while`-Knoten des Syntaxbaums, die auf dem Pfad von der Syntaxbaumwurzel zu k liegen. Eine Lastausgleichsoperation wird an einem Knoten k annotiert, falls k ein `for`- oder `while`-Knoten ist oder falls k ein `if`-Knoten mit mindestens einer umgebenden Schleife ist.

Die Häufigkeit einer Lastausgleichsoperation wird durch die zugehörige *Lastausgleichsgranularität* definiert. Die Lastausgleichsgranularität eines mit einer Lastausgleichsoperation

annotierten Konstrukts gibt die Abarbeitungsanzahl des entsprechenden Konstrukts an, nach der jeweils eine Lastausgleichsoperation ausgeführt wird. Für einen Syntaxbaumknoten k mit annotierter Lastausgleichsannotation wird die Lastausgleichsgranularität als Produkt aus der Iterationsanzahl aller k umgebenden Schleifen berechnet.

5.4. Codegenerierungsphase des CM-task Compilers

Der statische und der semi-dynamische Compileransatz enthalten unterschiedliche Codegenerierungsphasen, die im Folgenden vorgestellt werden.

5.4.1. Codegenerator für statische Koordinationsprogramme

Der statische Codegenerator arbeitet in zwei Schritten, die jeweils durch einen Tiefensuchlauf über einen gegebenen abstrakten Syntaxbaum einer Verbundmoduldefinition im erweiterten Rahmenprogramm realisiert sind. Dies bedeutet, dass an den Knoten des betrachteten Syntaxbaums Annotationen für die ausführende Prozessorgruppe, für die Daten- und Kommunikationsabhängigkeiten und für die auszuführenden Datenumverteilungsoperationen vorhanden sind.

Der Tiefensuchlauf des ersten Schrittes erzeugt aus den lokalen Annotationen der Syntaxbaumknoten jeweils eine globale Liste mit allen annotierten Prozessorgruppen, allen annotierten Kommunikationsabhängigkeiten und allen Datenumverteilungsoperationen. Aus der Liste der annotierten Prozessorgruppen werden die zur Ausführung des jeweiligen Verbundmoduls benötigten MPI Kommunikatoren bestimm und entsprechender Koordinationscode zur Initialisierung bzw. zur Freigabe der MPI Kommunikatoren erzeugt (vgl. Listing 9 auf Seite 66 Zeilen 13-16 bzw. Zeilen 39-41). Identische Prozessorgruppen werden dabei auf denselben MPI Kommunikator abgebildet.

Für jede ermittelte Kommunikationsabhängigkeit wird eine Kommunikationsstruktur erzeugt und im globalen Datenteil des ausgegebenen Koordinationsprogrammes abgelegt. Die erzeugte Kommunikationsstruktur stellt Informationen zur Realisierung von Kommunikationsoperationen zwischen zeitgleich ausgeführten Basismodulen bereit und wird zur Laufzeit den an der zugehörigen Kommunikationsabhängigkeit beteiligten Basismodulen zur Verfügung gestellt.

Für jede ermittelte Datenumverteilungsoperation wird das Kommunikationsmuster zwischen Quell- und Zielprozessoren vorberechnet (vgl. Abschnitt 5.5). Dieses Kommunikationsmuster wird ebenfalls im globalen Datenteil des Koordinationsprogrammes gespeichert.

Die zweite Phase des statischen Codegenerators basiert auf einem syntaxgerichteten Übersetzungsschema. Die Realisierung erfolgt durch einen Tiefensuchlauf über den betrachteten abstrakten Syntaxbaum, wobei für jeden Knoten des Syntaxbaums die entsprechenden Codefragmente für Prozessorauswahl, für die Ausführung der Empfangsoperationen, für die Bereitstellung der Kommunikationsstrukturen, für die Ausführung des zugeordneten Konstrukts und für die Ausführung der Sendeoperationen eingefügt wird (vgl. Abschnitt 4.6.1). Vor diesem Tiefensuchlauf werden die `parfor`- und `cparfor`-Schleifen entrollt (s. Abschnitt 5.1.2).

5.4.2. Codegenerator für semi-dynamische Koordinationsprogramme

Im semi-dynamischen Compileransatz erfolgt die Codegenerierung für ein gegebenes Verbundmodul in zwei Arbeitsschritten. Der erste Schritt übersetzt die durch das Rahmenprogramm

festgelegte Ausführungsreihenfolge und die zugewiesenen Prozessorgruppen in eine Menge von *Schedulestrukturen*. Die generierten Datenstrukturen werden als statisch initialisierte globale Daten im ausgegebenen Koordinationsprogramm abgespeichert.

Der zweite Arbeitsschritt erzeugt die Koordinationsfunktion des betrachteten Verbundmoduls mit Hilfe eines syntaxgerichteten Übersetzungsschemas. Die Realisierung erfolgt durch einen Tiefensuchlauf über den abstrakten Syntaxbaum des jeweiligen Verbundmoduls, wobei für jeden Knoten abhängig vom zugehörigen Konstrukt Codefragmente erzeugt werden. Die in diesem Arbeitsschritt erzeugten Codefragmente (z.B. zur Prozessorauswahl) sind in Abschnitt 4.6.2 aufgeführt und erklärt. Im Folgenden wird der Aufbau und die Erzeugung der Schedulestrukturen näher beschrieben.

Eine *Schedulestruktur* ist eine Datenstruktur, die den Schedule eines Programmblockes im semi-dynamischen Koordinationsprogramm repräsentiert, d.h. den Schedule eines Schleifenrumpfes einer `for`- oder `while`-Schleife, eines Bedingungszweiges oder eines gesamten Verbundmoduls. Der Aufbau einer Schedulestruktur ist dreistufig:

- Die untere Stufe besteht aus je einer *Taskstruktur* für jeden im entsprechenden Programmblock auftretenden Basis- oder Verbundmodulaufruf, jede `for`- oder `while`-Schleife und jede Bedingung (`if`). Die Taskstruktur speichert für das zugehörige Konstrukt die aktuell zugeordnete Prozessorgruppe mit dem entsprechenden MPI Kommunikator, die zur Ausführung benötigte Mindestanzahl an Prozessoren sowie die dynamisch ermittelten Leistungsdaten.

  Die Leistungsdaten bestehen aus der Ausführungsanzahl und der kumulierten Ausführungszeit des zugehörigen Konstrukts. Die Mindestprozessoranzahl ergibt sich aus der in der Schedulingphase des CM-task Compilers festgelegten Ausführungsreihenfolge und entspricht der maximalen Anzahl zeitgleich ausgeführter Basismodule im Schleifenrumpf der jeweiligen Schleife bzw. in den Bedingungszweigen der jeweiligen Bedingung.

- Die mittlere Stufe der Schedulestruktur besteht aus einer Menge von *Schichtstrukturen*, wobei eine Schichtstruktur die zeitgleiche Ausführung einer Menge von Modulen definiert. Ein Modul kann dabei ein Basis- oder Verbundmodul oder ein komplettes `for`-, `while`- oder `if`-Konstrukt sein. In einer Schichtstruktur werden zwei Felder gespeichert. Das erste Feld enthält Verweise auf die Taskstrukturen der zeitgleich auszuführenden Module. Das zweite Feld enthält Verweise auf kodierte Kommunikationsabhängigkeiten, die zwischen den zeitgleich auszuführenden Basismodulen der jeweiligen Schichtstruktur bestehen. Jede Kommunikationsabhängigkeit enthält ein Feld mit Verweisen auf die Taskstrukturen der beteiligten Basismodule und einen Verweis auf die zugehörige Kommunikationsstruktur.

  Zur Laufzeit ist ein Lastausgleich nur zwischen denjenigen Modulen möglich, deren Taskstrukturen in derselben Schichtstruktur gespeichert sind. Die Verweise auf die Taskstrukturen und Kommunikationsabhängigkeiten werden benötigt, um die dynamischen Leistungsdaten zu sammeln und nach erfolgtem Lastausgleich die entsprechenden MPI Kommunikatoren der Taskstrukturen sowie die in den Kommunikationsstrukturen gespeicherten externen Kommunikatoren anzupassen.

- Die obere Stufe einer Schedulestruktur ist die *Blockstruktur*, die die Nacheinanderausführung einer Menge von Schichtstruktur definiert. Dazu wird in der Blockstruktur ein Feld mit Verweisen auf die entsprechenden Schichtstrukturen gespeichert.

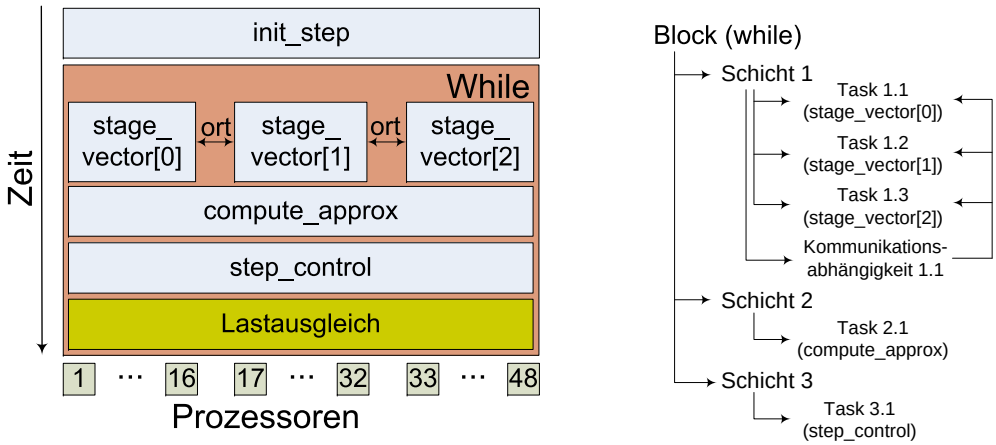


Abbildung 32: Ausführungsschema für Iterierte Runge-Kutta Verfahren mit enthaltenen Kommunikationsabhängigkeiten (links) und Illustration der Schedulestruktur für den Rumpf der `while`-Schleife (rechts). Die Schedulestruktur besteht aus einer Blockstruktur, die Verweise auf die nacheinander auszuführenden Schichtstrukturen speichert. Jede Schichtstruktur enthält Verweise auf die Taskstrukturen der zeitgleich auszuführenden Module und auf die Kommunikationsabhängigkeiten zwischen diesen Modulen. Die Kommunikationsabhängigkeit verweist auf die Taskstrukturen der kommunizierenden Basismodule.

Der konkrete Aufbau der Schedulestruktur wird in Abschnitt C.3 des Anhangs durch eine *C*-Datentypdefinition angegeben. Abbildung 32 illustriert die Schedulestruktur für das semi-dynamische Koordinationsprogramm für IRK Verfahren.

Der Aufbau der Schedulestruktur wurde mit dem Ziel gewählt, zur Laufzeit Lastausgleichsoperationen mit möglichst geringem Koordinationsoverhead zu ermöglichen. Der Nachteil dieses Aufbaus besteht darin, dass nicht jede durch das Rahmenprogramm definierbare Ausführungsreihenfolge in eine entsprechende Schedulestruktur übersetzt werden kann. Beispielsweise kann ein `par`-Konstrukt mit enthaltenem `seq`-Konstrukt nicht durch eine Schedulestruktur dargestellt werden.

Der CM-task Compiler trägt dieser Einschränkung Rechnung, indem im semi-dynamischen Compileransatz eine entsprechend angepasste Version des Schedulingalgorithmus aus Kapitel 3 eingesetzt wird. Die Anpassung betrifft das Scheduling einer Schicht des Supertask Graph (s. Algorithmus 2 auf Seite 30). In der angepassten Version dieses Schrittes wird entweder die zeitgleiche Ausführung aller Supertasks einer Schicht oder die Nacheinanderausführung aller Supertasks einer Schicht festgelegt, je nachdem welche Variante zur niedrigeren Ausführungszeit der entsprechenden Schicht führt. In beiden Varianten entsteht ein Schedule, der durch die beschriebene Schedulestruktur dargestellt werden kann.

Die Erzeugung der Schedulestruktur erfolgt durch einen Tiefensuchlauf über den abstrakten Syntaxbaums eines gegebenen Verbundmoduls im erweiterten Rahmenprogramm. Bei diesem Tiefensuchlauf wird für jeden Blattknoten eine Taskstruktur erzeugt. An denjenigen `par`- und `cpar`-Knoten *k*, die nicht Kindknoten eines anderen `par`- oder `cpar`-Knotens sind, wird eine Schichtstruktur erzeugt, in der die Taskstrukturen aller sich im Teilbaum des Syntaxbaums mit Wurzel *k* befindender Blattknoten enthält. An `for`-, `while`-, `if`- und Wurzelknoten wird

jeweils eine Blockstruktur erzeugt, die die im entsprechenden Teilbaum des Syntaxbaums erzeugten Schichtstrukturen in der Reihenfolge der Tiefensuche enthält. Die `parfor`- und `cparfor`-Knoten werden vor diesem Schritt im Syntaxbaum entrollt (s. Abschnitt 5.1) und wie `par`- bzw. `cpar`-Knoten behandelt.

5.5. Datenumverteilungsbibliothek

Die Datenumverteilungsbibliothek des CM-task Compilerframeworks stellt Datenumverteilungsoperationen für blockzyklische und replizierte Datenverteilungen mehrdimensionaler Felder bereit. Die Realisierung der Datenumverteilungsoperationen erfolgt durch direkte Punkt-zu-Punkt Kommunikationsoperationen zwischen den Prozessoren der Quellprozessorgruppe und den Prozessoren der Zielprozessorgruppe.

Die auszuführenden Kommunikationsoperationen zur Umverteilung eines d-dimensionalen Feldes werden in zwei Arbeitsschritten bestimmt.

(i) **Bestimmung eindimensionaler Kommunikationsmuster.**
Im ersten Schritt wird für jede Felddimension ein eigenes Kommunikationsmuster bestimmt. Für eine bestimmte Dimension wird anhand der parametrisierten Datenverteilungsvektoren der Quell- und Zielverteilungstyp (repliziert oder blockzyklisch) sowie die Quell- und Zielprozessoranzahl dieser Dimension bestimmt. Zur Berechnung des Kommunikationsmusters werden in diesem Schritt virtuelle Quell- und Zielprozessorgruppen mit den entsprechenden Prozessoranzahlen verwendet.

Für blockzyklische Quelldatenverteilungen sind die auszuführenden Kommunikationsoperationen eindeutig bestimmt und werden durch den in [48] vorgestellten Algorithmus berechnet. Dieser Algorithmus teilt die umzuverteilenden Feldelemente in Superblöcke mit identischem Kommunikationsmuster ein. Das Kommunikationsmuster wird nur für einen ausgewählten Superblock berechnet und anschließend auf die weiteren Superblöcke übertragen.

Für replizierte Quelldatenverteilungen bestehen mehrere mögliche Kommunikationsmuster, da jeder Quellprozessor alle Elemente der umzuverteilenden Datenstruktur besitzt. Die Datenumverteilungsbibliothek gewährleistet in diesem Fall ein ausgewogenes Kommunikationsverhalten, indem die auszuführenden Sendeoperationen gleichmäßig auf die Quellprozessoren verteilt werden. Dazu wird jedem virtuellen Zielprozessor reihum ein virtueller Quellprozessor zugeordnet, der für die Bereitstellung aller für den jeweiligen Zielprozessor benötigter Daten verantwortlich ist.

(ii) **Kombination der eindimensionale Kommunikationsmuster.**
Die auszuführenden Kommunikationsoperationen zwischen einem bestimmten Quellprozessor p_1 und einem bestimmten Zielprozessor p_2 werden mit Hilfe der durch die parametrisierten Datenverteilungsvektoren beschriebenen Prozessorgitter berechnet. Die Gitterkoordinaten von p_1 und p_2 bestimmen die Positionen innerhalb der virtuellen Quell- bzw. Zielprozessorgruppen der einzelnen Dimensionen. Anhand der berechneten eindimensionalen Kommunikationsmuster werden diejenigen Feldelemente bestimmt, die von p_1 zu p_2 übertragen werden müssen.

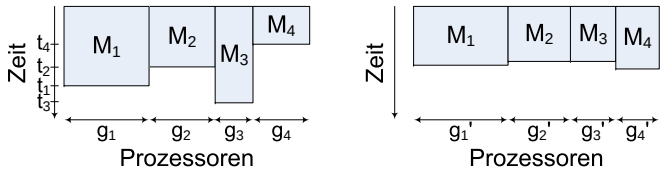


Abbildung 33: Illustration der Situation vor (links) und nach (rechts) einer Lastausgleichsoperation zwischen vier zeitgleich ausgeführten Modulen $\{M_1, M_2, M_3, M_4\}$.

5.6. Lastausgleichsbibliothek

In diesem Abschnitt wird der im CM-task Compilerframework verwendete dynamische Lastausgleichsalgorithmus beschrieben. Dieser Algorithmus wird zur Laufzeit semi-dynamischer Koordinationsprogramme zur Anpassung der Prozessorgruppen von n zeitgleich ausgeführten Modulen $M_1, \ldots, M_n$ verwendet. Ein derartiges Modul ist ein Basis- oder Verbundmodulaufruf, eine komplette Schleife (`for` oder `while`) oder eine komplette Bedingung (`if`-Konstrukt).

Der Lastausgleichsalgorithmus trifft seine Entscheidungen basierend auf der für Modul M_i gemessenen Ausführungszeit t_i und der aktuell verwendeten Prozessorgruppengröße g_i, $i = 1, \ldots, n$. Abbildung 33 (links) illustriert die Abarbeitungssituation vor Ausführung des Lastausgleichsalgorithmus für $n = 4$ Module.

Die Ausgabe des Lastausgleichsalgorithmus umfasst eine Entscheidung, ob eine Modifikation der Prozessorgruppen erforderlich ist und gegebenenfalls eine angepasste Prozessorgruppengröße g_i' für Modul M_i. Abbildung 33 (rechts) zeigt die Abarbeitungssituation nach erfolgtem Lastausgleich. Bei der Berechnung der Prozessorgruppengrößen g_i' muss der Algorithmus beachten, dass für Verbundmodulaufrufe, Schleifen und Bedingungen eine Mindestanzahl von Prozessoren $minp_i$ erforderlich ist. Diese Mindestanzahl entspricht der maximalen Anzahl zeitgleich ausgeführter Basismodule im entsprechenden Verbundmodul, dem entsprechenden Schleifenrumpf bzw. den entsprechenden Bedingungszweigen. Da die Ausführungsreihenfolge zur Compilezeit feststeht, wird die erforderliche Mindestprozessoranzahl für jedes Modul statisch durch eine bottom-up Traversierung des entsprechenden abstrakten Syntaxbaums ermittelt.

Der Lastausgleichsalgorithmus ist in Algorithmus 8 zusammengefasst und gliedert sich in die folgenden drei Arbeitsschritte.

(i) **Feststellen eines Lastungleichgewichts**
Im ersten Teilschritt wird überprüft, ob ein eventuell vorliegendes Lastungleichgewicht den zusätzlichen Aufwand einer Lastausgleichsoperation rechtfertigt. Als Maß r für das bestehende Lastungleichgewicht wird das Verhältnis aus der maximalen und der minimalen Ausführungszeit der betrachteten Module verwendet. Ein Lastausgleich erfolgt, falls das Lastungleichgewicht einen vorgegebenen Wert R überschreitet.

Die Wahl eines guten Wertes für R muss sowohl die Eigenschaften der konkreten Anwendung als auch plattformspezifische Werte wie die Anzahl der verfügbaren Prozessoren berücksichtigen. Eine höhere Prozessoranzahl erlaubt bspw. feinere Anpassungen der Prozessorgruppengrößen und ermöglicht damit auch niedrigere Werte für R. Für Berechnungen deren Laufzeit stark schwanken kann, sind dagegen höhere Werte für R sinnvoll, um zu häufige Anpassungen zu vermeiden. Die in Abschnitt 6.3.2 durchgeführ-

Algorithmus 8 : Dynamischer Lastausgleichsalgorithmus

Eingabe : Menge von n Modulen $\{M_1,\ldots,M_n\}$
Eingabe : gemessene Ausführungszeiten $t_i, i = 1,\ldots,n$
Eingabe : aktuelle Prozessorgruppengrößen $g_i, i = 1,\ldots,n$
Eingabe : minimale Prozessoranzahlen $minp_i, i = 1,\ldots,n$
Eingabe : verfügbare Prozessoranzahl P; maximales Lastungleichgewicht R
Rückgabe : **true**, falls Lastausgleich stattfinden soll
Ausgabe : neue Prozessorgruppengrößen $g'_i, i = 1,\ldots,n$

```
1  begin
       /* (i) Feststellen eines Lastungleichgewichts          */
2      r = max_{i=1,...,n} t_i / min_{i=1,...,n} t_i;          /* Laufzeitverhältnis */
3      if (r > R) then
           /* (ii) Berechnung angepasster Gruppengrößen       */
4          W = sum_{i=1}^{n} t_i * g_i ;          /* gesamte Berechnungsarbeit */
5          for (j = 1,...,n) do
6              g'_j = max{ minp_j, round( (t_j*g_j)/W * P ) }
           /* (iii) Korrektur der Gruppengrößen               */
7          p = sum_{i=1}^{n} g'_i;                /* Summe der Prozessorzahlen */
8          while (p ≠ P) do
9              if (p < P) then
10                 wähle Modul M_i, so dass g_i * t_i / g'_i maximal ist
11                 g'_i = g'_i + 1; p = p + 1;
12             else
13                 wähle Modul M_i mit g'_i > minp_i und (g_i * t_i)/(g'_i - 1) minimal
14                 g'_i = g'_i - 1; p = p - 1;
15         return true;
16     else return false;
17 end
```

ten Laufzeituntersuchungen verwenden den Wert $R = 1.1$, der sich durch empirische Untersuchungen für die betrachteten Anwendungen als günstig erwiesen hat.

(ii) **Berechnung angepasster Gruppengrößen**
Die angepasste Prozessorgruppengröße g'_i eines Moduls M_i wird durch den Anteil der Berechnungsarbeit von M_i an der Gesamtberechnungsarbeit W aller n Module bestimmt. Die Berechnungsarbeit eines Moduls M_i ist als Produkt aus Prozessorgruppengröße und Ausführungszeit $(t_i * g_i)$ definiert und kann als Flächeninhalt von M_i in Abbildung 33 (links) aufgefasst werden.

(iii) **Korrektur der Gruppengrößen**
Rundungsfehler und die Berücksichtigung der Mindestprozessoranzahlen $minp_i$ im vorhergehenden Schritt können dazu führen, dass die Summe der angepassten Prozessorgruppengrößen p nicht mit der verfügbaren Prozessoranzahl P übereinstimmt. In diesem

Arbeitsschritt erfolgt daher sukzessive eine Anpassung der Gruppengrößen g_i', wobei die folgenden beiden Fälle unterschieden werden:

(a) Wurden zuwenige Prozessoren zugeordnet, wird einem Modul M_i, dessen erwartete Laufzeit mit der angepassten Gruppengröße g_i' maximal ist, ein zusätzlicher Prozessor zugeordnet. Für die Berechnung der erwarteten Laufzeit wird eine lineare Skalierung der Ausführungszeit angenommen.

(b) Wurden zuviele Prozessoren verteilt, wird die Gruppengröße g_i' eines Moduls M_i reduziert. Dieses Modul wird derart ausgewählt, dass die erwartete Laufzeit mit der reduzierten Gruppengröße minimal ist. Analog zum vorherigen Fall wird eine lineare Skalierbarkeit des Moduls angenommen.

Die Komplexität des vorgestellten Lastausgleichsalgorithmus lässt sich wie folgt abschätzen. Das Feststellen, ob ein Lastausgleich notwendig ist (Schritt (i)), erfordert einen Durchlauf über alle Module, also $\mathcal{O}(n)$ Operationen. Die Berechnung adaptierter Prozessorgruppengrößen (Schritt (ii)), benötigt einen konstanten Aufwand pro Modul, was zu einer Komplexität von $\mathcal{O}(n)$ führt. Jede ausgeführte Gruppengrößenkorrektur (Schritt (iii)) benötigt zur Auswahl eines geeigneten Moduls $\mathcal{O}(n)$ Schritte. Die Anzahl der ausgeführten Korrekturschritte ist in vielen praktischen Fällen gering, kann aber im ungünstigsten Fall $\mathcal{O}(P)$ betragen. Insgesamt besitzt der Lastausgleichsalgorithmus eine Komplexität von $\mathcal{O}(nP)$ und ist daher besonders gut für den Einsatz zur Laufzeit einer Anwendung geeignet.

5.7. Zusammenfassung und verwandte Arbeiten

In diesem Kapitel wurde die Realisierung der Komponenten des CM-task Compilers, der Datenumverteilungsbibliothek und der Lastausgleichsbibliothek vorgestellt.

Die **Analysephase** des CM-task Compilers bestimmt die in einem gegebenen Spezifikationsprogramm implizit definierten Daten- und Kommunikationsabhängigkeiten durch bottom-up Attributierung der abstrakten Syntaxbäume der im entsprechenden Spezifikationsprogramm definierten Verbundmodule.

Zur Bestimmung der Datenabhängigkeiten in der der CM-task Spezifikationssprache zugrundeliegenden TwoL-Grammatik [92] wurden zwei Algorithmen entwickelt. In [102] wird ein top-down Ansatz vorgestellt, der einen gegebenen abstrakten Syntaxbaum durchläuft und für jeden angetroffenen Lesezugriff den zugehörigen Schreibzugriff durch einen Rückwärtslauf über die Liste der in der Tiefensuche besuchten Knoten bestimmt. Dieser Algorithmus unterstützt im Gegensatz zum CM-task Compiler keine Schleifen und Bedingungen.

Ein bottom-up Ansatz zur Bestimmung der Datenabhängigkeiten in einem TwoL-Spezifikationsprogramm wird in [37] verwendet. Dieser Algorithmus konstruiert zunächst den durch die Konstruktoren des gegebenen Spezifikationsprogrammes definierten Abhängigkeitsgraph. Der zugehörige Schreibzugriff eines Lesezugriffes wird durch einen Rückwärtslauf über die Kanten des konstruierten Graph ermittelt. Für Schleifen und Bedingungen wird analog zum CM-task Compiler eine virtuelle Parameterliste bestimmt. Dazu wird eine bottom-up Attributierung der entsprechenden Grammatik definiert.

Die **Schedulingphase** des CM-task Compilers verwendet einen hierarchischen Schedulingalgorithmus, der für eine gegebene Anwendung eine Menge von Teilschedules in Form von Syntaxbaumfragmenten erzeugt, die durch mehrere Transformationsschritte in ein Rahmenprogramm übersetzt werden.

Der Paradigm Compiler [86, 85] verwendet ebenfalls einen hierarchischen Schedulingansatz zur Berechnung eines globalen Schedules für eine gemischt parallele Anwendung auf Basis paralleler Tasks. Im Unterschied zum CM-task Compiler werden zur Kostenbestimmung von Schleifen und Bedingungen nur Schedules für ausgewählte Prozessoranzahlen berechnet. Die Kostenwerte für die übrigen Prozessoranzahlen werden durch eine Kurvenanpassung auf ein Funktionstemplate nach dem Amdahlschen Gesetz bestimmt. Die Ausgabe des berechneten Schedules in einer vom Anwendungsentwickler lesbaren Form ist nicht Bestandteil dieses Ansatzes.

Für Anwendungsspezifikationen im TwoL-Modell wird in [37] ein genetischer Schedulingalgorithmus verwendet, der auf einem flachen Taskgraphen der gesamten Anwendung arbeitet. Die Übersetzung des erzeugten Schedules in ein TwoL-Rahmenprogramm erfolgt durch Umwandlung des den Schedule darstellenden gerichteten Graph in einen SP-Graph. Schleifen und Bedingungen werden in diesem Ansatz nicht explizit betrachtet.

Compilerunterstützung für das hierarchische Scheduling gemischt paralleler Anwendungen wird in [121] diskutiert. Das Scheduling erfolgt mit Hilfe eines schichtenbasierten Algorithmus auf einem Abhängigkeitsgraph, der aus einem gegebenen Programm aufgebaut wird. Im zugrundeliegenden Anwendungsmodell werden die parallelen Tasks im Unterschied zum CM-task Ansatz aus einer Menge von sequentiellen Tasks zusammengesetzt.

Die **Datenverteilungsphase** des CM-task Compilers legt in einem gegebenen Rahmenprogramm die Datenverteilungen für die Eingabe- und Ausgabeparameter von Schleifen und Bedingungen fest und bestimmt die Datenumverteilungsoperationen, die für einen korrekten Datenfluss im zu erzeugenden Koordinationsprogramm benötigt werden. Beide Teilschritte arbeiten mit Heuristiken, die auf eine Reduzierung der Anzahl der eingefügten Datenumverteilungsoperationen bzw. auf eine Reduzierung der entstehenden Datenumverteilungskosten durch Verwendung von möglichst identischen Quell- und Zielprozessorgruppen abzielen.

Die **Codegenerierungsphase** des CM-task Compilers verwendet ein syntaxgerichtetes Übersetzungsschema zur Transformation eines gegebenen erweiterten Rahmenprogrammes in ein Koordinationsprogramm. Im statischen Compileransatz werden die Ausführungsreihenfolge und die verwendeten Prozessorgruppen der Modulaufrufe fest im ausgegebenen Koordinationsprogramm kodiert. Im semi-dynamischen Compileransatz wird der in der Schedulingphase des CM-task Compilers berechnete Schedule in eine Schedulestruktur übersetzt. Diese Datenstruktur bildet die Schnittstelle zwischen semi-dynamischem Koordinationsprogramm und Lastausgleichsbibliothek. Die Schedulestruktur speichert dynamische Leistungsdaten und legt fest, zwischen welchen Modulen zur Laufzeit ein Lastausgleich stattfindet.

Die **Datenumverteilungsbibliothek** des CM-task Compilerframeworks verwendet zwei verschiedene Algorithmen zur Berechnung der auszuführenden Kommunikationsoperationen zur Umverteilung eines d-dimensionalen Feldes. Der in [48] vorgestellte Algorithmus wird für blockzyklische Quelldatenverteilungen verwendet. Für replizierte Quelldatenverteilungen werden die auszuführenden Kommunikationsoperationen zyklisch auf die Quellprozessoren verteilt, so dass ein ausgewogenes Kommunikationsverhalten entsteht.

Die Datenumverteilung zwischen verschiedenen blockzyklischen Datenverteilungen mehrdimensionaler Felder wurde vor allem in Zusammenhang mit HPF untersucht, z.B. in [119, 104, 43]. Der Fokus dieser Ansätze liegt auf datenparallelen Anwendungen, in denen alle verfügbaren Prozessoren zur Verteilung der Daten verwendet werden. Dagegen unterstützt die Datenumverteilungsbibliothek des CM-task Compilerframeworks beliebige Quell- und Zielprozessorgruppen.

Ein Algorithmus zur Datenumverteilung zwischen beliebigen Quell- und Zielprozessorgruppen wurde in [77] vorgestellt. Für diesen Algorithmus muss die Blockgröße der blockzyklischen Zieldatenverteilung ein ganzzahliges Vielfaches der Blockgröße der Quelldatenverteilung sein. Die Datenumverteilungsbibliothek des CM-task Compilerframeworks unterstützt dagegen beliebige Blockgrößen für die beteiligten Datenverteilungen. Der *FALLS*-Algorithmus [86] wurde zur effizienten Datenumverteilung zwischen blockzyklischen Datenverteilungen auf beliebigen Prozessorgruppen entwickelt. In [48] wurde aufgezeigt, dass der in der Datenumverteilungsbibliothek verwendete Algorithmus eine geringere Berechnungskomplexität besitzt und zu geringeren Ausführungszeiten der Datenumverteilungsoperationen führt.

Die **Lastausgleichsbibliothek** des CM-task Compilerframeworks enthält einen Lastausgleichsalgorithmus zur dynamischen Anpassung der Prozessorgruppengrößen von zeitgleich ausgeführten Modulen. Der Algorithmus berechnet die angepassten Prozessorgruppengrößen mit Hilfe der Berechnungsarbeit der entsprechenden Module, die als Produkt aus der jeweiligen Prozessorgruppengröße und der jeweiligen gemessenen Ausführungszeit definiert ist.

Dynamische Lastausgleichsoperationen werden häufig für irreguläre Anwendungen bestehend aus sequentiellen Tasks eingesetzt und können bspw. auf Graphpartitionierungsverfahren basieren [17]. Dynamische Lastausgleichsstrategien für datenparallele Anwendungen werden in [38] vorgestellt und hinsichtlich der Komplexität und Konvergenzkriterien analysiert.

6. Anwendungen und Experimente

In diesem Kapitel wird die Realisierung verschiedener Anwendungen aus dem Bereich des wissenschaftlichen Rechnens mit dem CM-task Compilerframework diskutiert und die Leistungsfähigkeit der erzeugten Koordinationsprogramme mit verschiedenen handgeschriebenen Programmversionen verglichen. Die für die Laufzeitmessungen genutzten Plattformen werden in Abschnitt 6.1 vorgestellt. Abschnitt 6.2 beschreibt die Anwendungsprogramme. Die erzielten Messergebnisse werden in Abschnitt 6.3 vorgestellt und ausgewertet. Den Abschluss des Kapitels bildet eine Zusammenfassung in Abschnitt 6.4.

6.1. Parallele Plattformen

Im Folgenden werden die Hardwareeigenschaften sowie die Softwareumgebung der für die Experimente verwendeten Plattformen beschrieben. Tabelle 4 gibt einen Überblick der wichtigsten Eigenschaften.

CHiC. Der **C**hemnitz **H**igh performance L**i**nux **C**luster integriert insgesamt 2120 Rechenkerne, die auf 530 Knoten mit je zwei Dual-Core Prozessoren vom Typ AMD Opteron 2218 „Santa Rosa“ verteilt sind. Die Taktrate der Prozessoren beträgt 2.6 GHz und führt zu einer maximalen Rechenleistung von 5.2 GFlops/s pro Rechenkern. Die Rechenknoten sind durch ein 10 GBit/s Infiniband Netzwerk miteinander verbunden. Für die in diesem Kapitel vorgestellten Messungen wurden die MVAPICH 1.0 MPI Bibliothek und der Pathscale Compiler 3.1 mit der höchsten Optimierungsstufe (`-O3`) eingesetzt.

SGI Altix 4700. Die SGI Altix 4700 Plattform besteht aus insgesamt 19 Partitionen mit je 512 Rechenkernen. Die Messungen wurden innerhalb einer Partition durchgeführt, die 128 Knoten mit jeweils zwei Intel Itanium2 „Montecito“ Dual-Core Prozessoren enthält. Die Prozessoren sind mit 1.6 GHz getaktet und besitzen eine maximale Rechenleistung von 6.4 GFlops/s pro Kern. Jeder Rechenknoten besitzt zwei Links zum NUMAlink4 Verbindungsnetzwerk, das eine bidirektionale Bandbreite von 6.4 GByte/s pro Link bereitstellt. Als MPI Bibliothek wurde das SGI MPT 1.16 Message Passing Toolkit verwendet und als Compiler kam der Intel Compiler 11.0 mit der höchsten Optimierungsstufe (`-O3`) zum Einsatz.

JuRoPA. Der JuRoPA (Jülich Research on Petaflop Architectures) Cluster besitzt 2208 Rechenknoten, die jeweils mit zwei Intel Xeon X5570 „Nehalem“ Quad-Core Prozessoren bestückt sind. Die Prozessoren erreichen bei einer Taktrate von 2.93 GHz eine maximale Rechenleistung von 11.72 GFlops/s pro Kern. Als Verbindungsnetzwerk wird ein Infiniband Netz mit vierfacher Datenrate (quad data rate, QDR) verwendet. Die für die Messungen benutzte Softwareumgebung besteht aus der ParaStation MPI Bibliothek von ParTec und dem Intel Compiler 11.0 mit der höchsten Optimierungsstufe (`-O3`).

Heterogener Cluster. Der heterogene Cluster verfügt über insgesamt 104 Rechenkerne, die auf die folgenden vier Knotentypen aufgeteilt sind:

- 48 Kerne AMD Opteron 865 „Egypt“ mit 1.8 GHz auf sechs Knoten mit je vier Dual-Core Prozessoren;

Tabelle 4: Überblick der wichtigsten Hardware- und Softwareparamter der Zielplattformen.

Plattform	CPU	Knoten	Aufbau Knoten	MPI	Netzwerk
CHic	AMD Opteron 2.6 GHz	530	2×2 Kerne	MVAPICH 1.0	Infiniband 10 GBit/s
SGI Altix	Intel Itanium2 1.6 GHz	128	2×2 Kerne	SGI MPT 1.16	NUMAlink4 6.4 GByte/s
JuRoPA	Intel Xeon 2.93 GHz	2208	2×4 Kerne	ParaStation MPI v5.0	Infiniband QDR
Heterogen	Opteron 1.8 GHz Opteron 1.9 GHz Opteron 2.0 Ghz Xeon 2.33 GHz	6 1 1 3	4×2 Kerne 4×4 Kerne 8×2 Kerne 2×4 Kerne	MVAPICH2 1.0.3	Infiniband 10 GBit/s

- 16 Kerne AMD Opteron 8347 „Barcelona“ mit 1.9 GHz auf einem Knoten mit vier Quad-Core Prozessoren;
- 16 Kerne AMD Opteron 870 „Egypt“ mit 2.0 GHz auf einem Knoten mit acht Dual-Core Prozessoren und
- 24 Kerne Intel Xeon E5345 „Clovertown“ mit 2.33 GHz auf drei Knoten mit je zwei Quad-Core Prozessoren.

Ein Infinibandnetzwerk mit einer Bandbreite von 10 GBit/s verbindet die einzelnen Rechenknoten und wird durch die MVAPICH2 1.0.3 MPI Bibliothek unterstützt. Die Übersetzung der Anwendungsprogramme erfolgte mit Hilfe des Intel Compilers 10.1 mit der höchsten Optimierungsstufe (`-O3`).

6.2. Anwendungsprogramme

In dieser Arbeit werden zwei Klassen von Anwendungen betrachtet, die sich aufgrund ihres modularen Aufbaus besonders gut für eine Implementierung im CM-task Programmiermodell eignen. Die erste Klasse umfasst numerische Lösungsverfahren für Systeme gewöhnlicher Differentialgleichungen, die in jedem Zeitschritt eine feste Anzahl von Stufenvektoren berechnen. In diesen Verfahren kann feinkörnige Parallelität durch eine parallele Berechnung der einzelnen Komponenten des Gleichungssystems und grobkörnige Parallelität durch eine parallele Berechnung der Stufenvektoren ausgenutzt werden. Als Beispiele dieser Klasse werden Iterierte Runge-Kutta Verfahren und Parallele Adams Verfahren betrachtet.

Die zweite Anwendungsklasse sind Löser für partielle Differentialgleichungen, deren Lösungsgebiet in lose gekoppelte Teilgebiete aufgeteilt ist. Daraus resultiert feinkörnige Parallelität innerhalb der Teilgebiete und grobkörnige Parallelität zwischen verschiedenen Teilgebieten.

Tabelle 5: Überblick der Eigenschaften der Anwendungsbenchmarks.

Name	Anzahl Basis-modul-aufrufe	Anzahl Daten-abhängig-keiten	Anzahl Datenum-verteilungs-operationen	Anzahl Kommuni-kationsab-hängigkeiten	Module pro Kommuni-kationsab-hänigkeit	Kommuni-kations-muster
IRK	$s+3$	$3s+8$	$2s$	1	s	orthogonal(VI)
PAB	$K+2$	$2K+5$	$3K-1$	1	K	orthogonal(VI)
PABM	$K+2$	$2K+5$	$3K-1$	1	K	orthogonal(VI)
LU-MZ	16	0	0	32	2	Gitter(IV)
SP-MZ	Z	0	0	$2Z$	2	Gitter(IV)
BT-MZ	Z	0	0	$2Z$	2	Gitter(IV)

Als Beispiele werden drei durch die NAS-MZ Benchmark Suite bereitgestellte Anwendungsbenchmarks verwendet. Im Folgenden werden die einzelnen Vertreter der beiden Klassen näher vorgestellt. Die wichtigsten Eigenschaften sind in Tabelle 5 zusammengefasst.

6.2.1. Löser für Systeme gewöhnlicher Differentialgleichungen

Iterierte Runge-Kutta (IRK) Verfahren [112, 93] werden in Kapitel 4 als Anwendungsbeispiel betrachtet. Das Spezifikationsprogramm ist auf Seite 50 abgebildet und in Abbildung 15 auf Seite 54 graphisch dargestellt. Der durch den CM-task Compiler berechnete Schedule ist in Abbildung 16 auf Seite 60 veranschaulicht und auf Seite 66 ist ein Ausschnitt des erzeugten statischen Koordinationsprogrammes abgebildet. Die verwendeten Basismodule sind mit Hilfe von MPI implementiert.

Parallele Adams Verfahren [111, 100] beinhalten die PAB und die PABM Methode, die in einem Zeitschritt jeweils K Stufenvektoren berechnen und durch CM-task Spezifikationsprogramme mit einer identischen Struktur beschrieben werden. Unterschiede zwischen den Methoden bestehen in den ausgeführten Rechen- und Kommunikationsoperationen innerhalb der Basismodule.

Abbildung 34 illustriert die Struktur des Spezifikationsprogrammes für die PABM Methode, das auf Seite 167 im Anhang abgebildet ist. Die Berechnung der K Stufenvektoren erfordert die zeitgleiche Ausführung von K Instanzen des Basismoduls `pabm_step`. Zwischen diesen Instanzen werden Teilergebnisse durch eine orthogonale Kommunikationsoperation ausgetauscht. Die Modulinstanzen besitzen identische Rechen- und Kommunikationskosten, wobei die modulinterne Kommunikation durch eine (PAB Methode) bzw. mehrere (PABM Methode) Multibroadcastoperationen (`MPI_Allgather`) dominiert wird. Für dieses Basismodul wird eine reine MPI Implementierung verwendet. Die Basismodule `init_step` und `update_step` initialisieren bzw. aktualisieren den aktuellen Zeitindex `x` und die Schrittweite `h`. Diese beiden Module beinhalten eine konstante Anzahl von Rechenoperationen und benötigen keine interne Kommunikation.

Die Schedulingphase des CM-task Compilers berechnet einen Schedule, der K disjunkte

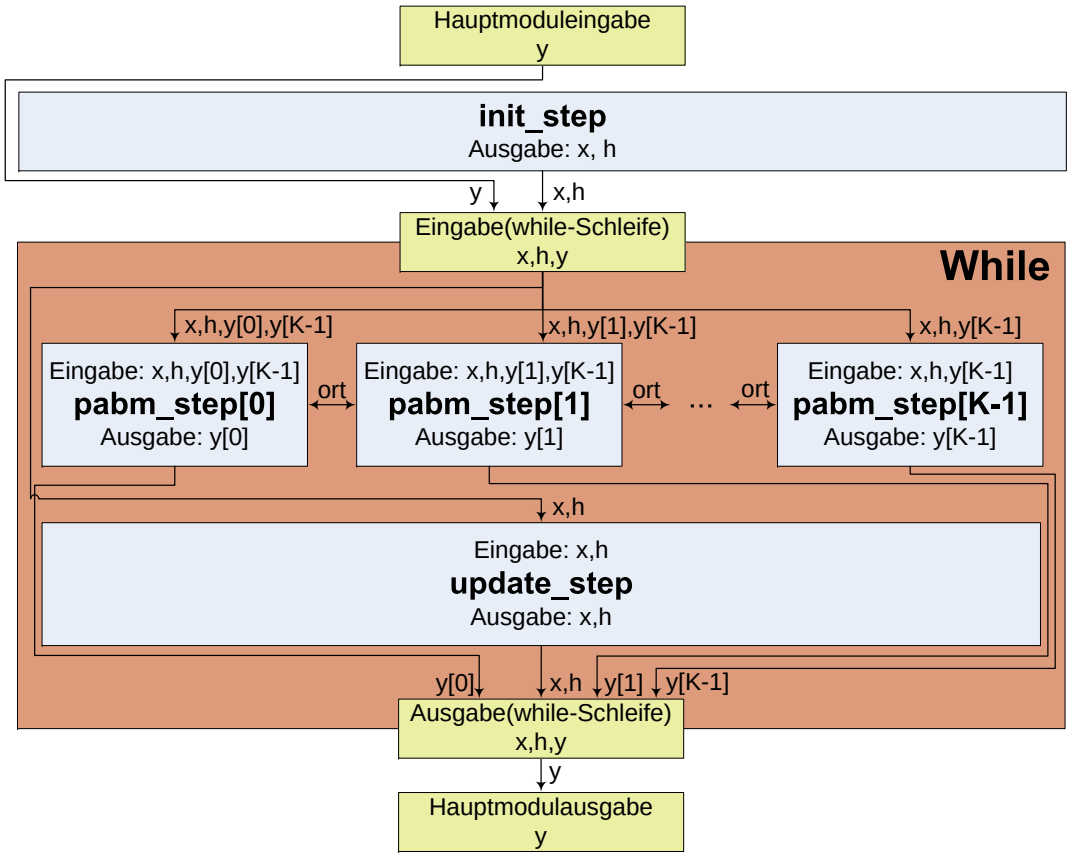


Abbildung 34: Illustration der Berechnungsstruktur Paralleler Adams Verfahren. Dem Hauptmodul werden `K` Stufenvektoren in der Variable `y` zur Verfügung gestellt. Das Basismodul `init_step` initialisiert den Zeitindex `x` und die Schrittweite `h`. Die `while`-Schleife berechnet die Zeitschritte des Verfahrens bis ein vorgegebener Zeitindex erreicht wird. Im Schleifenrumpf wird Stufenvektor `y[i]` durch den Basismodulaufruf `pabm_step[i] (i=0,...,K-1)` berechnet. Zwischen den `K` Aufrufen des Basismoduls `pabm_step` besteht eine Kommunikationsabhängigkeit, die durch den gemeinsamen Kommunikationsparameter `ort` spezifiziert ist. Am Ende eines Zeitschrittes erfolgt die Anpassung des Zeitindexes und der Schrittweite durch das Basismodul `update_step`. Die Ausgabe des Hauptmoduls besteht aus den `K` Stufenvektoren des letzten ausgeführten Zeitschrittes.

Prozessorgruppen gleicher Größe zur Ausführung der K Modulinstanzen von `pabm_step` verwendet. Die Datenverteilungsphase des CM-task Compilers legt für die `while`-Schleife eine Abspeicherung der K Stufenvektoren `y`$[i]$, $i = 1,\ldots,K$, auf jeweils einer der K Prozessorgruppen fest. Dadurch werden insgesamt $3K-1$ Datenumverteilungsoperationen benötigt. Davon werden $2K$ Operationen außerhalb der `while`-Schleife ausgeführt, um die K Stufenvektoren von der Hauptmoduleingabe zur Schleifeneingabe bzw. von der Schleifenausgabe zur Hauptmodulausgabe umzuverteilen. Diese Umverteilungen werden benötigt, da im Spezifikationsprogramm für die in das Hauptmodul eingegebenen bzw. die vom Hauptmodul

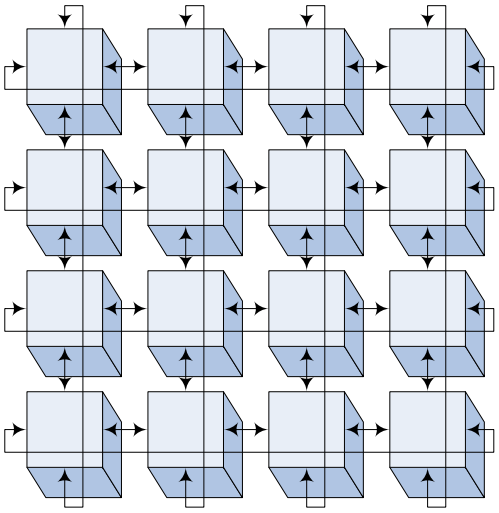


Abbildung 35: Aufteilung des globalen dreidimensionalen Lösungsgebietes in 4×4 Zonen und Veranschaulichung des Randwertaustausches durch Pfeile.

ausgegebenen Stufenvektoren eine Speicherung auf allen Prozessoren definiert ist. Innerhalb der `while`-Schleife werden $K-1$ Umverteilungen benötigt, um den Stufenvektor $\mathrm{y}[K-1]$ allen Prozessorgruppen zur Verfügung zu stellen.

6.2.2. NAS-MZ Benchmarks

Die NAS Parallel Multi-Zone Benchmark Suite (NAS-MZ) [113, 55] enthält synthetische Benchmarks, die die Berechnungs- und Kommunikationsstruktur realer Anwendungen aus dem Bereich der Strömungsmechanik nachbilden. Die Suite umfasst drei verschiedene Lösungsverfahren (LU-MZ, SP-MZ und BT-MZ) und stellt Referenzimplementierungen sowohl für einen gemeinsamen Speicher als auch für ein hybrides MPI+OpenMP Programmiermodell bereit. Für die folgenden Betrachtungen wurden modifizierte Programmvarianten erstellt, die auf einer reinen MPI Implementierung basieren.

Jeder der drei Benchmarks verwendet ein globales dreidimensionales Lösungsgebiert, das entlang zweier Koordinatenachsen in lose gekoppelte Teilgebiete aufgeteilt ist. Diese Teilgebiete werden im Folgenden als Zonen bezeichnet. Ein Zeitschritt eines Benchmarks umfasst die getrennte Berechnung einer Näherungslösung für jede Zone und einen anschließenden Randwertaustausch zwischen benachbarten Zonen. Abbildung 35 veranschaulicht die Zonenaufteilung und die resultierenden Kommunikationsoperationen zwischen den Zonen. Für jeden Benchmark existieren verschiedene Problemklassen (S, W, A, B, C, D, E, F), die sich in der Größe des Lösungsgebietes, der Anzahl der Zonen und der Anzahl der auszuführenden Zeitschritte unterscheiden.

Lower-Upper Symmetric Gauss-Seidel Multi-Zone (LU-MZ) Benchmark. Das Lösungsgebiet des LU-MZ Benchmarks wird in allen Benchmarkklassen in 16 gleichgroße Zonen aufgeteilt, die in einem 4×4 Gitter angeordnet sind. Durch diese Zonenaufteilung besitzen

Benchmarkklassen mit einem größeren Lösungsgebiet einen höheren Anteil an feinkörniger Parallelität innerhalb der Zonen, wohingegen der Grad grobkörniger Parallelität für alle Benchmarkklassen gleich ist.

Für die Realisierung des LU-MZ Benchmarks mit dem CM-task Compilerframework werden die Berechnungen einer Zone durch jeweils ein Basismodul ausgeführt. Diese Berechnungen umfassen die Initialisierung der Zone mit vorgegebenen Startwerten und die Ausführung der vorgegebenen Anzahl der Zeitschritte. Die Zeitschritte enthalten einen Randwertaustausch mit benachbarten Zonen, der durch externe Kommunikation realisiert ist. Daraus ergibt sich ein gitterbasiertes Kommunikationsmuster (Kommunikationsmuster IV, s. Abbildung 12 auf Seite 46), das eine gleichzeitige Ausführung aller 16 Modulinstanzen erfordert. Die Ausgabe der Basismodule besteht aus Kontrollwerten, wie der Abweichung der berechneten Näherungslösung von der vorgegebenen Referenzlösung.

Das resultierende Spezifikationsprogramm für den LU-MZ Benchmark ist im Anhang auf Seite 168 dargestellt. Die 16 Basismodulaufrufe werden durch zwei verschachtelte `cparfor`-Schleifen spezifiziert. Die für die Basismodulaufrufe verwendeten Kommunikationsparameter sind derart gewählt, dass ein gitterbasiertes Kommunikationsmuster entsteht. Die Kosten eines Basismoduls werden vereinfacht durch die Anzahl der Diskretisierungspunkte pro Prozessor abgeschätzt. Daraus ergeben sich identische Kostenfunktionen für alle Basismodulaufrufe, wodurch in der Schedulingphase des CM-task Compilers 16 gleichgroße Prozessorgruppen zur Ausführung der Basismodule bestimmt werden. Das durch den CM-task Compiler erzeugte Koordinationsprogramm enthält keine Datenumverteilungsoperationen, da die notwendigen Kommunikationsoperationen zwischen den Prozessorgruppen bereits in die Basismodule integriert sind.

Scalar Pentadiagonal Multi-Zone (SP-MZ) Benchmark. Der SP-MZ Benchmark teilt das globale Lösungsgebiet in z gleichgroße Zonen auf, die in einem $\sqrt{z} \times \sqrt{z}$ Gitter angeordnet sind. Der konkrete Wert von z hängt von der Benchmarkklasse ab und liegt zwischen 4 für Klasse S und 16384 für Klasse F. Benchmarkklassen mit einem größeren Lösungsgebiet definieren eine höhere Zonenanzahl, so dass die einzelnen Zonen in allen Klassen ungefähr gleichgroß sind.

Aufgrund der hohen Zonenanzahl in einigen Benchmarkklassen verwendet die Realisierung des SP-MZ Bechmarks mit dem CM-task Compilerframework Basismodule, die eine konfigurierbare Anzahl c benachbarter Zonen berechnen. Der Parameter c wird vom Anwender definiert und ist für alle ausgeführten Basismodule identisch. Ein Wert von $c = z$ führt zu einer rein datenparallelen Ausführung, wohingegen durch $c = 1$ die maximal vorhandene Taskparallelität ausgenutzt wird.

In einem Zeitschritt berechnet jedes Basismodul nacheinander die Näherungslösung für die c zugewiesenen Zonen und führt anschließend den Randwertaustausch aus. Der Randwertaustausch eines Basismoduls umfasst sowohl interne Kommunikation zum Datenaustausch zwischen den c lokalen Zonen des jeweiligen Basismoduls als auch externe Kommunikation mit denjenigen Basismodulen, die die an die lokalen Zonen angrenzenden Zonen berechnen. Insgesamt werden $Z = \frac{z}{c}$ Basismodule zeitgleich ausgeführt, zwischen denen aufgrund des Randwertaustausches ein gitterbasiertes Kommunikationsmuster (Kommunikationsmuster IV, s. Abbildung 12 auf Seite 46) entsteht. Die Ausgabe der Basismodule besteht analog zum LU-MZ Benchmark aus Kontrollwerten.

Das Spezifikationsprogramm des SP-MZ Benchmarks ist im Anhang auf Seite 169 abgebildet.

Die Z Basismodulaufrufe werden analog zum LU-MZ Benchmark durch zwei verschachtelte `cparfor`-Schleifen spezifiziert. Der Iterationsraum dieser Schleifen ist im Gegensatz zum LU-MZ Benchmark von der Gesamtzonenanzahl und der Anzahl der Zonen pro Basismodul abhängig. Die spezifizierten Kostenwerte für die Basismodule ergeben sich wie im LU-MZ Benchmark aus der Anzahl der Diskretisierungspunkte pro Prozessor. Da die Kostenwerte für alle Z Basismodulaufrufe identisch sind, werden in der Schedulingphase des CM-task Compilers Z gleichgroße Prozessorgruppen zur Ausführung der Module festgelegt. Analog zum LU-MZ Benchmark werden keine Datenumverteilungsoperationen benötigt.

Block Tridiagonal Multi-Zone (BT-MZ) Benchmark. Die Zonenanzahl z im BT-MZ Benchmark ist wie im SP-MZ Benchmark abhängig von der Benchmarkklasse. Die Zonen des BT-MZ Benchmarks besitzen jedoch unterschiedliche Größen, wobei die größte Zone ungefähr die zwanzigfache Anzahl der Diskretisierungspunkte der kleinsten Zone besitzt.

Die Realisierung des BT-MZ Benchmarks im CM-task Compilerframework erfolgt analog zum SP-MZ Benchmark, d.h. ein Basismodul berechnet eine manuell festgelegte Anzahl benachbarter Zonen c. Das zugehörige Spezifikationsprogramm ist im Anhang auf Seite 170 dargestellt. Unterschiede gegenüber dem SP-MZ Benchmark bestehen nur bei der Spezifikation der Kostenwerte, da aufgrund der unterschiedlichen Zonengrößen unterschiedliche Kosten für die $Z = \frac{z}{c}$ zeitgleich auszuführenden Basismodulaufrufe entstehen. Als Kostenwerte werden wie für die LU-MZ und SP-MZ Benchmarks die Anzahl der Diskretisierungspunkte pro Prozessor verwendet. Die spezifizierten Kostenwerte bewirken, dass die Schedulingphase des CM-task Compilers Z disjunkte Prozessorgruppen unterschiedlicher Größe bestimmt. Die Gruppengröße wird durch den Lastbalancierungsschritt des CM-task Schedulingalgorithmus derart festgelegt, dass jeder Prozessor ungefähr die gleiche Anzahl an Diskretisierungspunkten zu berechnen hat. Dies bedeutet, dass Basismodule mit einer höheren Anzahl an Diskretisierungspunkten durch Prozessorgruppen mit höherer Prozessoranzahl berechnet werden.

6.3. Auswertung der Experimente

Im Folgenden werden die Resultate von Laufzeitmessungen vorgestellt, die unter unterschiedlichen Prämissen durchgeführt wurden. In Teilabschnitt 6.3.1 wird zunächst die Leistungsfähigkeit statischer Koordinationsprogramme untersucht. Semi-dynamische Koordinationsprogramme werden anschließend in Teilabschnitt 6.3.2 evaluiert. Teilabschnitt 6.3.3 bewertet verschiedene Mappingstrategien zur Ausführung statischer Koordinationsprogramme auf Multicore-Systemen.

6.3.1. Evaluierung des statischen Compileransatzes

Iterierte Runge-Kutta Verfahren

Die Laufzeituntersuchungen für IRK Verfahren vergleichen die folgenden vier parallelen Implementierungen.

- Die **handgeschriebene datenparallele** Implementierung (s. Abbildung 36 (a)) berechnet alle s Stufenvektoren nacheinander auf allen verfügbaren Prozessoren. Diese Programmversion besitzt den Vorteil, dass keine Datenumverteilungsoperationen zwischen den einzelnen Stufenvektorberechnungen benötigt werden. Der Nachteil liegt im hohen Anteil

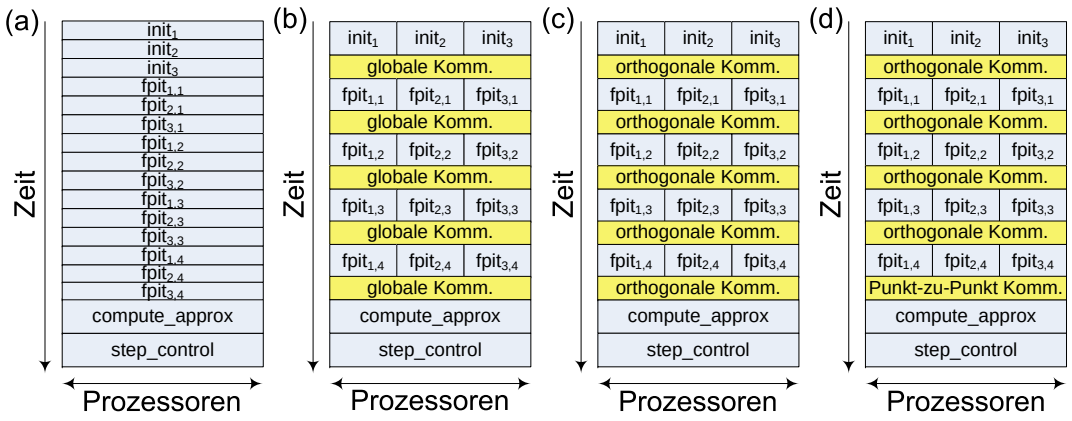


Abbildung 36: Illustration des Ablaufs eines Zeitschrittes der datenparallelen (a), der taskparallelen (b), der handgeschriebenen orthogonalen (c) und der generierten orthogonalen Programmversion (d) für Iterierte Runge-Kutta Verfahren. Die Berechnung der $s = 3$ Stufenvektoren besteht aus einem Initialisierungsschritt $init_i$ und $m = 4$ Fixpunkiterationsschritten $fpit_{i,j}$ ($i = 1,2,3; j = 1,\ldots,4$). Anschließend wird der Näherungsvektor aktualisiert ($compute\_approx$) und die Schrittweitenkontrolle durchgeführt ($step\_control$).

globaler Kommunikationsoperationen, die zur Berechnung der Stufenvektoren benötigt werden.

- Die **handgeschriebene taskparallele** Programmversion (s. Abbildung 36 (b)) repräsentiert eine Implementierung, wie sie typisch für Programmiermodelle auf Basis paralleler Tasks ist. Die Berechnung der Stufenvektoren erfolgt durch s disjunkte Prozessorgruppen gleicher Größe. Damit können die Kommunikationsoperationen, die zur Berechnung eines Stufenvektors erforderlich sind, auf Teilmengen der Prozessoren beschränkt werden. Im Vergleich zur datenparallelen Version werden jedoch zusätzliche Kommunikationsoperationen zum Datenaustausch zwischen den Prozessorgruppen benötigt. Diese Kommunikationsoperationen sind durch globale Kommunikationsphasen realisiert.

- Die **handgeschriebene orthogonale** Programmversion (s. Abbildung 36 (c)) repräsentiert eine Implementierung, wie sie das CM-task Programmiermodell ermöglicht. Analog zur taskparallelen Version werden s disjunkte Prozessorgruppen verwendet und die Kommunikationsoperationen zur Stufenvektorberechnung auf Teilmengen von Prozessoren beschränkt. Der erforderliche Datenaustausch zwischen den Prozessorgruppen ist durch ein orthogonales Kommunikationsmuster (s. Abbildung 14 auf Seite 48) realisiert, so dass diese Kommunikationsoperationen ebenfalls auf Teilmengen von Prozessoren ausgeführt werden und der Anteil globaler Kommunikation im Vergleich zur taskparallelen Implementierung verringert wird.

- Die **generierte orthogonale** Programmversion (s. Abbildung 36 (d)) setzt sich zusammen aus dem vom CM-task Compiler erzeugten statischen Koordinationsprogramm und den entsprechenden Basismodulimplementierungen. Die Prozessorgruppenaufteilung und die verwendeten Kommunikationsoperationen zur Berechnung der Stufenvektoren

entsprechen der handgeschriebenen orthogonalen Implementierung. Unterschiede bestehen am Ende eines Zeitschrittes bei der Aktualisierung des Näherungsvektors. Die handgeschriebene Version stellt die erforderliche Datenverteilung durch orthogonale Kommunikationsoperationen her, wohingegen die generierte Version die Punkt-zu-Punkt Kommunikationsoperationen der Datenumverteilungsbibliothek des CM-task Compilerframeworks verwendet.

Für die Laufzeitmessungen werden zwei Differentialgleichungssysteme mit unterschiedlichen Berechnungseigenschaften verwendet. Die zweidimensionale Brusselator-Gleichung mit Diffusion (BRUSS2D) [44] ist dünn besetzt, d.h. die Berechnung einer Komponente des Systems benötigt eine feste Anzahl arithmetischer Operationen. Als dicht besetztes System wird ein Schrödinger-Poisson-System (SCHROED) [89] verwendet. Die Auswertungszeit einer Komponente dieses Systems ist linear abhängig von der Dimension des Gleichungssystems.

Abbildung 37 zeigt eine Auswahl der Messergebnisse für ein IRK Verfahren, das $s = 4$ Stufenvektoren mit $m = 6$ Fixpunktiterationsschritten berechnet. Die Messungen zeigen deutliche Leistungsunterschiede zwischen einer datenparallelen, einer taskparallelen und einer orthogonalen Implementierung. Diese Unterschiede sind für das dünn besetzte Brusselator-System aufgrund des hohen Kommunikationsanteils an der Gesamtlaufzeit besonders ausgeprägt.

Im Vergleich zu einer rein datenparallelen Realisierung schneidet die taskparallele Implementierung aufgrund der zusätzlichen Kommunikationsoperationen zum Austausch der Stufenvektoren in den meisten Fällen schlechter ab. Eine gemischt parallele Ausführung kann erst durch Ausnutzung orthogonaler Kommunikationsmuster für den Datenaustausch zwischen den Prozessorgruppen einen deutlichen Leistungsgewinn gegenüber einer datenparallelen Ausführung erzielen. Beispielsweise kann die Ausführungszeit für das dünn besetzte System gegenüber der datenparallelen Version um den Faktor fünf (CHiC Cluster) bzw. 2.7 (JuRoPA Cluster) reduziert werden. Die Laufzeitreduktion für den JuRoPA Cluster fällt geringer aus, da aufgrund der höheren Rechenkernanzahl pro Knoten und des schnelleren Verbindungsnetzwerkes die globalen Kommunikationsoperationen besser skalieren.

Der Overhead der generierten Programmversion gegenüber der handgeschriebenen orthogonalen Version liegt sowohl auf dem CHiC Cluster als auch auf dem JuRoPA Cluster zwischen einem und zwei Prozent. Auf der SGI Altix sind durch die generierte Version für nahezu alle getesteten Prozessoranzahlen sogar minimal höhere Speedupwerte möglich. Dieses Verhalten lässt auf eine höhere Leistungsfähigkeit der in der generierten Version verwendeten Punkt-zu-Punkt Kommunikationsoperationen im Vergleich zu den kollektiven Kommunikationsoperationen der handgeschriebenen Variante schließen.

Zusammen belegen die Resultate sowohl den Vorteil orthogonaler Realisierungen als auch den geringen Koordinationsoverhead generierter Programmversionen und der darin enthaltenen Datenumverteilungsoperationen.

Parallele Adams Verfahren

Für Parallele Adams Verfahren werden die folgenden Programmversionen verwendet, die analog zu den IRK Verfahren definiert sind.

- Die **handgeschriebene datenparallele** Programmversion berechnet nacheinander K Stufenvektoren gemeinsam durch alle Prozessoren und verwendet ausschließlich globale kollektive Kommunikationsoperationen.

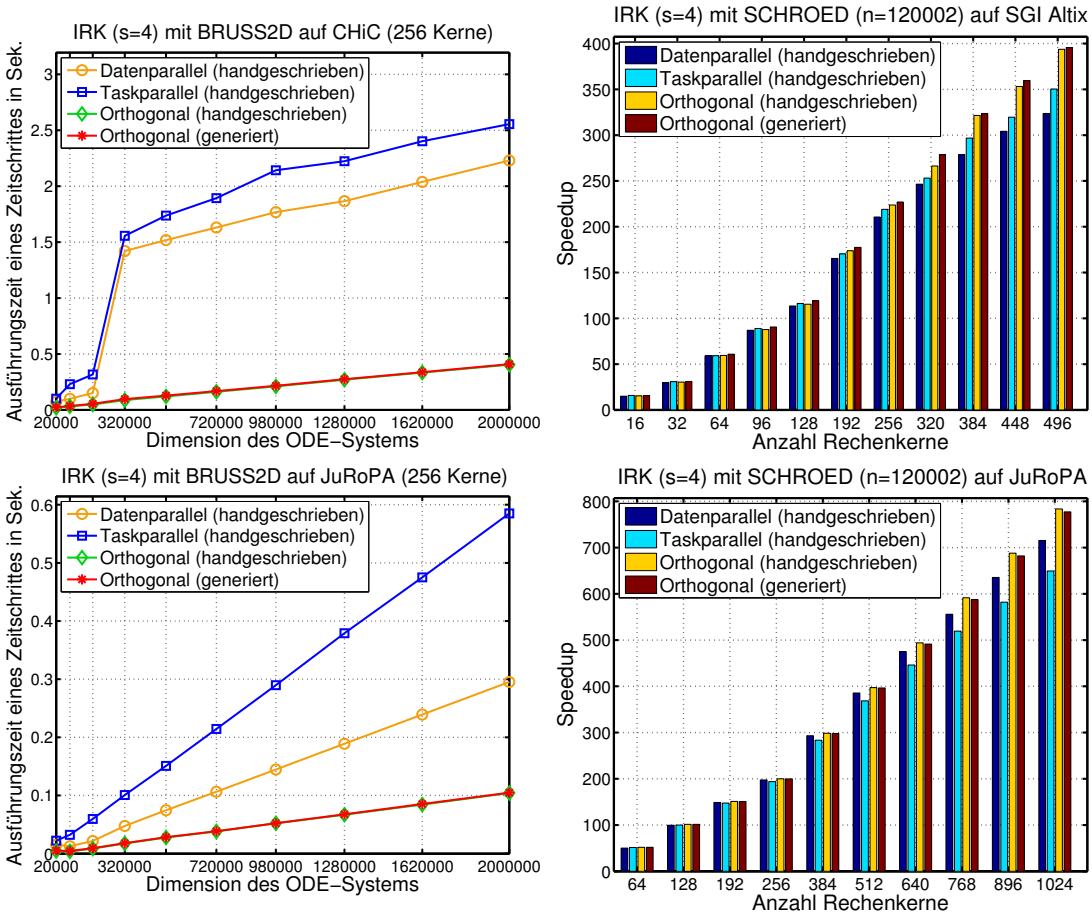


Abbildung 37: Messergebnisse für einen Zeitschritt eines Iterierten Runge-Kutta Verfahrens mit $s = 4$ Stufenvektoren. Die linke Spalte zeigt die Ausführungszeiten für verschiedene Dimensionen eines dünn besetzten Systems auf 256 Rechenkernen des CHiC Clusters (oben links) und des JuRoPA Clusters (unten links). Die handgeschriebene und die generierte orthogonale Programmversion erzielen für diese beiden Messungen nahezu identische Ausführungszeiten. Die rechte Spalte zeigt die Speedupwerte für ein dicht besetztes System abhängig von der Prozessoranzahl für die SGI Altix (oben rechts) und den JuRoPA Cluster (unten rechts).

- Die **handgeschriebene taskparallele** Programmversion verwendet K gleich große Prozessorgruppen zur zeitgleichen Berechnung von K Stufenvektoren. Der Datenaustausch zwischen den Prozessorgruppen ist durch globale Kommunikationsoperationen realisiert.
- Die **handgeschriebene orthogonale** Realisierung berechnet die K Stufenvektoren auf K gleich großen Prozessorgruppen und verwendet orthogonale Kommunikation zum Datenaustausch zwischen den Prozessorgruppen.
- Die **generierte orthogonale** Programmversion entspricht dem vom CM-task Compiler erzeugten statischen Koordinationsprogramm. Die Berechnungen werden analog zur

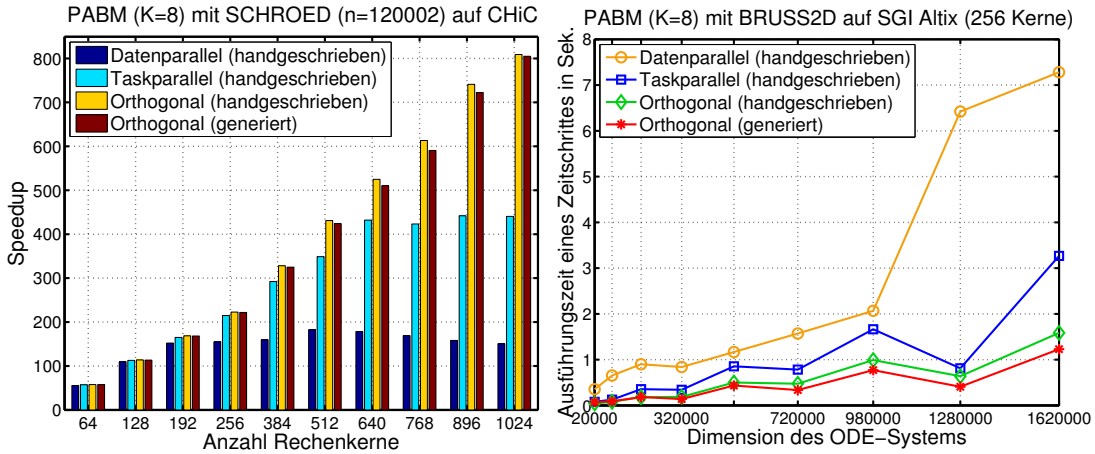


Abbildung 38: Messergebnisse der PABM Methode mit $K = 8$ Stufenvektoren. Das linke Diagramm zeigt die Speedupwerte eines dicht besetzten Systems für verschiedene Prozessorzahlen des CHiC Clusters. Die Ausführungszeiten eines Zeitschrittes für ein dünn besetztes System auf 256 Rechenkernen der SGI Altix sind im rechten Diagramm dargestellt.

handgeschriebenen orthogonalen Version auf K disjunkten Prozessorgruppen ausgeführt und der erforderliche Datenaustausch zur Stufenvektorberechnung erfolgt durch ein orthogonales Kommunikationsmuster. Der Austausch des berechneten Näherungsvektors am Ende eines Zeitschrittes erfolgt durch die Datenumverteilungsbibliothek des CM-task Compilerframeworks.

Abbildung 38 zeigt die Speedupwerte für ein dicht besetztes System auf dem CHiC Cluster (links) und die Ausführungszeiten eines Zeitschrittes für ein dünn besetztes System auf der SGI Altix (rechts) unter Verwendung der PABM Methode mit $K = 8$ Stufenvektoren. Im Gegensatz zu den IRK Verfahren ist schon eine einfache taskparallele Implementierung der datenparallelen Version deutlich überlegen. Dieses Verhalten kann mit dem höheren Anteil gruppenbasierter Kommunikationsoperationen und der höheren Prozessorgruppenanzahl in der PABM Methode erklärt werden. Durch eine orthogonale Realisierung werden analog zu den IRK Verfahren signifikante Leistungsvorteile erzielt. Insbesondere die Speedupwerte auf dem CHiC Cluster belegen, dass nur durch die orthogonale Programmversion eine Skalierung bis zu 1024 Rechenkerne erreicht werden kann.

Der Overhead der generierten Programmversion beträgt auf dem CHiC Cluster bis zu vier Prozent bei 768 Rechenkerne. Für höhere Prozessorzahlen nimmt der Overhead ab und liegt bei unter einem Prozent bei 1024 Rechenkernen. Auf der SGI Altix werden durch die generierte Programmversion wie schon bei den IRK Verfahren geringere Laufzeiten aufgrund der verwendeten Punkt-zu-Punkt Kommunikationsoperationen erzielt.

LU-MZ Benchmark

Für die Leistungsmessungen des LU-MZ Benchmarks werden die folgenden Programmversionen verwendet.

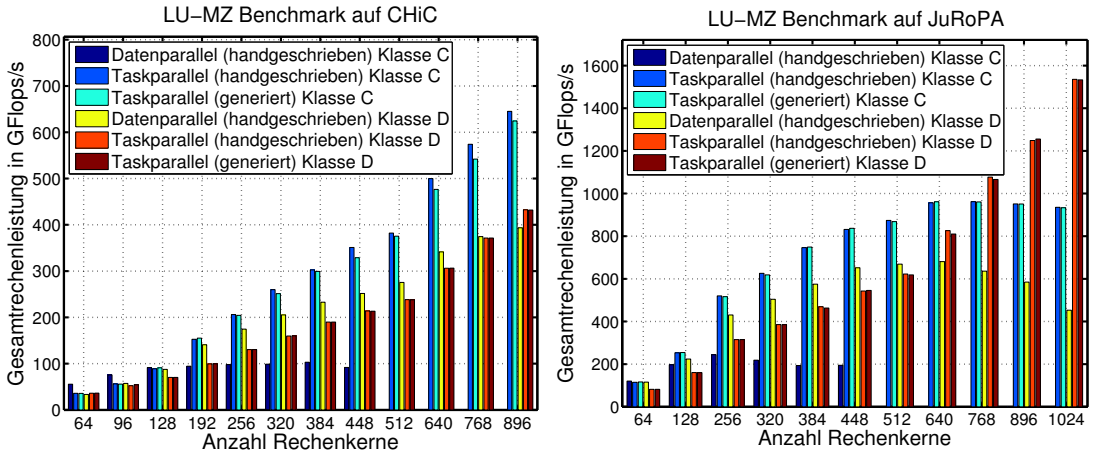


Abbildung 39: Leistungswerte für verschiedene Programmversionen des LU-MZ Benchmarks für die Benchmarkklassen C und D auf dem CHiC Cluster (links) und auf dem JuRoPA Cluster (rechts).

- Die **handgeschriebene datenparallele** Programmversion berechnet 16 Zonen nacheinander auf allen verfügbaren Prozessoren. Die Datenverteilung innerhalb der Zonen ist derart festgelegt, dass für den Randwertaustausch keine Kommunikationsoperationen benötigt werden.

- Die **handgeschriebene taskparallele** Programmversion bildet 16 gleichgroße Prozessorgruppen, die jeweils eine Zone berechnen. Am Ende jedes Zeitschrittes erfolgt der Randwertaustausch zwischen benachbarten Zonen durch direkte Punkt-zu-Punkt Kommunikationsoperationen zwischen den entsprechenden Prozessoren.

- Die **generierte taskparallele** Programmversion entspricht dem durch den CM-task Compiler generierten statischen Koordinationsprogramm. Die Aufteilung der Berechnungen und die verwendeten Kommunikationsoperationen sind identisch mit der handgeschriebenen taskparallelen Programmversion.

Abbildung 39 zeigt die vom LU-MZ Benchmark ausgegebenen Leistungswerte in GFlops/s auf dem CHiC Cluster (links) und auf dem JuRoPA Cluster (rechts). Die Messungen verwenden die Benchmarkklassen C und D, die jeweils $z = 16$ Zonen der Größe $120 \times 80 \times 28$ bzw. $408 \times 304 \times 34$ besitzen. Klasse C enthält nicht genügend Parallelität, um die datenparallele Programmversion auf mehr als 448 Prozessoren abzuarbeiten. Die erzielten Leistungswerte sind für den JuRoPA Cluster aufgrund der schnelleren Prozessoren und des leistungsfähigeren Verbindungsnetzwerkes deutlich höher als für den CHiC Cluster.

Auf beiden Plattformen ist die datenparallele Programmversion den anderen Implementierungen bei einer niedrigen Prozessoranzahl überlegen, da keine Kommunikationsoperationen für den Randwertaustausch anfallen und die Speicherhierarchie aufgrund der geringeren Datenmenge, die jeder Prozessor für eine bestimmte Zone berechnen muss, besser ausgenutzt werden kann. Mit wachsender Prozessoranzahl treten in dieser Programmversion aber Skalierbarkeitsprobleme aufgrund der steigenden Kommunikations- und Synchronisationszeiten innerhalb der Zonen auf. Im Gegensatz dazu werden diese Probleme bei den taskparallelen

Programmversionen reduziert und dadurch insbesondere bei hohen Prozessorzahlen deutlich höhere Leistungswerte erzielt. Beispielsweise erzielen die taskparallelen Versionen auf dem JuRoPA Cluster mit 1024 Prozessorkernen die mehr als dreifache Leistung einer datenparallelen Realisierung.

Die Leistungsunterschiede zwischen handgeschriebener und generierter Programmversion betragen für Benchmarkklasse *C* maximal fünf Prozent, während für Klasse *D* beide Versionen eine nahezu identische Leistung erzielen. Die Unterschiede resultieren einerseits aus dem generierten Koordinationscode zur Verwaltung der Prozessorgruppen und andererseits aus der Ermittlung der benötigten Kommunikationsoperationen zum Randwertaustausch. Während in der handgeschriebenen Version alle für die Kommunikation benötigten Informationen bei Programmstart außerhalb der Messungen ermittelt werden, muss dieser Berechnungsschritt in der generierten Variante innerhalb der Basismodule erfolgen, da erst bei Ausführung der Basismodule die genaue Prozessorgruppenaufteilung des generierten Koordinationsprogrammes bekannt ist.

SP-MZ Benchmark

Die Laufzeitexperimente für den SP-MZ Benchmark verwenden Benchmarkklasse *C* mit $z = 256$ Zonen der Größe $30 \times 20 \times 28$ und Benchmarkklasse *D* mit $z = 1024$ Zonen der Größe $51 \times 38 \times 34$. Als Programmversionen werden eine handgeschriebene datenparallele Implementierung, die in einem Zeitschritt alle z Zonen nacheinander berechnet, und verschiedene generierte Koordinationsprogramme verwendet. Diese Koordinationsprogramme werden durch den statischen Ansatz des CM-task Compilers erzeugt und unterscheiden sich in der Anzahl der zeitgleich ausgeführten Basismodule *Z*. Im Folgenden wird auf die Verteilung der Daten einer Zone auf die p die jeweilige Zone ausführenden Prozessoren eingegangen. Diese Datenverteilung ist für alle Programmversionen gleich und beeinflusst die sich ergebenden Wartezeiten und die maximale Prozessoranzahl pro Zone.

Das Lösungsverfahren im SP-MZ Benchmark basiert auf einem ADI (alternating direction implicit) Verfahren, das in jedem Zeitschritt aus mehreren Phasen besteht. In jeder dieser Phasen bestehen Datenabhängigkeiten zwischen den Diskretisierungspunkten entlang einer der drei Koordinatenachsen, so dass die in dieser Dimension benachbarten Punkte nacheinander berechnet werden müssen. Dagegen sind die Diskretisierungspunkte entlang der beiden anderen Koordinatenachsen unabhängig voneinander und können parallel berechnet werden. Als Datenverteilung für derartige Verfahren ist eine Multipartitionierung [13] geeignet. Dabei wird für eine quadratische Prozessoranzahl p das Lösungsgebiet einer Zone in $\sqrt{p} \times \sqrt{p} \times \sqrt{p}$ gleichgroße quaderförmige Zellen aufgeteilt. Jedem Prozessor werden $\sqrt{p}$ Zellen derart zugeteilt, dass der jeweilige Prozessor in jeder der drei Koordinatenachsen für jede der $\sqrt{p}$ Zellenkoordinaten genau eine Zelle besitzt. Durch diese Anordnung können Wartezeiten in jeder der Phasen des Algorithmus verhindert werden.

Für nichtquadratische Prozessorzahlen wurde in [16] eine Erweiterung vorgeschlagen, die ebenfalls Wartezeiten in jeder der Algorithmenphasen verhindern kann. Dieser Ansatz benötigt jedoch die Aufteilung einer Zone in eine hohe Anzahl von Zellen und ist deshalb für die im SP-MZ Benchmark vorkommenden geringen Zonengrößen nicht geeignet. Daher wird ein anderer Ansatz gewählt. Das Lösungsgebiet wird in $p_1 \times p_2 \times p_2$ Zellen mit $p_1 p_2 = p$ und $p_1 > p_2$ aufgeteilt. Jedem Prozessor werden p_2 Zellen zugeteilt, so dass der jeweilige Prozessor für jede Zellenkoordinate in y- und in z-Richtung des Koordinatensystems genau eine Zelle besitzt.

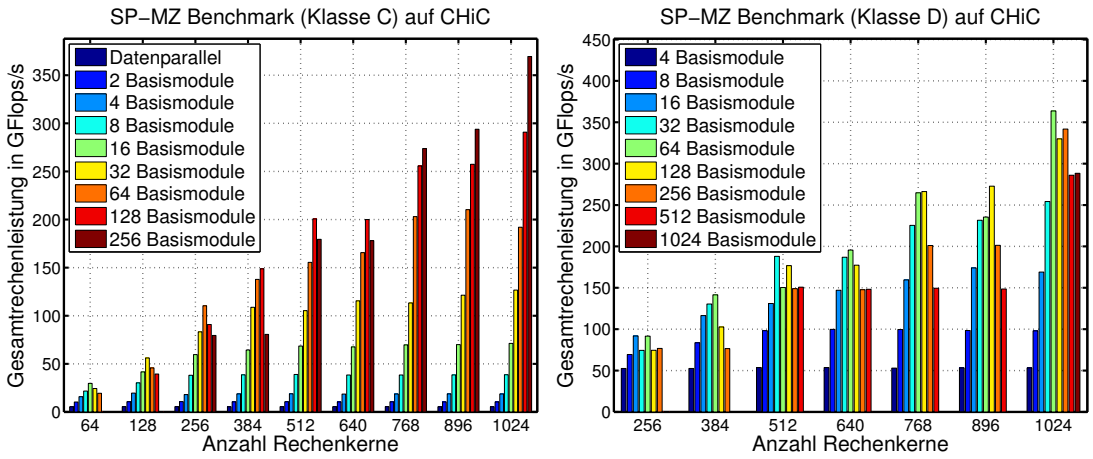


Abbildung 40: Leistungswerte des SP-MZ Benchmarks für eine datenparallele Programmversion und generierte taskparallele Programmversionen mit unterschiedlicher Anzahl zeitgleich ausgeführter Basismodule auf dem CHiC Cluster für Benchmarkklasse C (links) und Benchmarkklasse D (rechts).

In x-Richtung des Koordinatensystems ist eine derartige Zuteilung nicht möglich, da $p_1 > p_2$ Zellenkoordinaten vorhanden sind. Daher entstehen Wartezeiten in den Algorithmenphasen, die entlang der x-Richtung ausgeführt werden.

Aus der verwendeten Datenverteilung und der vom Benchmark definierten Zonengröße ergibt sich eine maximale Prozessoranzahl pro Zone. Die durch die Datenverteilung erzeugten Zellen müssen mindestens drei Diskretisierungspunkte in jeder der Koordinatenachsen besitzen. Daraus folgt, dass eine Zone von Benchmarkklasse C in maximal $10 \times 6 \times 6$ Zellen unterteilt werden kann und somit maximal $p = 60$ Prozessoren ausgenutzt werden. Für Benchmarkklasse D können $17 \times 11 \times 11$ Zellen gebildet werden, so dass $p = 187$ Prozessoren ausgenutzt werden. In den im Folgenden vorgestellten Messungen blieben überzählige Rechenkerne untätig.

Abbildung 40 zeigt die gemessenen Leistungswerte für die Programmversionen des SP-MZ Benchmarks auf dem CHiC Cluster. Für Benchmarkklasse D sind nur Resultate für die taskparallele Versionen mit mindestens vier CM-tasks angegeben, da auf den Knoten des CHiC Clusters nicht genügend Hauptspeicher zur Ausführung der datenparallelen und der taskparallelen Version mit zwei Basismodulen zur Verfügung stand.

Die Resultate zeigen für beide Benchmarkklassen, dass eine datenparallele Ausführung und die taskparallelen Versionen mit einer niedrigen Anzahl von Basismodulen nicht konkurrenzfähig sind. Dieses Verhalten resultiert aus der geringen Größe der Zellen und dem damit verbundenen häufigen Datenaustausch zwischen den einer Zone zugewiesenen Prozessoren.

Für Benchmarkklasse C gilt, dass bei höheren Prozessorzahlen eine höhere Anzahl von Basismodulen zur Erreichung der besten Leistung benötigt wird. Auf 256 Prozessoren werden die höchsten Leistungswerte durch 64 Basismodule erreicht, während die Programmversion mit 128 Basismodulen in den Messungen für 384, 512 und 640 Rechenkerne überlegen ist. Erst ab 768 Kernen führt die Ausnutzung maximaler Taskparallelität durch 256 Basismodulen zu den höchsten Leistungswerten.

Für die Benchmarkklasse D ergibt sich ein unregelmäßigeres Bild. Auf 1024 Rechenkernen erreichen die Programmversionen mit 64 und 256 Basismodulen die höchste Rechenleistung.

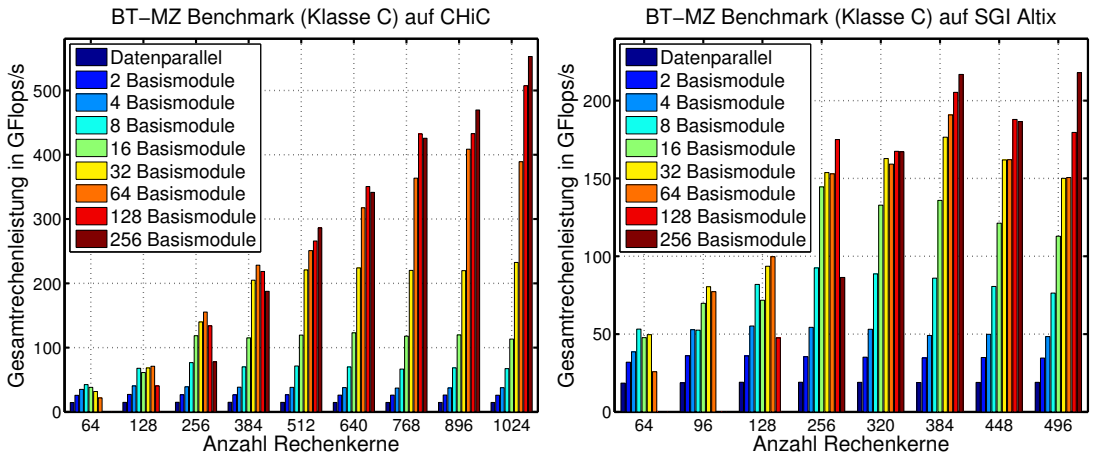


Abbildung 41: Leistungswerte für Benchmarkklasse C des BT-MZ Benchmarks für eine datenparallele Programmversion und generierte taskparallele Programmversionen mit unterschiedlicher Anzahl zeitgleich ausgeführter Basismodule auf dem CHiC Cluster (links) und auf der SGI Altix (rechts).

In diesen Programmversionen wird jede Zone durch 16 bzw. durch vier Rechenkerne berechnet, so dass aufgrund der quadratischen Prozessoranzahl eine Multipartitionierung ohne zusätzliche Wartezeiten verwendet wird. Aus analogen Gründen erzielen auf 512 Rechenkernen die Versionen mit 32 und 128 Basismodulen die höchsten Leistungswerte.

Insgesamt liegen die gemessenen Leistungswerte deutlich unter denen des LU-MZ Benchmarks. Dieses Verhalten kann vor allem durch die deutlich höhere Zonenanzahl erklärt werden. Daraus ergibt sich sowohl ein höherer Verwaltungsaufwand für die Zonen als auch eine höhere Anzahl benötigter Kommunikationsoperationen für den Randwertaustausch zwischen den Zonen.

BT-MZ Benchmark

Die Laufzeitmessungen für den BT-MZ Benchmark vergleichen analog zum SP-MZ Benchmark eine handgeschriebene datenparallele Ausführung mit vom CM-task Compiler erzeugten Koordinationsprogrammen. Dafür wird im Folgenden die Benchmarkklasse C mit $z = 256$ Zonen betrachtet. Die Zonen besitzen im BT-MZ Benchmark unterschiedliche Größen, wobei die kleinste Zone $13 \times 8 \times 28$ und die größte Zone $57 \times 38 \times 28$ Diskretisierungspunkte enthält.

Der BT-MZ Benchmark verwendet dieselbe Datenverteilung innerhalb der Zonen wie der SP-MZ Benchmark. Daraus folgt, dass die kleinste Zone durch maximal acht Prozessoren ($4 \times 2 \times 2$ Zellen, zwei Zellen pro Prozessor) und die größte Zone durch maximal 171 Prozessoren ($19 \times 9 \times 9$ Zellen, neun Zellen pro Prozessor) berechnet werden kann. Diese Beschränkungen werden durch die im Spezifikationsprogramm verwendeten Kostenfunktionen für die Basismodule nicht erfasst, so dass durch den CM-task Compiler auch höhere Prozessoranzahlen zugewiesen werden können. Die überzähligen Prozessoren bleiben in diesem Fall untätig.

Abbildung 41 vergleicht die Resultate einer datenparallelen Realisierung mit taskparallelen Implementierungen mit unterschiedlicher Basismodulanzahl Z. Als Plattformen dienen der CHiC Cluster (links) und die SGI Altix (rechts). Die datenparallele Programmversion und die

taskparallelen Versionen mit niedriger Basismodulanzahl erreichen nur niedrige Leistungswerte aufgrund der begrenzten Anzahl ausnutzbarer Prozessoren und der geringen Größe der erzeugten Zellen.

Die Versionen mit einer hohen Basismodulanzahl zeigen speziell auf dem CHiC Cluster eine sehr gute Skalierbarkeit der Leistungswerte. Beispielsweise erhöht sich die Rechenleistung der Programmversion mit $Z = 256$ Basismodulen bei einer Verdopplung der Rechenkernanzahl von 256 auf 512 um den Faktor 3.7. Auf der SGI Altix wird ebenfalls eine deutliche Leistungssteigerung dieser Programmversion bei Erhöhung der Rechenkernanzahl von 256 auf 384 erreicht.

Dieses Verhalten resultiert aus den durch die unterschiedliche Zonengrößen entstehenden Lastungleichgewichten zwischen den verwendeten Prozessorgruppen. Beispielsweise wird für die Version mit $Z = 256$ Basismodulen bei einer Ausführung auf 256 Kernen je ein Rechenkern pro Basismodul verwendet. In diesem Fall wird die Ausführungszeit eines Zeitschrittes des Benchmarks durch die Ausführungszeit der größten Zone auf einem Rechenkern bestimmt. Für eine Ausführung dieser Programmversion auf 512 Kernen werden durch die Schedulingphase des CM-task Compilers Prozessorgruppen unterschiedlicher Größe definiert. Die kleinste Zone wird in diesem Fall durch einen einzelnen Kern berechnet, wohingegen für die größte Zone sechs Rechenkerne verwendet werden. Dadurch kann vor allem die für die größte Zone benötigte Ausführungszeit deutlich reduziert werden.

Die beobachteten Leistungssteigerungen für diese Programmversionen zeigen, dass durch die Wahl geeigneter Prozessorgruppengrößen die Lastungleichgewichte reduziert werden können und unterstreichen dabei die Einsetzbarkeit des CM-task Schedulingalgorithmus.

6.3.2. Evaluierung semi-dynamischer Koordinationsprogramme

Die Leistungsfähigkeit semi-dynamischer Koordinationsprogramme wird für IRK Verfahren und Parallele Adams Verfahren auf einer heterogenen Plattform untersucht. Diese Anwendungen wurden gewählt, da die ausgeführten Zeitschritte für das Compilerframework sichtbar sind und entsprechende Lastausgleichsoperationen eingefügt werden können. Für die Messungen in diesem Abschnitt werden modifizierte Spezifikationsprogramme dieser Anwendungen verwendet, die Basismodule in Form paralleler Tasks enthalten und keine orthogonalen Kommunikationsmuster ausnutzen. Diese Modifikation ist notwendig, da die orthogonalen Kommunikationsoperationen zu einer Synchronisation der Prozessorgruppen führen und daher keine Laufzeitunterschiede der Basismodule im Koordinationsprogramm messbar sind.

Der für die Laufzeitexperimente verwendete heterogene Cluster enthält vier verschiedene Prozessortypen. Die relative Rechenleistung eines Kernes dieser Prozessortypen im Vergleich zum langsamsten Rechenkern ist in Abbildung 42 (links) dargestellt. Die Messungen wurden mit einer sequentiellen Implementierung eines IRK Verfahrens mit $s = 4$ Stufenvektoren für das dicht besetzte Gleichungssystem ausgeführt. Die Resultate zeigen einen Leistungsunterschied zwischen schnellstem und langsamstem Rechenkern von ungefähr 60 Prozent. Analoge Ergebnisse werden für die PABM Methode mit $K = 8$ Stufenvektoren erzielt (s. Abbildung 42 (rechts)).

Die parallele Ausführungszeit eines Basismoduls auf einer heterogenen Plattform wird durch den langsamsten ausführenden Rechenkern des jeweiligen Basismoduls bestimmt. Daher werden für die Messung der parallelen Ausführungszeit die MPI Prozessornummern nach absteigender Rechenleistung der entsprechenden Rechenkerne vergeben. Diese Anordnung

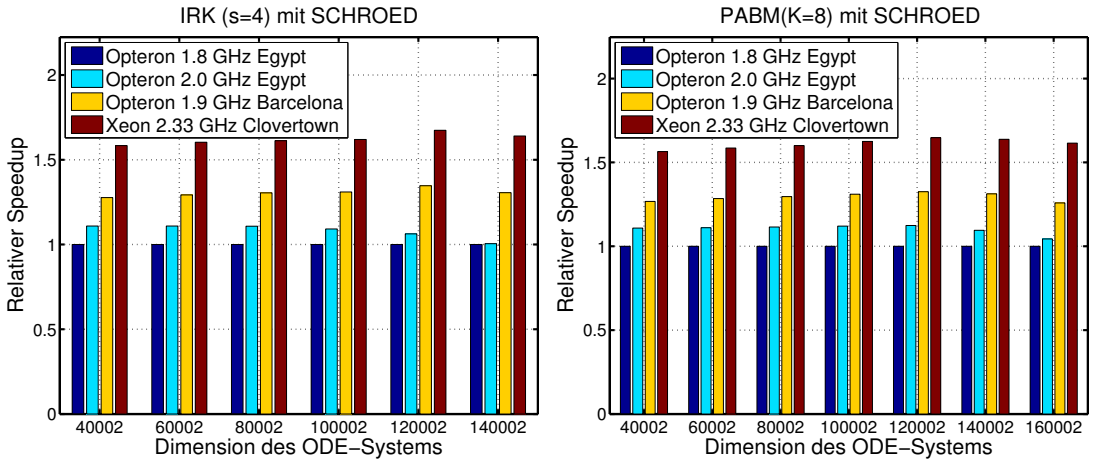


Abbildung 42: Relative Rechenleistung der Kerne des heterogenen Clusters im Vergleich zum langsamsten Rechenkern für sequentielle Implementierungen eines Iterierten Runge-Kutta Verfahrens mit $s = 4$ Stufenvektoren (links) und der PABM Methode mit $K = 8$ Stufenvektoren (rechts).

besitzt den Vorteil, dass die verwendeten Basismodule durch Rechenkerne mit möglichst gleicher Rechenleistung abgearbeitet werden.

Die Laufzeitmessungen mit dem dünn besetzten Gleichungssystem zeigen eine Laufzeitverschlechterung der semi-dynamischen Koordinationsprogramme gegenüber einer statischen Implementierung (ohne Abbildung). Dieses Verhalten resultiert aus den ungleichen Prozessorgruppengrößen, die durch Lastausgleichsoperationen bei einer semi-dynamischen Ausführung entstehen. Daraus ergibt sich ein unregelmäßiges Kommunikationsmuster zwischen den Prozessorgruppen, das gegenüber dem orthogonalen Datenaustausch eines statischen Koordinationsprogrammes zu erheblich höheren Kommunikationszeiten führt. Daher werden im Folgenden die Resultate für das dicht besetzte System diskutiert, für das aufgrund des wesentlich höheren Rechenanteils eine auf die Prozessorgeschwindigkeit angepasste Verteilung der Berechnungen wichtiger als ein regelmäßiges Kommunikationsmuster ist.

Abbildung 42 enthält einen Vergleich der Ausführungszeiten eines Zeitschrittes für IRK Verfahren (links) und für die PABM Methode mit $K = 8$ Stufenvektoren (rechts). Die erzielten Ergebnisse sind für beide Anwendungsfälle ähnlich. Für Gleichungssysteme, deren Dimension unterhalb von $n = 100002$ liegt, besitzt die semi-dynamische Realisierung eine bis zu fünf Prozent höhere Ausführungszeit als die datenparallele Version. Dieser Overhead entsteht vor allem durch die häufigen Lastausgleichsoperationen, die aufgrund von Schwankungen der Laufzeit der Basismodule zwischen den Zeitschritten entstehen. Insbesondere für die PABM Methode werden viele Lastausgleichsoperationen ausgeführt, da aus der höheren Anzahl gleichzeitig ausgeführter Basismodule kleinere Prozessorgruppengrößen resultieren, die vermehrt zu Rundungsungenauigkeiten im dynamischen Lastausgleichsalgorithmus führen.

Dagegen konnte die Lastausgleichsbibliothek für größere Gleichungssysteme bereits nach wenigen Zeitschritten eine Prozessorgruppenaufteilung finden, die zu einem guten Lastgleichgewicht führt und in den meisten Fällen keine weitere Anpassung erfordert. Die Laufzeitresultate belegen, dass auf einem heterogenen Cluster durch eine Anpassung der Prozessorgruppen-

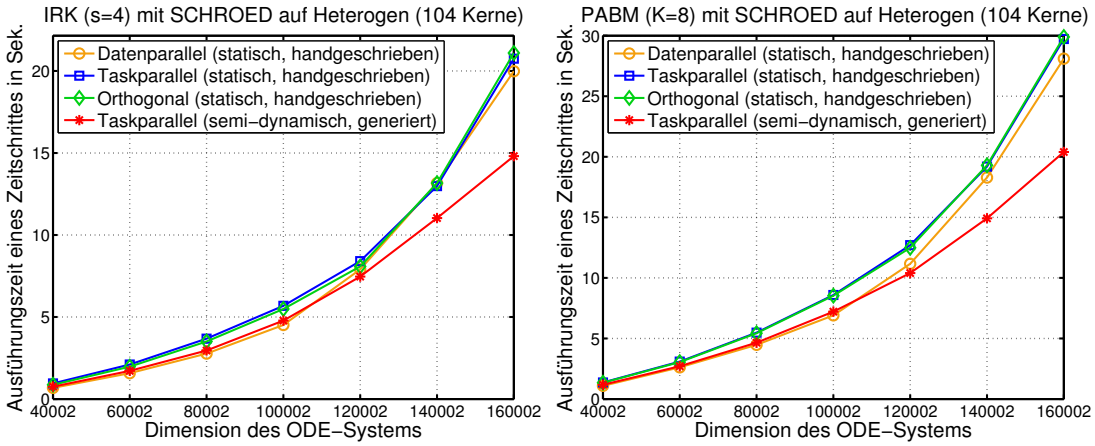


Abbildung 43: Vergleich der Ausführungszeiten eines Zeitschrittes des semi-dynamischen Koordinationsprogrammes mit verschiedenen statischen Programmversionen für Iterierte Runge-Kutta Verfahren mit $s = 4$ Stufenvektoren (links) und für die PABM Methode mit $K = 8$ Stufenvektoren (rechts).

größen, wie sie der semi-dynamische Compileransatz ermöglicht, die Laufzeit gegenüber einer statischen Ausführung signifikant reduziert werden kann.

6.3.3. Mappingstrategien für Multicore-Prozessoren

Die Abarbeitung einer CM-task Anwendung auf einem Multicore- oder SMP-Cluster erfordert neben einem Schedule, der die Ausführungsreihenfolge und die Anzahl der ausführenden Rechenkerne festlegt, auch eine Abbildung der CM-tasks auf konkrete physische Rechenkerne (Mapping). Für das Mapping von Anwendungen basierend auf parallelen Tasks wurden in [30] verschiedene Strategien entwickelt, die ebenfalls für CM-task Programme geeignet sind. In diesem Abschnitt wird zunächst ein kurzer Überblick der Strategien gegeben und anschließend die Auswirkung der Mappingstrategien auf die Ausführungszeit von IRK Verfahren und der PABM Methode diskutiert.

Für die Beschreibung der Mappingstrategien wird ein konkreter Ausführungszeitpunkt eines CM-task Programmes betrachtet, an dem der zugehörige Schedule eine zeitgleiche Ausführung einer Menge von CM-tasks festlegt. Eine Mappingstrategie muss diese CM-tasks auf disjunkte Mengen physischer Rechenkerne abbilden, so dass die Anzahl der zugeordneten Rechenkerne mit der durch den Schedule festgelegten Prozessorgruppengröße übereinstimmt. Für ein komplettes Programm muss eine derartige Abbildung für jeden Ausführungszeitpunkt definiert werden.

Im Folgenden werden verschiedene Strategien beschrieben, um zeitgleich ausgeführte CM-tasks auf eine Multicore-Plattform abzubilden.

- Die **konsekutive Mappingstrategie** verwendet die Rechenkerne desselben Knotens zur Ausführung eines CM-tasks. Falls ein CM-task mehr Rechenkerne benötigt, als ein einzelner Knoten enthält, werden mehrere Rechenknoten zur Ausführung verwendet. Damit wird die Anzahl der durch einen CM-task verwendeten Rechenknoten minimiert

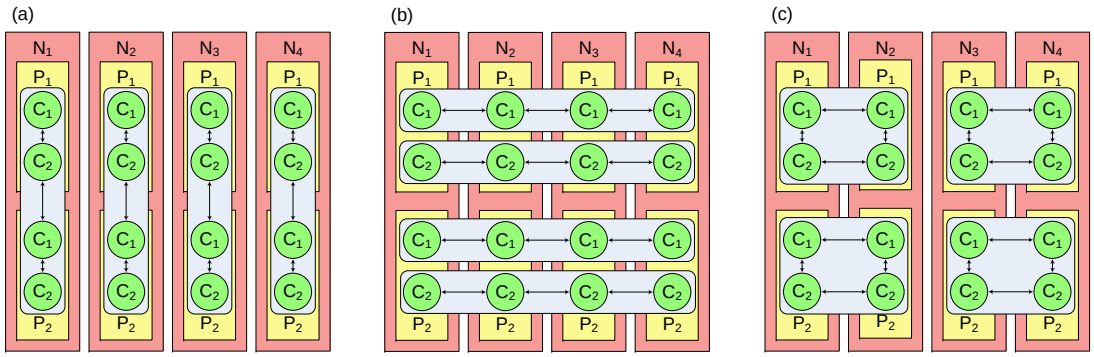


Abbildung 44: Illustration der konsekutiven (a), der verteilten (b) und der gemischten Mapingstrategie mit $d = 2$ (c) für vier zeitgleich ausgeführte CM-tasks auf eine Plattform mit vier Knoten $\{N_1, \ldots, N_4\}$. Jedem CM-task wurden durch den Schedulingalgorithmus vier Rechenkerne zugewiesen. Jeder Knoten der Plattform besteht aus zwei Prozessoren P_1 und P_2 mit jeweils zwei Rechenkernen C_1 und C_2. Die Pfeile symbolisieren interne Kommunikation der CM-tasks.

und Kommunikation innerhalb eines CM-tasks findet hauptsächlich zwischen den Kernen eines Knotens statt. Abbildung 44(a) illustriert die konsekutive Mappingstrategie.

- Die **verteilte Mappingstrategie** verwendet die korrespondierenden Rechenkerne verschiedener Rechenknoten zur Ausführung eines CM-tasks. Damit werden die internen Kommunikationsoperationen der CM-tasks gleichmäßig über verschiedene Knoten verteilt und kommunizierende Prozesse verschiedener CM-tasks, bspw. bei einem orthogonalen Kommunikationsmuster, können demselben Rechenknoten zugeordnet werden. Ein Beispiel dieser Strategie ist in Abbildung 44(b) angegeben.

- Die **gemischte Mappingstrategie** stellt eine Verallgemeinerung von konsekutivem und verteiltem Mapping dar und wird mit Hilfe eines Parameters d beschrieben. Dieser Parameter legt fest, wieviele Rechenkerne eines Knotens demselben CM-task zugeordnet werden sollen. Entspricht der Parameter d der Gesamtzahl der Rechenkerne pro Knoten, wird ein konsekutives Mapping erreicht. Ein verteiltes Mapping resultiert für $d = 1$. Abbildung 44(c) zeigt ein Beispiel für $d = 2$.

Der CM-task Compiler unterstützt die beschriebenen Mappingstrategien durch einen optionalen Arbeitsschritt, der nach der Schedulingphase ausgeführt wird und die berechneten Prozessorgruppen im Rahmenprogramm entsprechend anpasst. Die Auswahl einer geeigneten Mappingstrategie obliegt dem Anwender, wobei sowohl die Art und die Anzahl der ausgeführten Kommunikationsoperationen als auch die Kommunikationsleistung der Zielplattform berücksichtigt werden sollte. Bei Auslassen des Mappingschrittes entspricht das durch den CM-task Compiler erzeugte Koordinationsprogramm einem konsekutiven Mapping.

In den generierten orthogonalen Programmversionen für IRK Verfahren und Parallele Adams Verfahren dominieren kollektive Kommunikationsoperationen. Tabelle 6 gibt einen Überblick der ausgeführten Operationen eines Zeitschrittes. Dabei wird zwischen gruppenbasierter Kommunikation, die innerhalb von Prozessorgruppen zur Stufenvektorberechnung ausgeführt wird,

Tabelle 6: Kollektive Kommunikationsoperationen eines Zeitschrittes der generierten orthogonalen Programmversionen für Iterierte Runge-Kutta und Parallele Adams Verfahren.

Benchmark	gruppenbasierte Kommunikation	orthogonale Kommunikation	globale Kommunikation
IRK	m Multibroadcasts	m Multibroadcasts	1 Multibroadcast
PAB	1 Multibroadcast	1 Multibroadcast	–
PABM	$1+it$ Multibroadcasts	1 Multibroadcast	–

orthogonaler Kommunikation, die zwischen diesen Prozessorgruppen stattfindet, und globaler Kommunikation unter Beteiligung aller verfügbarer Prozessoren unterschieden. In der Tabelle bezeichnet m die Anzahl der Fixpunktiterationsschritte des IRK Verfahrens und it die Iterationsschritte der PABM Methode. Beide Parameter werden zur Compilezeit festgelegt.

Abbildung 45 (links) zeigt die Abhängigkeit der Ausführungszeit einer globalen Multibroadcastoperation von der gewählten Mappingstrategie auf dem CHiC Cluster. Die niedrigsten Ausführungszeiten werden durch ein konsekutives Mapping erzielt. Dieses Verhalten resultiert aus der Implementierung der zugrundeliegenden MPI Bibliothek, die für größere Nachrichten einen ringbasierten Algorithmus verwendet, der zu Kommunikationsoperationen zwischen Rechenkernen mit benachbarten Rängen führt. Durch ein konsekutives Mapping erfolgt diese Kommunikation hauptsächlich innerhalb eines Rechenknotens und kann durch Optimierungen für einen gemeinsamen Speicher profitieren.

Die Abhängigkeit der Kommunikationsleistung gruppenbasierter und orthogonaler Kommunikationsoperationen von der gewählten Mappingstrategie wird mit Hilfe des Multi-Allgather Benchmarks der Intel MPI Benchmark Suite [52] untersucht. Dieser Benchmark erzeugt eine feste Anzahl gleichgroßer Prozessorgruppen und führt zeitgleich in diesen Gruppen Multibroadcastoperationen aus. Abbildung 45 (rechts) zeigt die Ergebnisse für den CHiC Cluster mit 256 Rechenkernen. Die dargestellten Kurven für vier Prozessorgruppen entsprechen der gruppenbasierten Kommunikation in einem Differentialgleichungslöser mit vier Stufenvektoren. Die niedrigste Ausführungszeit wird durch ein konsekutives Mapping erreicht, da für diese Mappingstrategie analog zu der zuvor betrachteten globalen Multibroadcastoperation Kommunikation hauptsächlich zwischen den Rechenkernen desselben Knotens stattfindet.

Die Kurven für 64 Prozessorgruppen in Abbildung 45 (rechts) stellen die Ausführungszeit einer orthogonalen Kommunikationsoperation in einem Differentialgleichungslöser mit vier Stufenvektoren dar. Die niedrigste Ausführungszeit wird durch ein verteiltes Mapping erreicht, da sich die kommunizierenden Rechenkerne in diesem Fall auf demselben Rechenknoten befinden.

Zusammen zeigen die Resultate, dass ein konsekutives Mapping zu den geringsten Ausführungszeiten von globalen und gruppenbasierten Kommunikationsoperationen führt. Bei orthogonalen Kommunikationsoperationen führt diese Mappingstrategie jedoch zu den höchsten Ausführungszeiten. Auf der anderen Seite besitzt das verteilte Mapping die geringsten Ausführungszeiten bei orthogonaler Kommunikation, wohingegen bei globaler und gruppenbasierter Kommunikations die höchsten Ausführungszeiten entstehen. Die Ausführungszeiten des gemischten Mappings liegen für alle drei Kommunikationstypen zwischen konsekutivem und verteiltem Mapping.

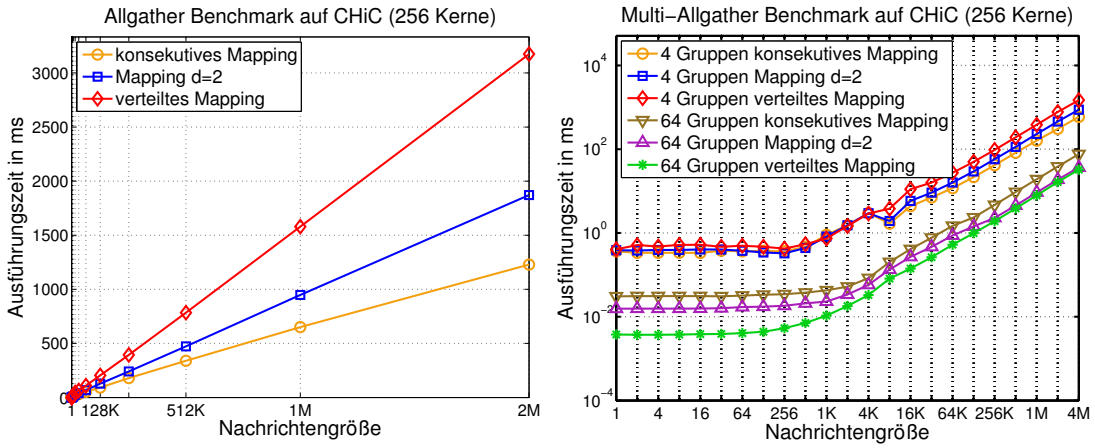


Abbildung 45: Messergebnisse des Allgather Benchmarks (links) und des Multi-Allgather Benchmarks (rechts) auf dem CHiC Cluster mit 256 Rechenkernen. Das linke Diagramm vergleicht die Ausführungszeiten einer globalen Multibroadcastoperation (`MPI_Allgather`) abhängig von der Mappingstrategie. Der Multi-Allgather Benchmark führt parallele Multibroadcastoperationen in disjunkten Prozessorgruppen aus. Das rechte Diagramm zeigt die Messergebnisse dieses Benchmarks für eine Aufteilung der Rechenkerne in vier Gruppen mit je 64 Kernen und in 64 Gruppen mit je vier Kernen.

Im Folgenden wird die Auswirkung der vorgestellten Mappingstrategien auf die Ausführungszeit der verschiedenen Differentialgleichungslöser betrachtet. Abbildung 46 zeigt die Messergebnisse der generierten orthogonalen Programmvarianten der IRK, PAB und PABM Benchmarks abhängig von der gewählten Mappingstrategie auf dem CHiC Cluster und dem JuRoPA Cluster. Zum Vergleich sind die handgeschriebenen datenparallelen und die handgeschriebenen taskparallelen Versionen angegeben. Für diese Programmversionen wurde eine konsekutive Mappingstrategie verwendet.

Die Resultate für das IRK Verfahren zeigen, dass auf beiden Plattformen die niedrigsten Ausführungszeiten durch eine konsekutive Mappingstrategie erzielt werden. Auf dem JuRoPA Cluster kann durch ein gemischtes Mapping mit $d = 4$ eine fast identische Laufzeit erreicht werden, wohingegen die anderen Strategien zu deutlich höheren Ausführungszeiten führen. Dieses Verhalten kann vor allem mit den vorhandenen globalen Kommunikationsoperationen, für die eine konsekutive Mappingstrategie zu der niedrigsten Ausführungszeit führt, erklärt werden.

In der PABM Methode dominieren gruppenbasierte Kommunikationsoperationen, so dass eine Anordnung von möglichst vielen Prozessen eines CM-tasks auf einem Rechenknoten vorteilhaft ist. Dies bestätigen die Laufzeitmessungen, die eine deutliche Überlegenheit der konsekutiven Mappingstrategie zeigen. Im Gegensatz dazu beinhaltet die PAB Methode eine identische Anzahl gruppenbasierter und orthogonaler Kommunikationsoperationen. Wie die Laufzeitresultate belegen, ist in diesem Fall eine gemischte Mappingstrategie vorteilhaft, die einen Kompromiss aus beiden Kommunikationstypen herstellt.

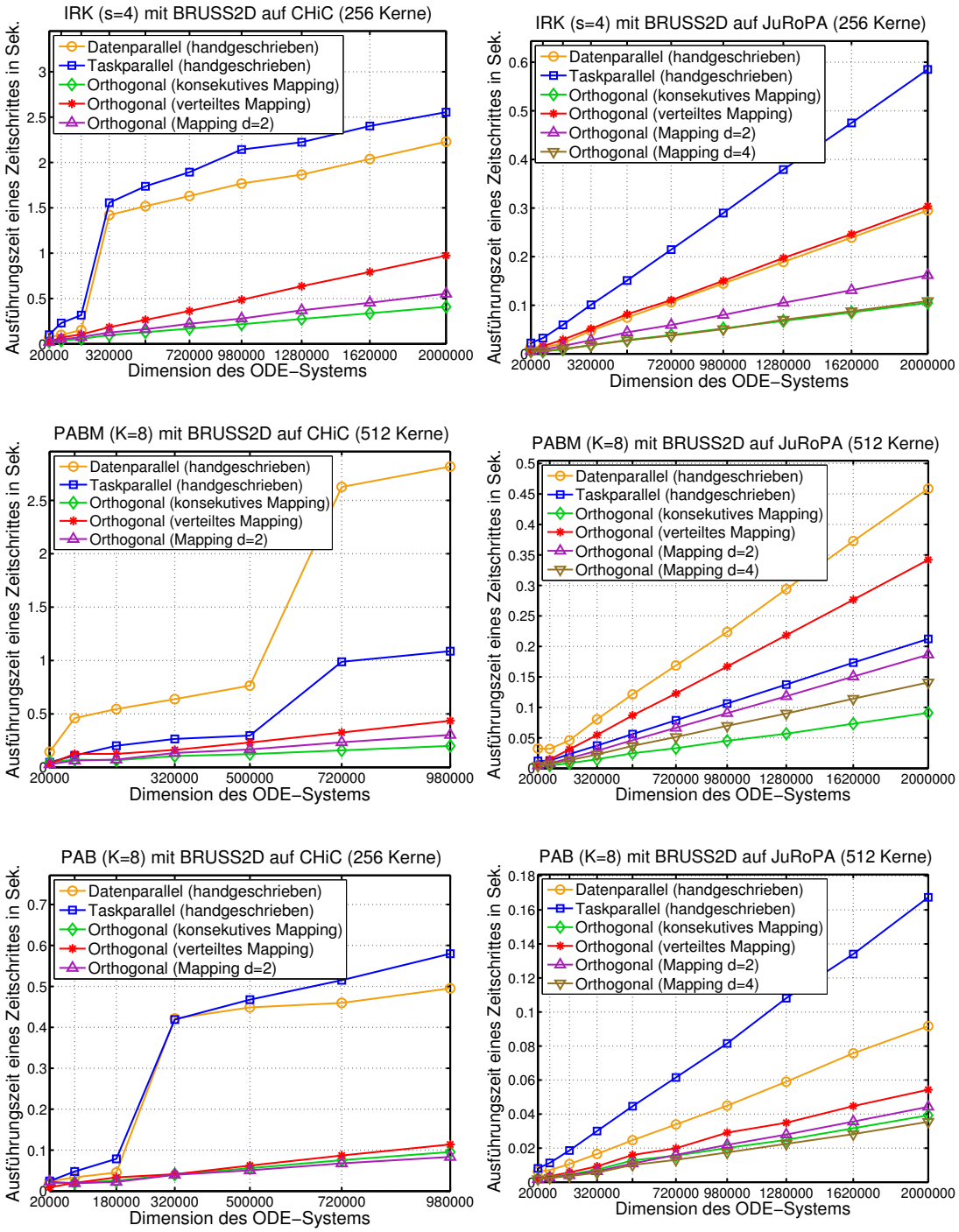


Abbildung 46: Vergleich der Ausführungszeiten eines Zeitschrittes des statischen Koordinationsprogrammes mit verschiedenen Mappingstrategien für Iterierte Runge-Kutta Verfahren (obere Zeile), die PABM Methode mit $K = 8$ Stufenvektoren (mittlere Zeile) und die PAB Methode mit $K = 8$ Stufenvektoren (untere Zeile) auf dem CHiC Cluster (linke Spalte) und dem JuRoPA Cluster (rechte Spalte).

6.4. Zusammenfassung

In diesem Kapitel wurde die Implementierung verschiedener Anwendungsprogramme mit dem CM-task Compilerframework betrachtet. Diese Anwendungen umfassen verschiedene numerische Verfahren zur Lösung gewöhnlicher Differentialgleichungssysteme und verschiedene Löser für partielle Differentialgleichungen.

Die vorgestellten Messergebnisse vergleichen die mit dem CM-task Compiler erzeugten Programmvarianten mit verschiedenen handgeschriebenen Implementierungen. Für die erzeugten statischen Koordinationsprogramme konnte für alle Anwendungen ein deutlicher Leistungsvorteil gegenüber den entsprechenden datenparallelen Programmversionen nachgewiesen werden. Desweiteren wurde durch Messungen belegt, dass der Overhead der generierten Programmversionen gegenüber handgeschriebenen CM-task Implementierungen gering ist und in den meisten Fällen unter zwei Prozent liegt.

Die Messungen für die erzeugten semi-dynamischen Koordinationsprogramme zeigen, dass auf einem heterogenen Cluster durch eine dynamische Anpassung der Prozessorgruppengrößen von zeitgleich ausgeführten Modulen deutlich niedrigere Laufzeiten gegenüber einer statischen Abarbeitung erzielt werden.

Für Multicore-Plattformen wurden in diesem Kapitel Mappingstrategien zur Abbildung von CM-tasks auf physische Rechenkerne vorgestellt. Die Messergebnisse für unterschiedliche Anwendungen und Plattformen zeigen, dass verschiedene Mappingstrategien zu deutlichen Unterschieden in der Ausführungszeit führen. Die Mappingstrategie, die die geringste Ausführungszeit erzielt, hängt sowohl vom Kommunikationsverhalten der Anwendung als auch von der Kommunikationsleistung der Plattform ab.

7. Zusammenfassung

Viele modulare Anwendungen aus dem Bereich des wissenschaftlichen Rechnens weisen sowohl taskparallele als auch datenparallele Eigenschaften auf und erfordern den regelmäßigen Datenaustausch zwischen verschiedenen Programmteilen. Das Programmiermodell der kommunizierenden Multiprozessortasks (CM-tasks) unterstützt die Strukturierung derartiger Anwendungsprogramme in Form von CM-tasks, die auf beliebigen Prozessorgruppen ausgeführt werden können und durch Koordination und Kommunikation in Verbindung stehen. Im Vergleich zu existierenden Programmiermodellen auf Basis paralleler Tasks unterstützt das CM-task Modell zusätzlich die Modellierung von Kommunikation zwischen zeitgleich ausgeführten Tasks. Dadurch können im CM-task Programmiermodell grobkörnigere Tasks definiert werden, die optimierte Kommunikationsmuster zum Austausch von berechneten Zwischenergebnissen einsetzen.

Die Abhängigkeiten zwischen den CM-tasks eines Programmes werden durch einen CM-task Graph dargestellt. Ein CM-task Graph besteht aus einer Menge von Knoten, die die CM-tasks des jeweiligen Anwendungsprogrammes repräsentieren, und einer Kantenmenge, die sowohl gerichtete als auch bidirektionale Kanten enthält. Eine gerichtete Kante zwischen zwei Knoten erfordert die Nacheinanderausführung der entsprechenden CM-tasks, wohingegen eine bidirektionale Kante zwischen zwei Knoten die gleichzeitige Ausführung der entsprechenden CM-tasks erfordert.

Die Ausführung einer CM-task Anwendung auf einer gegebenen parallelen Plattform erfordert einen Schedule, der die Ausführungsreihenfolge und die ausführenden Prozessorgruppen für die CM-tasks dieser Anwendung festlegt. Im Allgemeinen existieren viele verschiedene Schedules für eine gegebene Anwendung, die zu unterschiedlichen Ausführungszeiten der betreffenden Anwendung führen können. Der in dieser Arbeit vorgestellte CM-task Schedulingalgorithmus berechnet für eine durch einen CM-task Graph mit annotierten Kostenfunktionen gegebene CM-task Anwendung einen Schedule, der zu einer möglichst geringen Ausführungszeit der entsprechenden Anwendung führt. Durch Anwendung des Schedulingalgorithmus auf synthetische CM-task Graphen wurde nachgewiesen, dass der CM-task Schedulingalgorithmus eine geringe Laufzeit besitzt und die erzeugten Schedules sowohl einer rein datenparallelen als auch einer rein taskparallelen Ausführung überlegen sind.

Die Entwicklung von Anwendungsprogrammen im CM-task Programmiermodell wird durch das CM-task Compilerframework unterstützt. Das Framework enthält Sprachen, Bibliotheken und den CM-task Compiler. Die Sprachen umfassen die plattformunabhängige CM-task Spezifikationssprache zur Definition paralleler Anwendungen und die anwendungsunabhängige Beschreibungssprache für parallele Plattformen. Die Spezifikationssprache unterstützt die Definition von CM-tasks in Form von Basismodulen und die Definition von CM-task Programmen in Form von Verbundmodulen. Ein Verbundmodul enthält einen hierarchisch strukturierten Modulausdruck, auf dessen unterster Hierarchieebene Aufrufe von Basis- und Verbundmodulen stehen. Auf den höheren Hierarchieebenen des Modulausdrucks stehen Konstruktoren, die die Nacheinanderausführung, die zeitgleiche Ausführung oder die Unabhängigkeit von Programmteilen definieren.

Der CM-task Compiler erzeugt aus einer gegebenen Anwendungsspezifikation und einer gegebenen Plattformbeschreibung schrittweise ein ausführbares Koordinationsprogramm. Die Transformationsschritte umfassen das Erkennen der implizit vorhandenen Daten- und Kommunikationsabhängigkeiten, die Berechnung eines globalen hierarchischen Schedules und das

Einfügen von Datenumverteilungsoperationen. Die Ausgabe dieser Transformationsschritte besteht aus einer Anwendungsspezifikation mit zusätzlichen Annotationen. Der letzte Transformationsschritt erzeugt das Koordinationsprogramm, das eine auf die konkrete Plattform angepasste Implementierung der spezifizierten Anwendung darstellt. Die Ausführung des erzeugten Koordinationsprogrammes wird durch die Datenumverteilungsbibliothek und die Lastausgleichsbibliothek des CM-task Compilerframeworks unterstützt.

Der CM-task Compiler unterstützt sowohl einen statischen als auch einen semi-dynamischen Ansatz zur Erzeugung des Koordinationsprogrammes. Im statischen Ansatz des CM-task Compilers wird der zugrundeliegende Schedule für das zu erzeugende Koordinationsprogramm zur Übersetzungszeit festgelegt. Dadurch werden Compileroptimierungen wie die Eliminierung von Datenumverteilungsoperationen oder die Vorberechnung von Kommunikationsmustern für die zur Laufzeit ausgeführten Datenumverteilungsoperationen ermöglicht und durchgeführt.

Im semi-dynamischen Ansatz des CM-task Compilers wird zur Übersetzungszeit nur die Ausführungsreihenfolge der CM-tasks der spezifizierten Anwendung festgelegt. Für die diese CM-tasks ausführenden Prozessorgruppen wird zur Übersetzungszeit eine initiale Belegung berechnet, die zur Laufzeit der Anwendung basierend auf dynamisch ermittelten Leistungsdaten an die zugrundeliegende Plattform angepasst werden kann. Daraus resultiert eine flexible Abarbeitung, die speziell auf heterogenen Architekturen zu einer besseren Ausnutzung der Rechenressourcen beiträgt.

Die Datenumverteilungsbibliothek des CM-task Compilerframeworks stellt die zur korrekten Abarbeitung der vom CM-task Compiler erzeugten Koordinationsprogramme benötigten Datenumverteilungsoperationen zur Verfügung. Diese Bibliothek unterstützt replizierte und blockzyklische Datenverteilungen für mehrdimensionale Felder auf beliebigen Quell- und Zielprozessorgruppen. Die durch die Bibliothek bereitgestellten Funktionen umfassen statische und dynamische Datenumverteilungsoperationen. Die statischen Umverteilungsoperationen verwenden ein zur Übersetzungszeit durch den CM-task Compiler berechnetes Kommunikationsmuster zwischen Quell- und Zielprozessoren, wohingegen bei den dynamischen Umverteilungsoperationen das Kommunikationsmuster zur Laufzeit berechnet wird.

Die Lastausgleichsbibliothek des CM-task Compilerframeworks stellt Funktionen und Datenstrukturen zur dynamischen Anpassung der Prozessorgruppen in semi-dynamischen Koordinationsprogrammen bereit. Als Datenstruktur zur Speicherung des aktuellen Schedules eines semi-dynamischen Koordinationsprogrammes wird die Schedulestruktur verwendet, die für jeden auszuführenden CM-task die aktuell zugeordnete Prozessorgruppe sowie dynamische Leistungsdaten speichert. Kern der Lastausgleichsbibliothek des CM-task Compilerframeworks ist der Lastausgleichsalgorithmus, der basierend auf den gemessenen Ausführungszeiten zeitgleich ausgeführter CM-tasks entscheidet, ob eine Lastausgleichsoperation durchgeführt werden soll und gegebenenfalls an die Berechnungsarbeit angepasste Prozessorgruppengrößen berechnet.

Die Einsetzbarkeit des CM-task Programmiermodells und die Verwendung des CM-task Compilerframeworks wurden am Beispiel von verschiedenen Anwendungen aus dem Bereich des wissenschaftlichen Rechnens demonstriert. Konkret wurden Lösungsverfahren für Systeme gewöhnlicher Differentialgleichungen und Benchmarks aus dem Bereich der Strömungsmechanik verwendet. Anhand von Messergebnissen auf unterschiedlichen parallelen Rechenplattformen wurde für diese Anwendungen nachgewiesen, dass eine CM-task basierte Realisierung zu deutlich höheren Speedupwerten als eine rein datenparallele Implementierung und als eine Implementierung mit traditionellen parallelen Tasks führt. Desweiteren zeigen die Messungen nur einen geringen Overhead für die vom CM-task Compiler erzeugten

Koordinationsprogramme gegenüber einer handgeschriebenen Implementierung im CM-task Programmiermodell.

Auf heterogenen Plattformen treten bei der Ausführung einer Anwendung mit statisch berechneten Prozessorgruppengrößen Lastungleichgewichte zwischen den Prozessorgruppen aufgrund der ungleichen Prozessorgeschwindigkeiten auf. Durch die in den Koordinationsprogrammen des semi-dynamischen Compileransatzes des CM-task Compilers integrierte Lastausgleichsbibliothek werden diese Lastungleichgewichte zur Laufzeit der Anwendung durch eine entsprechende Anpassung der Prozessorgruppengrößen ausgeglichen. Die Messergebnisse für verschiedene semi-dynamische Koordinationsprogramme zeigen, dass der Leistungsvorteil einer derartigen Anpassung den Overhead für die zusätzlichen Lastausgleichsoperationen übersteigt und somit eine niedrigere Ausführungszeit im Vergleich zu einer statischen Ausführung ohne Lastausgleichsoperationen erreicht wird.

Die in dieser Arbeit erzielten Ergebnisse können insbesondere in Bezug auf die effiziente Ausnutzung von Multicore-Clustern und heterogenen Systemumgebungen erweitert werden. Anhand von Messergebnissen wurde in dieser Arbeit aufgezeigt, dass die Laufzeit einer CM-task Anwendung erheblich von der Abbildung der entsprechenden CM-tasks auf die Rechenkerne eines Multicore-Clusters beeinflusst wird. Die Auswahl einer geeigneten Abbildungsstrategie obliegt jedoch dem Benutzer, der dafür detaillierte Kenntnisse über das Kommunikationsverhalten der Anwendung und die Kommunikationsleistung der Zielplattform besitzen muss. An dieser Stelle kann das CM-task Compilerframework um zusätzliche Komponenten zur Unterstützung des Anwendungsentwicklers erweitert werden, bspw. um ein erweitertes Kostenmodell, das die Ausführungszeit von CM-tasks abhängig von den verwendeten Rechenkernen eines Multicore-Clusters darstellen kann.

Die parallele Effizienz von Anwendungen lässt sich auf Multicore-Clustern häufig durch Verwendung eines hybriden Programmieransatzes steigern, bei dem innerhalb eines Clusterknotens ein threadbasiertes Modell wie OpenMP und zwischen Clusterknoten ein nachrichtenbasiertes Modell wie MPI verwendet wird. Eine mögliche Erweiterung des CM-task Compilerframeworks besteht daher in der Unterstützung hybrider CM-task Anwendungen, die bspw. durch eine Definition von threadparallelen oder hybriden Basismodulen in einer erweiterten Version der Spezifikationssprache erfolgen kann. Da die Ausführung derartige Basismodule nur innerhalb eines Rechenknotens erfolgen kann, entstehen zusätzliche Einschränkungen, die beim Scheduling dieser Anwendungen geeignet berücksichtigt werden müssen.

Heterogene Rechencluster mit direkt gekoppelten Knoten werden durch den semi-dynamischen Ansatz des CM-task Compilers unterstützt. Durch die ausgeführten Lastausgleichsoperationen zur Reduzierung von Lastungleichgewichten zwischen den Prozessorgruppen entsteht jedoch zusätzlicher Koordinationsoverhead zur Laufzeit einer Anwendung. Durch Berücksichtigung heterogener Plattformen in der Schedulingphase des statischen Compileransatzes können derartige Lastungleichgewichte ebenfalls vermieden werden, ohne dass der zusätzliche Koordinationsoverhead durch Lastausgleichsoperationen entsteht. Für diese Erweiterung werden eine erweiterte Plattformbeschreibungssprache, ein erweitertes Kostenmodell und entsprechende Schedulingalgorithmen benötigt.

A. Nutzerschnittstelle des CM-task Compilers

Der CM-task Compiler ist in *Java* implementiert und damit weitgehend plattformunabhängig einsetzbar. Der Aufruf erfolgt durch

```
java -jar cmcomp.jar <Kommandozeilenparameter>,
```

wobei `cmcomp.jar` das den CM-task Compiler enthaltende Java Archiv bezeichnet. Im Folgenden wird ein Überblick der unterstützten Kommandozeilenparameter gegeben, wobei optionale Parameter in eckigen Klammern angegeben sind.

Kommandozeilenparameter für Eingabe- und Ausgabedateien

`-i <Eingabedatei>`

legt den Dateinamen der Eingabedatei (Spezifikationsprogramm, erweitertes Spezifikationsprogramm, Rahmenprogramm oder erweitertes Rahmenprogramm) fest.

`[-il <Eingabedateityp>]`

legt den Typ der mit `-i` spezifizierten Eingabedatei fest. Die folgenden Typen werden unterstützt:

1. Spezifikationsprogramm
2. erweitertes Spezifikationsprogramm
3. Rahmenprogramm
4. erweitertes Rahmenprogramm

Standard: Eingabe ist ein Spezifikationsprogramm (Option 1).

`-o <Ausgabedatei>`

legt den Dateiname des auszugebenden erweiterten Spezifikationsprogrammes, Rahmenprogrammes, erweiterten Rahmenprogrammes oder Koordinationsprogrammes fest.

`[-p <Pfadname>]`

legt den Pfadnamen für die Ausgabedatei(en) fest.
Standard: Ausgabe erfolgt im aktuellen Verzeichnis.

Kommandozeilenparameter zur Steuerung der Übersetzung

`[-step]`

Ausführen eines einzelnen Transformationsschrittes (ausgeführter Schritt ist abhängig vom mit `-il` definierten Eingabedateityp).
Standard: Ausführen aller Transformationsschritte bis zur Ausgabe des Koordinationsprogrammes.

`[-sdyn]`

Verwendung des semi-dynamischen Compileransatzes.
Standard: Verwendung des statischen Compileransatzes.

Kommandozeilenparameter für die Analysephase

`[-u <Anzahl>]`

Entrollen von `for`-Schleifen mit kleiner oder gleich *Anzahl* Iterationen.
Standard: Kein Entrollen von `for`-Schleifen (entspricht *Anzahl*=0).

Kommandozeilenparameter für die Schedulingphase

`-m <Plattformbeschreibung>`

legt den Dateinamen der Plattformbeschreibung fest.

`[-s <Schedulingalgorithmus>]`

legt den zu verwendenden Schedulingalgorithmus für das Scheduling des Supertask Graph fest. Die folgenden Algorithmen werden unterstützt:

0	datenparalleler Schedulingalgorithmus
1	taskparalleler Schedulingalgorithmus
2	CM-task Schedulingalgorithmus (s. Kapitel 3)
3	modifizierter CM-task Schedulingalgorithmus (s. Abschnitt 5.4.2)
4	TwoL-Tree [94]
5	CPA [83]
6	CPR [82]

Standard: CM-task Schedulingalgorithmus (Option 2) für statischen Compileransatz bzw. modifizierter CM-task Schedulingalgorithmus (Option 3) für semi-dynamischen Compileransatz.

`[-ms <Mappingstrategie>]`

legt die zu verwendenden Mappingstrategie für Multicore-Plattformen fest (s. Abschnitt 6.3.3). Die folgenden Strategien werden unterstützt:

1	konsekutives Mapping
2	verteiltes Mapping
100+*d*	gemischtes Mapping mit Parameter *d*

Standard: Verwendung der konsekutiven Mappingstrategie (Option 1).

Kommandozeilenparameter für die Codegenerierungsphase

`[-oh <Headerdatei>]`

legt den Namen der erzeugten Headerdatei fest.
Standard: Name der Ausgabedatei mit Dateiendung `.h`.

`[-od <Datendatei>]`

legt den Namen der Ausgabedatei mit den global definierten Hilfsvariablen des Koordinationsprogrammes fest.
Standard: Name der Ausgabedatei mit Dateiendung `_data.c`.

`[-dynredistr]`

Verwendung der dynamischen Datenumverteilungsoperationen im statischen Compileransatz.
Standard: Verwendung der statischen Datenumverteilungsoperationen (statischer Compileransatz) bzw. der dynamischen Datenumverteilungen (semi-dynamischer Compileransatz).

B. Syntax der Sprachen des CM-task Compilerframeworks

B.1. Spezifikationssprache

Die Sprachelemente der Spezifikationssprache umfassen Schlüsselworte, Bezeichner, Zahlenkonstanten, Kommentare und Satzzeichen. Für Schlüsselworte und Bezeichner gilt, dass zwischen Groß- und Kleinschreibung unterschieden wird. Leerzeichen und Zeilenumbrüche dienen als Trennzeichen und besitzen keine weitergehende Bedeutung.

Kommentare

Kommentare werden in der Spezifikationssprache analog zur Programmiersprache *C++* definiert, d.h. es gibt zwei verschiedene Arten von Kommentaren:

- einzeilige Kommentare beginnen mit einem doppelten Schrägstrich (`//`) und enden mit dem Zeilenende und
- mehrzeilige Kommentare beginnen mit dem Zeichenpaar `/*` und enden mit dem Zeichenpaar `*/`.

Schlüsselworte

Reservierte Schlüsselworte der Spezifikationssprache, die nicht für Bezeichner verwendet werden dürfen, sind `p`, `P`, `const`, `type`, `array`, `of`, `usertype`, `char`, `int`, `float`, `double`, `distrib`, `on`, `userdistrib`, `cmtask`, `in`, `out`, `inout`, `comm`, `runtime`, `cmgraph`, `cmmain`, `var`, `while`, `for`, `if`, `else`, `par`, `cpar`, `seq`, `parfor`, `cparfor`, `crels`, `prels`, `redistr`, `true` und `false`.

Bezeichner, Ausdrücke und Zahlenkonstanten

Bezeichner der Spezifikationssprache bestehen aus Buchstaben, Zahlen und Unterstrichen und beginnen mit einem Buchstaben. Nachfolgend werden Bezeichner durch das Token `ID` dargestellt. Zahlenkonstanten können als ganze Zahlen (Token: `INUM`) oder in Gleitkomma- bzw. Exponentialschreibweise (Token: `FNUM`) angegeben werden.

Arithmetische Ausdrücke werden nachfolgend durch das Nichtterminalsymbol `expr` beschrieben und bestehen aus

- Zahlenkonstanten;
- Bezeichnern und Feldausdrücken zum Zugriff auf Variablen und Konstanten;
- Klammern als Vorrangsymbolen;
- den binären linksassoziativen Operatoren $+$, $-$, $*$, $/$ und $\%$ für Addition, Subtraktion, Multiplikation, Division und Modulorechnung;
- dem binären rechtsassozistiven Operator $\hat{}$ zur Potenzierung;
- vordefinierten Funktionsaufrufen zur Berechnung der Quadratwurzel (`sqrt`) und zur Berechnung des dualen Logarithmus (`log`) und

- nutzerfefinierten Funktionsaufrufen, die im Rahmen der Plattformbeschreibung (s. Abschnitt B.5) definiert werden können.

Tabelle 7 zeigt die Syntax arithmetischer Ausdrücke.

Das nachfolgend verwendete Nichtterminalsymbol `cexpr` stellt logische Ausdrücke im Spezifikationsprogramm dar und besteht aus den binären linksassoziativen logischen Operatoren `||` und `&&` für logische Disjunktion und Konjunktion, dem unären Negationsoperator `!`, den binären Vergleichsoperatoren ==, !=, <=, <, > und >=, den Konstanten `true` und `false` sowie Bezeichnern. Die Syntax logischer Ausdrücke ist in Tabelle 7 abgebildet.

Tabelle 7: Grammatik der Spezifikationssprache (1): Arithmetische und logische Ausdrücke.

expr	→	expr + term | expr − term
term	→	term ∗ factor | term / factor | term % factor
factor	→	factor ^ simple
simple	→	**INUM** | **FNUM** | **ID** | **ID** dimlist | **ID (** exprlist **)**
	|	**(** expr **)** | **sqrt (** expr **)** | **log (** expr **)**
dimlist	→	**[** expr **]** | **[** expr **]** dimlist
exprlist	→	expr | expr **,** exprlist
cexpr	→	cexpr **||** xorexpr
xorexpr	→	xorexpr ^^ andexpr
andexpr	→	andexpr && equalityexpr
equalityexpr	→	expr **==** expr | expr **!=** expr | inequalityexpr
inequalityexpr	→	expr < expr | expr <= expr | expr >= expr | expr > expr | notexpr
notexpr	→	**!** notexpr | simplecexpr
simplecexpr	→	**true** | **false** | **(** cexpr **)**

Gültigkeit und Namensräume

Die Spezifikationssprache unterscheidet zwischen globalen Bezeichnern und lokalen Bezeichnern. Die globalen Bezeichner umfassen die Namen von definierten Basis- und Verbundmodulen, Datentypen, Datenverteilungstypen und Konstanten. Die Gültigkeit dieser Bezeichner beginnt mit ihrer Definition und erstreckt sich auf die gesamte restliche Spezifikation. Lokale Bezeichner sind die Namen der formalen Parameter und der lokalen Variablen eines Verbundmoduls. Die Gültigkeit dieser Bezeichner endet mit der Definition des entsprechenden Moduls.

Globale Konstanten, formale Parameter und lokale Variablen bilden einen gemeinsamen Namensraum, Modulnamen, Datentypen und Datenverteilungstypen besitzen jeweils eigene Namensräume.

Syntax der Spezifikationssprache

Tabelle 8 zeigt die Syntax der Spezifikationssprache anhand einer kontextfreien Grammatik.

Tabelle 8: Grammatik der Spezifikationssprache (2): Konstanten-, Datentyp-, Datenverteilungstyp-, Basismodul- und Verbundmoduldefinitionen.

prog	→	definition prog | definition	*Spezifikationsprogramm*
definition	→	constdef | typedef | distribdef	*einzelne Definition*
	|	basicmoddef | compmoddef	
constdef	→	**const ID =** expr **;**	*Konstantendefinition*
typedef	→	ftypedef | utypedef	*Datentypdefinition*
ftypedef	→	**type ID = array** dims **of** basetype **;**	*Felddatentypdefinition*
dims	→	**[** expr **]** dims | **[** expr **]**	*Felddimensionen*
basetype	→	**char** | **int** | **float** | **double**	*Basisdatentypen*
utypedef	→	**type ID = usertype (INUM) ;**	*Nutzerdatentypdefinition*
distrdef	→	fdistrdef | udistrdef	*Datenverteilungstypdefinition*
fdistrdef	→	**distrib ID: ID =** distrdims **;**	*Felddatenverteilung*
distrdims	→	distrdim distrdims | distrdim	*Verteilungsdimensionen*
distrdim	→	**[** distrtype **on** expr **]**	*Datenverteilungsdimension*
distrtype	→	**replic** | **cyclic** | **block**	*Datenverteilungsmuster*
	|	**blockcyclic (** expr **)**	
udistrdef	→	**distrib ID : ID = userdistrib (INUM) ;**	*Nutzerdatenverteilung*
basicmoddef	→	**cmtask ID (** paramlist **) runtime** expr **;**	*Basismoduldefinition*
paramlist	→	pgroup **,** paramlist | pgroup	*Parameterliste*
pgroup	→	pnames **: ID** paccdistr	*Gruppe von Parametern*
pnames	→	**ID ,** pnames | **ID**	*Parameternamen*
paccdistr	→	**:** pacctype pdistrtype | ε	*Zugriff und Datenverteilung*
pacctype	→	**in** | **out** | **inout** | **comm**	*Zugriffstyp*
pdistrtype	→	**: ID** | ε	*Datenverteilungstyp*
compmoddef	→	moddecl **ID (** paramlist **)** { cmodbody }	*Verbundmoduldefinition*
moddecl	→	**cmmain** | **cmgraph**	
cmodbody	→	vardefs modexpr	*Modulrumpf*
vardefs	→	vardef vardefs | ε	*Liste von Variablendefinitionen*
vardef	→	**var** varnames **: ID ;**	*Variablendefinition*
varnames	→	**ID ,** varnames | **ID**	*Variablennamen*
modexpr	→	**seq** { modexprlist }	*Datenabhängigkeit*
	|	**par** { modexprlist }	*Unabhängigkeit*
	|	**for (ID =** range **)** { modexpr }	*sequentielle Schleife*
	|	**while (** cexpr **) #** expr { modexpr }	*sequentielle Schleife*
	|	**parfor (ID =** range **)** { modexpr }	*parallele Schleife*
	|	**if (** cexpr **)** { modexpr }	*einseitige Bedingung*
	|	**if (** cexpr **)** { modexpr } **else** { modexpr }	*zweiseitige Bedingung*
	|	commexpr	*Kommunikationsverbund*
commexpr	→	**ID (** arglist **) ;**	*Modulaufruf*
	|	**cpar** { commexprlist }	*Kommunikationsabhängigkeit*
	|	**cparfor (ID =** range **)** { commexpr }	*parallele Schleife*
modexprlist	→	modexpr modexprlist | modexpr	*Modulausdrucksliste*
commexprlist	→	commexpr commexprlist | commexpr	*Kommunikationsverbundsliste*
arglist	→	expr **,** arglist | expr	*Liste aktueller Parameter*
range	→	expr **:** expr stepsize	*Iterationsbereich*
stepsize	→	**:** expr | ε	*Schrittweite*

B.2. Erweiterte Spezifikationssprache

Die erweiterte Spezifikationssprache enthält gegenüber der Spezifikationssprache zusätzliche Annotationen für Identifikatoren, Daten- und Kommunikationsabhängigkeiten, die innerhalb von Verbundmoduldefinitionen auftreten. Die Identifikatoren bilden innerhalb einer Verbundmoduldefinition einen eigenen Namensraum. Tabelle 9 zeigt die Syntax der erweiterten Spezifikationssprache. Die nicht in der Tabelle aufgeführten Nichtterminalsymbole sind identisch mit der Spezifikationssprache definiert.

Tabelle 9: Grammatik der erweiterten Spezifikationssprache.

eprog	→	edefinition eprog | edefinition	*erweitertes Spezifikationsprogramm*
edefinition	→	constdef | typedef | distribdef	*einzelne Definition*
	|	basicmoddef | ecompmoddef	
ecompmoddef	→	moddecl **ID (** paramlist **)** { ecmodbody }	*Verbundmoduldefinition*
ecmodbody	→	vardefs emodexpr preldef	*Modulrumpf*
emodexpr	→	**seq** { emodexprlist preldef }	*Datenabhängigkeit*
	|	**par** { emodexprlist }	*Unabhängigkeit*
	|	nname **: for (** **ID =** range **)** { emodexpr preldef }	*sequentielle Schleife*
	|	nname **: while (** cexpr **) #** expr { emodexpr preldef }	*sequentielle Schleife*
	|	**parfor (** **ID =** range **)** { emodexpr }	*parallele Schleife*
	|	nname **: if (** cexpr **)** { emodexpr preldef }	*einseitige Bedingung*
	|	nname **: if (** cexpr **)** { emodexpr preldef } **else** { modexpr preldef }	*zweiseitige Bedingung*
	|	ecommexpr	*Kommunikationsverbund*
ecommexpr	→	nname **: ID (** arglist **) ;**	*Modulaufruf*
	|	**cpar** { ecommexprlist creldef }	*Kommunikationsabhängigkeit*
	|	**cparfor (** **ID =** range **)** { ecommexpr creldef }	*parallele Schleife*
emodexprlist	→	emodexpr emodexprlist | emodexpr	*Modulausdrucksliste*
ecommexprlist	→	ecommexpr ecommexprlist | ecommexpr	*Kommunikationsverbundsliste*
preldef	→	**prels** { prellist } | ε	*Annotation von Datenabh.-keiten*
prellist	→	prel **,** prellist | prel	*Datenabhängigkeitsliste*
prel	→	**(** nname $-$ **>** nname **,** { varlist } **)**	*Datenabhängigkeit*
creldef	→	**crels** { crellist } | ε	*Annotation von Komm.-abh.-keiten*
crellist	→	crel **,** crellist | crel	*Komm.-abhängigkeitsliste*
crel	→	**(** commdef **,** { varlist } **)**	*Kommunikationsabhängigkeit*
commdef	→	nname $--$ nname | nname	*Liste von Identifikatoren*
varlist	→	varname **,** varname | varname	*Liste von Variablennamen*
varname	→	varname **[** expr **]** | **ID**	*Variablenname mit Dimension*
nname	→	nname **[INUM]** | **ID**	*Identifikator für Konstrukt*

B.3. Syntax der Rahmenprogramme

Im Rahmenprogramm wird die Syntax von Verbundmodulen gegenüber dem erweiterten Spezifikationsprogramm um zusätzliche Prozessorgruppenannotationen und Lastausgleichsannotationen erweitert. Tabelle 10 zeigt die Grammatikregeln für Rahmenprogramme. Die nicht in der Tabelle aufgeführten Nichtterminalsymbole sind identisch mit der erweiterten Spezifikationssprache definiert.

Tabelle 10: Grammatik des Rahmenprogrammes: Verbundmoduldefinitionen.

rprog	→	rdefinition rprog | rdefinition	*Rahmenprogramm*
rdefinition	→	constdef | typedef | distribdef	*einzelne Definition*
	|	basicmoddef | rcompmoddef	
rcompmoddef	→	moddecl **ID (** paramlist **)** **on** pgroup { rcmodbody }	*Verbundmoduldefinition*
rcmodbody	→	vardefs rmodexpr preldef	*Modulrumpf*
rmodexpr	→	**seq** { rmodexprlist preldef }	*Datenabhängigkeit*
	|	**par** { rmodexprlist }	*Unabhängigkeit*
	|	nname **: for (** **ID =** range **) on** pgrouplist { rmodexpr preldef }	*sequentielle Schleife*
	|	nname **: while (** cexpr **) #** expr **on** pgrouplist { rmodexpr preldef }	*sequentielle Schleife*
	|	**parfor (** **ID =** range **)** { rmodexpr }	*parallele Schleife*
	|	nname **: if (** cexpr **) on** pgrouplist { rmodexpr preldef }	*einseitige Bedingung*
	|	nname **: if (** cexpr **) on** pgrouplist { rmodexpr preldef } **else** { modexpr preldef }	*zweiseitige Bedingung*
	|	rcommexpr	*Kommunikationsverbund*
rcommexpr	→	nname **: ID (** arglist **) on** pgrouplist **;**	*Modulaufruf*
	|	**cpar** { rcommexprlist creldef }	*Kommunikationsabhängigkeit*
	|	**cparfor (** **ID =** range **)** { rcommexpr creldef }	*parallele Schleife*
rmodexprlist	→	rmodexpr rmodexprlist | rmodexpr	*Modulausdrucksliste*
rcommexprlist	→	rcommexpr rcommexprlist | rcommexpr	*Kommunikationsverbundsliste*
pgrouplist	→	**[** plist **]** | pgroup	*Liste oder einzelne Gruppe*
plist	→	pgroup **,** plist | pgroup	*Prozessorgruppenliste*
pgroup	→	{ pentrylist }	*Prozessorgrupp*
pentrylist	→	pentry **,** pentrylist | pentry	*Liste von Teilgruppen*
pentry	→	**INUM**	*einzelner Prozessor*
	|	**INUM .. INUM**	*konsekutive Prozessornummern*

B.4. Syntax erweiterter Rahmenprogramme

Das erweiterte Rahmenprogramm enthält gegenüber dem Rahmenprogramm zusätzliche Annotationen für Datenumverteilungsoperationen und Lastausgleichsannotationen innerhalb von Verbundmoduldefinitionen. Tabelle 11 zeigt die Syntax erweiterter Rahmenprogramme. Die nicht aufgeführten Nichtterminalsymbole sind identisch zu den Rahmenprogrammen definiert.

Tabelle 11: Grammatik des erweiterten Rahmenprogrammes: Verbundmoduldefinitionen.

erprog	→	erdefinition erprog | erdefinition	*erweitertes Rahmenprogramm*
erdefinition	→	constdef | typedef | distribdef	*einzelne Definition*
	|	basicmoddef | ercompmoddef	
ercompmoddef	→	moddecl **ID (** paramlist **)** **on** pgroup { ercmodbody }	*Verbundmoduldefinition*
ercmodbody	→	vardefs erbody	*Modulrumpf*
erbody	→	ermodexpr preldef redistrdef	*Rumpf*
ermodexpr	→	**seq** { ermodexprlist preldef redistrdef }	*Datenabhängigkeit*
	|	**par** { ermodexprlist }	*Unabhängigkeit*
	|	nname **: for (** **ID =** range **) on** pgrouplist rebal { erbody }	*sequentielle Schleife*
	|	nname **: while (** cexpr **) #** expr **on** pgrouplist rebal { erbody }	*sequentielle Schleife*
	|	**parfor (** **ID =** range **)** { ermodexpr }	*parallele Schleife*
	|	nname **: if (** cexpr **) on** pgrouplist rebal { erbody }	*einseitige Bedingung*
	|	nname **: if (** cexpr **) on** pgrouplist rebal { erbody } **else** { erbody }	*zweiseitige Bedingung*
	|	ercommexpr	*Kommunikationsverbund*
ercommexpr	→	nname **: ID (** arglist **) on** pgrouplist **;**	*Modulaufruf*
	|	**cpar** { ercommexprlist creldef }	*Kommunikationsabhängigkeit*
	|	**cparfor (** **ID =** range **)** { ercommexpr creldef }	*parallele Schleife*
ermodexprlist	→	ermodexpr ermodexprlist | ermodexpr	*Modulausdrucksliste*
ercommexprlist	→	ercommexpr ercommexprlist	*Kommunikationsverbundsliste*
	|	ercommexpr	
rebal	→	**@ rebalance (** expr **)** | **@ rebalance** | ε	*Lastausgleichsannotation*
redistrdef	→	**redistr** { redistrlist } | ε	*Umverteilungsannotation*
redistrlist	→	redistr **,** redistrlist | redistr	*Umverteilungsliste*
redistr	→	**(** redistrst **− >** redistrst **)**	*Datenumverteilung*
redistrst	→	nname **:** varname **:** distrname **on** pgroup	*Umverteilungsquelle bzw. -ziel*
distrname	→	distrname **[** expr **]** | **ID**	*Quell-/Zielverteilungsname*

B.5. Plattformbeschreibungssprache

Die Plattformbeschreibungssprache definiert die verfügbare Prozessoranzahl sowie die Rechen- und Kommunikationsleistung einer parallelen Plattform. Tabelle 12 zeigt die Syntax der Plattformbeschreibungssprache.

Tabelle 12: Grammatik der Plattformspezifikation.

platform	→	**machine** { definitions }	*gesamte Plattform*
definitions	→	definition definitions | definition	*Eigenschaften der Plattform*
definition	→	**ID = INUM ;**	*Definition einer Integer-Konstante*
	|	**ID = FNUM ;**	*Definition einer Gleitkommakonstante*
	|	**ID** (idlist) = expr **;**	*Definition einer Funktion*
idlist	→	**ID ,** idlist | **ID**	*Parameterliste einer Funktion*

C. Schnittstellen und Datentypen des CM-task Compilerframeworks

C.1. Schnittstellen zur Einbindung von Nutzerdatentypen

Die vom CM-task Compiler erzeugten Koordinationsprogrammen verwenden die folgenden beiden Funktionen zur Speicheranforderung bzw. zur Speicherfreigabe von im Spezifikationsprogramm definierten Nutzerdatentypen. Die Implementierung dieser Funktionen erfolgt durch den Anwendungsentwickler.

```
void *CMC_Alloc_user (int typeid);
```

typeid - Typidentifikator des im Spezifikationsprogramm definierten Nutzerdatentyps.
Rückgabe: Zeiger auf den für den Nutzerdatentyp angeforderten Speicherbereich.

```
void CMC_Free_user (int typeid, void *data);
```

typeid - Typidentifikator des im Spezifikationsprogramm definierten Nutzerdatentyps.
data - Zeiger auf die zuvor mit `CMC_Alloc_user` angeforderte Datenstruktur.

C.2. Schnittstellen und Datentypen der Datenumverteilungsbibliothek

C.2.1. Statische Datenumverteilungsoperationen

Datenstruktur zur Speicherung der zur Compilezeit vorberechneten Kopier- und Kommunikationsoperationen für einen Quell- oder einen Zielprozessor einer statischen Datenumverteilungsoperation.

```
typedef struct cmc_redistr_desc {
    int numf; int *fofs; int *fsize; int bsize;
    int *counts; int *displs;
} CMC_REDISTR_DESC;
```

numf - Anzahl zu kopierender Fragmente.
fsize - Anzahl Feldelemente pro kopierendes Fragment.
fofs - Positionen der zu kopierenden Fragmente im Feld.
bsize - Größe des Sende- bzw. Empfangspuffers.
counts - Anzahl zu sendender bzw. zu empfangender Pufferelemente pro Ziel- bzw. Quellprozessor.
displs - Positionen der zu sendenden bzw. zu empfangenden Pufferelemente pro Ziel- bzw. Quellprozessor.

Funktion zur Realisierung der Sendephase einer statischen Datenumverteilungsoperation.

```
void CMC_Red_send_static (void *data, MPI_Datatype type,
    int sgsize, int *sprocids, CMC_REDISTR_DESC *sredistrdescs,
    int dgsize, int *dprocids, MPI_Comm comm, int tag);
```

data - Zeiger auf das umzuverteilende mehrdimensionale Feld.
type - MPI Datentyp des Basisdatentyps des umzuverteilenden Feldes.
sgsize - Anzahl der Quellprozessoren.
sprocids - Prozessornummern der Quellprozessoren im Kommunikator `comm`.
sredistrdescs - Definition der auszuführenden Kopier- und Sendeoperationen der Quellprozessoren.
dgsize - Anzahl der Zielprozessoren.
dprocids - Prozessornummern der Zielprozessoren im Kommunikator `comm`.
comm - MPI Kommunikator, der Quell- und Zielprozessoren enthält.
tag - eindeutige Kennung der Datenumverteilungsoperation.

Funktion zur Realisierung der Empfangsphase einer statischen Datenumverteilungsoperation.

```
void CMC_Red_recv_static (void *data, MPI_Datatype type,
    int sgsize, int *sprocids, int dgsize, int *dprocids,
    CMC_REDISTR_DESC *dredistrdescs, MPI_Comm comm, int tag);
```

data - Zeiger auf das umzuverteilende mehrdimensionale Feld.
type - MPI Datentyp des Basisdatentyps des umzuverteilenden Feldes.
sgsize - Anzahl der Quellprozessoren.
sprocids - Prozessornummern der Quellprozessoren im Kommunikator `comm`.
dgsize - Anzahl der Zielprozessoren.
dprocids - Prozessornummern der Zielprozessoren im Kommunikator `comm`.
dredistrdescs - Definition der auszuführenden Empfangs- und Kopieroperationen der Zielprozessoren.
comm - MPI Kommunikator, der Quell- und Zielprozessoren enthält.
tag - eindeutige Kennung der Datenumverteilungsoperation.

C.2.2. Dynamische Datenumverteilungsoperationen

Datenstruktur zur Kodierung des parametrisierten Datenverteilungsvektors eines im Spezifikationsprogramm definierten Felddatenverteilungstyps.

```
typedef struct cmc_distrib_desc {
    int **MSM; int **BSM;
} CMC_DISTRIB_DESC;
```

MSM - $P \times d$ Matrix zur Kodierung des virtuellen Prozessorgitters des parametrisierten Datenverteilungsvektors eines d-dimensionales Feldes für eine Plattform mit P Prozessoren.
BSM - $P \times d$ Matrix zur Kodierung der Blockgrößen des parametrisierten Datenverteilungsvektors eines d-dimensionalen Feldes für eine Plattform mit P Prozessoren.

Funktion zur Realisierung der Sendephase einer dynamischen Datenumverteilungsoperation.

```
void CMC_Red_send_dynamic (void *data, int numdims,
    int *dimsizes, MPI_Datatype type,
    int sgsize, int *sprocids, CMC_DISTRIB_DESC sdistr,
    int dgsize, int *dprocids, CMC_DISTRIB_DESC ddistr,
    MPI_Comm comm, int tag);
```

data - Zeiger auf umzuverteilendes mehrdimensionales Feld.
numdims - Anzahl der Felddimensionen.
dimsizes - Ausdehnung der Felddimensionen.
type - MPI Datentyp des Basisdatentyps des umzuverteilenden Feldes.
sgsize - Anzahl der Quellprozessoren.
sprocids - Prozessornummern der Quellprozessoren im Kommunikator `comm`.
sdistr - Kodierung des Quelldatenverteilungstyps.
dgsize - Anzahl der Zielprozessoren.
dprocids - Prozessornummern der Zielprozessoren im Kommunikator `comm`.
ddistr - Kodierung des Zieldatenverteilungstyps.
comm - MPI Kommunikator, der Quell- und Zielprozessoren enthält.
tag - eindeutige Kennung der Datenumverteilungsoperation.

Funktion zur Realisierung der Empfangsphase einer dynamischen Datenumverteilungsoperation.

```
void CMC_Red_recv_dynamic (void *data, int numdims,
    int *dimsizes, MPI_Datatype type,
    int sgsize, int *sprocids, CMC_DISTRIB_DESC sdistr,
    int dgsize, int *dprocids, CMC_DISTRIB_DESC ddistr,
    MPI_Comm comm, int tag);
```

data - Zeiger auf umzuverteilendes mehrdimensionales Feld.
numdims - Anzahl der Felddimensionen.
dimsizes - Ausdehnung der Felddimensionen.
type - MPI Datentyp des Basisdatentyps des umzuverteilenden Feldes.
sgsize - Anzahl der Quellprozessoren.
sprocids - Prozessornummern der Quellprozessoren im Kommunikator `comm`.
sdistr - Kodierung des Quelldatenverteilungstyps.
dgsize - Anzahl der Zielprozessoren.
dprocids - Prozessornummern der Zielprozessoren im Kommunikator `comm`.
ddistr - Kodierung des Zieldatenverteilungstyps.
comm - MPI Kommunikator, der Quell- und Zielprozessoren enthält.
tag - eindeutige Kennung der Datenumverteilungsoperation.

C.2.3. Nutzerdefinierte Datenumverteilungsoperationen

Funktionsschnittstelle zur Einbindung der Sendephase für Nutzerdatentypen. Die Implementierung dieser Funktion erfolgt durch den Anwendungsentwickler.

```
void CMC_Red_send_user (void *data, int typeid,
    int sgsize, int *sprocids, int sdistrid,
    int dgsize, int *dprocids, int ddistrid,
    MPI_Comm comm, int tag);
```

data - Zeiger auf umzuverteilende Datenstruktur.
typeid - Typidentifikator der umzuverteilenden Datenstruktur.
sgsize - Anzahl der Quellprozessoren.
sprocids - Prozessornummern der Quellprozessoren im Kommunikator `comm`.
sdistrid - Identifikator des Quelldatenverteilungstyps.
dgsize - Anzahl der Zielprozessoren.
dprocids - Prozessornummern der Zielprozessoren im Kommunikator `comm`.
ddistrid - Identifikator des Zieldatenverteilungstyps.
comm - MPI Kommunikator, der Quell- und Zielprozessoren enthält.
tag - eindeutige Kennung der Datenumverteilungsoperation.

Funktionsschnittstelle zur Einbindung der Empfangsphase für Nutzerdatentypen. Die Implementierung dieser Funktion erfolgt durch den Anwendungsentwickler.

```
void CMC_Red_recv_user (void *data, int typeid,
    int sgsize, int *sprocids, int sdistrid,
    int dgsize, int *dprocids, int ddistrid,
    MPI_Comm comm, int tag);
```

data - Zeiger auf umzuverteilende Datenstruktur.
typeid - Typidentifikator der umzuverteilenden Datenstruktur.
sgsize - Anzahl der Quellprozessoren.
sprocids - Prozessornummern der Quellprozessoren im Kommunikator `comm`.
sdistrid - Identifikator des Quelldatenverteilungstyps.
dgsize - Anzahl der Zielprozessoren.
dprocids - Prozessornummern der Zielprozessoren im Kommunikator `comm`.
ddistrid - Identifikator des Zieldatenverteilungstyps.
comm - MPI Kommunikator, der Quell- und Zielprozessoren enthält.
tag - eindeutige Kennung der Datenumverteilungsoperation.

C.3. Datenstrukturen und Schnittstellen der Lastausgleichsbibliothek

Datentypdefinition einer *Taskstruktur*, die in einem semi-dynamischen Koordinationsprogramm die Ausführung eines Basismoduls, eines Verbundmoduls, einer Schleife (`for`- oder `while`-Konstrukt) oder einer Bedingung (`if`-Konstrukt) repräsentiert.

```
typedef struct cmc_task {
    int gsize, *procids;
    MPI_Comm gcomm;
    int modified, execcount;
    double timeused;
    int minp;
    struct cmc_schedule *subschedule, *subschedule2;
} CMC_TASK;
```

`gsize` - Anzahl der aktuell dem zugehörigen Konstrukt zugewiesenen Prozessoren.
`procids` - Prozessornummern der aktuell dem zugehörigen Konstrukt zugewiesenen Prozessoren.
`gcomm` - MPI Kommunikator für die aktuell dem zugehörigen Konstrukt zugewiesene Prozessorgruppe.
`modified` - Flag zur Anzeige einer Änderung der zugewiesenen Prozessorgruppe des zugehörigen Konstrukts.
`execcount` - Ausführungsanzahl des zugehörigen Konstrukts.
`timeused` - kumulierte Ausführungszeit des zugehörigen Konstrukts.
`minp` - für die Ausführung des zugehörigen Konstrukts benötigte Mindestanzahl an Prozessoren.
`subschedule` - Zeiger auf die Schedulestruktur des entsprechenden Verbundmoduls (für Verbundmodulaufrufe), des Schleifenrumpfes (für Schleifen) oder des if-Zweiges (Bedingungen).
`subschedule2` - Zeiger auf die Schedulestruktur des else-Zweiges (Bedingungen).

Datentypdefinition einer Kommunikationsabhängigkeit in semi-dynamischen Koordinationsprogrammen.

```
typedef struct cmc_commdep {
    int numtasks;
    struct cmc_task **tasks;
    struct cmc_comm_desc *commdesc;
} CMC_COMMDEP;
```

`numtasks` - Anzahl der an der Kommunikationsabhängigkeit beteiligten Basismodulaufrufe.
`tasks` - Feld von Zeigern auf die Taskstrukturen der beteiligten Basismodulaufrufe.
`commdesc` - Zeiger auf zugehörige Kommunikationsstruktur (s. Seite 64).

Datentypdefinition einer *Schichtstruktur*, die in einem semi-dynamischen Koordinationsprogramm die gleichzeitige Ausführung einer Menge von Modulen definiert.

```
typedef struct cmc_layer {
    int numtasks;
    struct cmc_task **tasks;
    int numcdeps;
    struct cmc_commdep **cdeps;
} CMC_LAYER;
```

numtasks - Anzahl zeitgleich ausgeführter Module.
tasks - Feld von Zeigern auf die Taskstrukturen der zeitgleich ausgeführten Module.
numcdeps - Anzahl enthaltener Kommunikationsabhängigkeiten.
tasks - Feld von Zeigern auf die enthaltenen Kommunikationsabhängigkeiten.

Datentypdefinition einer *Schedulestruktur*, die in einem semi-dynamischen Koordinationsprogramm den Schedule eines Programmblockes (d.h. eines Verbundmoduls, eines Schleifenrumpfs oder eines Bedingungszweiges) repräsentiert.

```
typedef struct cmc_schedule {
    int numlayers;
    struct cmc_layer **layers;
} CMC_SCHEDULE;
```

numlayers - Anzahl nacheinander auszuführender Schichtstrukturen.
layers - Feld von Zeigern auf die entsprechenden Schichtstrukturen.

Funktion zur Anforderung der Kommunikatoren für einen Programmblock, d.h. für ein Verbundmodul, einen Schleifenrumpf oder einen Bedingungszweig.

```
void CMC_Lb_allocate (CMC_SCHEDULE *schedule, MPI_Comm comm);
```

schedule - Zeiger auf die Schedulestruktur des entsprechenden Programmblockes.
comm - MPI Kommunikator mit allen für den entsprechenden Programmblock verfügbaren Prozessoren.

Funktion zur Freigabe der Kommunikatoren eines Programmblockes, d.h. für ein Verbundmodul, einen Schleifenrumpf oder einen Bedingungszweig.

```
void CMC_Lb_deallocate (CMC_SCHEDULE *schedule);
```

schedule - Zeiger auf die Schedulestruktur des entsprechenden Programmblockes.

Funktion für den dynamischen Lastausgleich in einem Programmblock, d.h. in einem Verbundmodul, in einem Schleifenrumpf oder in einem Bedingungszweig.

```
void CMC_Lb_rebalance (CMC_SCHEDULE *schedule, MPI_Comm comm,
    int modified);
```

schedule - Zeiger auf die Schedulestruktur des entsprechenden Programmblockes.
comm - MPI Kommunikator mit allen für den entsprechenden Programmblock verfügbaren Prozessoren.
modified - Flag zur Anzeige der Veränderung der Prozessorgruppe des entsprechenden Programmblockes.

Funktion zur Realisierung des Lastausgleichsalgorithmus zwischen einer Menge zeitgleich ausgeführter Module.

```
int CMC_Lb_rebalance_alg (int numprocs, int numtasks,
    double *timeused, int *gsizes, int *mingsizes,
    int modified, int *newgsizes);
```

numprocs - Anzahl verfügbarer Prozessoren im zugehörigen Programmblock.
numtasks - Anzahl der zeitgleich ausgeführten Module.
timeused - gemessene Ausführungszeiten der zeitgleich ausgeführten Module.
gsizes - Prozessorgruppengrösen der zeitgleich ausgeführten Module.
mingsizes - Mindestprozessorzahlen der zeitgleich ausgeführten Module.
modified - Flag zur Anzeige der Veränderung der verfügbaren Prozessoranzahl des zugehörigen Programmblockes.
newgsizes - Ausgabe angepasster Prozessorgruppengrößen der zeitgleich ausgeführten Module.
Rückgabe: Entscheidung, ob ein Lastausgleich erforderlich ist.

D. Anwendungsspezifikationen

Listing 12: Spezifikationsprogramm für parallele Adams Verfahren.

```
// Konstantendefinitionen
const K = 8;                    // Anzahl Stufen
const n = 2*500^2;              // Dimension des ODE Systems
const It = 4;                   // Anzahl Fixpunktiterationen mit PAM

// Datentypdefinitionen
type scalar = array[1] of double;
type vector = array[n] of double;
type svectors = array[K][n] of double;

// Datenverteilungstypdefinitionen
distrib vector:block = [block on p];
distrib svectors:replic = [replic on 1][replic on p];

// Basismoduldefinitionen
cmtask init_step (x,h:scalar:out) runtime 2*T_op;
cmtask update_step (x,h:scalar:inout) runtime 2*T_op;
cmtask pabm_step (k:int, x,h:scalar:in, y_k:vector:inout:block,
  y1_k1:vector:in:block, ort:svectors:comm)
  runtime (It+1)*n/p*T_eval+(2*K+1)*n*T_op/p+It*3*n*T_op/p+
    T_ag(K, n*K/p)+T_bc(K, n*K/p)+(It+1)*T_ag(p/K, n*K/p);

// Hauptmoduldefinition
cmmain pabm (y:svectors:inout:replic) {
  // Deklaration lokaler Variablen
  var x,h : scalar;
  var ort : svectors;
  var k : int;

  // Modulausdruck
  seq {
    init_step (x,h);
    while (x[0] < X)#100 {
      seq {
        cparfor (k = 0:K-1) {
          pabm_step (k,x,h,y[k],y[K-1],ort);
        }
        update_step (x,h);
      }
    }
  }
}
```

Listing 13: Spezifikationsprogramm des LU-MZ Benchmarks.

```
// Konstantendefinitionen
const gx_size = 304;       // Gesamtgröße Lösungsgebiet x-Richtung
const gy_size = 208;       // Gesamtgröße Lösungsgebiet y-Richtung
const gz_size = 17;        // Gesamtgröße Lösungsgebiet z-Richtung

// Datentypdefinitionen
type dvec = array[5] of double;
type dscal = array[1] of double;
// Zonengrenze einer Zone in Nord-Süd-Richtung
type ns_border = array[5*(gz_size+gx_size/4)] of double;
// Zonengrenze einer Zone in West-Ost-Richtung
type we_border = array[5*(gz_size+gy_size/4)] of double;
// Zonengrenzen aller Zonen in Nord-Süd-Richtung
type ns_borders = array[4][4][5*(gz_size+gx_size/4)] of double;
// Zonengrenzen aller Zonen in West-Ost-Richtung
type we_borders = array[4][4][5*(gz_size+gy_size/4)] of double;

// Basismodul zur Berechnung einer Zone
cmtask lu_compute_zone (numsteps,xpos,ypos:int,
  rsdnm,errnm:dvec:out, frc:dscal:out,
  north,south:ns_border:comm, east,west:we_border:comm)
  runtime (gx_size/4)*(gy_size/4)*gz_size/p;

// Hauptmodul
cmmain lumz (numsteps:int, rsdnm, errnm:dvec, frc:dscal) {
  // Deklaration lokaler Variablen
  var ns_comm : ns_borders;     // Randwerte in Nord-Süd-Richtung
  var we_comm : we_borders;     // Randwerte in West-Ost-Richtung
  var i,j : int;

  // Modulausdruck
  cparfor (i=0:3) {
    cparfor (j=0:3) {
      lu_compute_zone (numsteps, i, j, rsdnm, errnm, frc,
        ns_comm[i][j], ns_comm[i][(j+3)%4], we_comm[i][j],
        we_comm[(i+3)%4][j]);
    }
  }
}
```

Listing 14: Spezifikation des SP-MZ Benchmarks.

```
// Konstantendefinitionen
const x_zones = 16;          // Anzahl Zonen x-Richtung
const y_zones = 16;          // Anzahl Zonen y-Richtung
const gx_size = 480;         // Gesamtgröße Lösungsgebiet x-Richtung
const gy_size = 320;         // Gesamtgröße Lösungsgebiet y-Richtung
const gz_size = 28;          // Gesamtgröße Lösungsgebiet z-Richtung
const x_pack = 2;            // Zonen pro Modulaufruf in x-Richtung
const y_pack = 1;            // Zonen pro Modulaufruf in y-Richtung

// Datentypdefinitionen
type dvec = array[5] of double;
// Zonengrenze einer Zone in Nord-Süd-Richtung
type ns_border = array[5*(gz_size+x_pack*(gx_size/x_zones))]
  of double;
// Zonengrenze einer Zone in West-Ost-Richtung
type we_border = array[5*(gz_size+y_pack*(gy_size/y_zones))]
  of double;
// Zonengrenzen aller Zonen in Nord-Süd-Richtung
type ns_borders = array[x_zones][y_zones][5*(gz_size+x_pack*
   (gx_size/x_zones))] of double;
// Zonengrenzen aller Zonen in West-Ost-Richtung
type we_borders = array[x_zones][y_zones][5*(gz_size+y_pack*
   (gy_size/y_zones))] of double;

// Basismodul zur Berechnung von numx*numy Zonen
cmtask sp_compute_zone (numsteps,xpos,ypos,numx,numy:int,
   rsdnrm,errnm:dvec:out, north,south:ns_border:comm,
   east,west:we_border:comm) runtime numx*(gx_size/x_zones)*
     numy*(gy_size/y_zones)*gz_size/p;

// Hauptmodul
cmmain spmz (numsteps:int, errnm, rsdnrm:dvec) {
  // Deklaration lokaler Variablen
  var ns_comm : ns_borders;      // Randwerte in Nord-Süd-Richtung
  var we_comm : we_borders;      // Randwerte in West-Ost-Richtung
  var i, j : int;

  // Modulausdruck
  cparfor (i=0:x_zones/x_pack-1) {
    cparfor (j=0:y_zones/y_pack-1) {
      sp_compute_zone (numsteps, i*x_pack, j*y_pack, x_pack,
        y_pack, errnm, rsdnrm, ns_comm[i][j],
        ns_comm[i][(j+y_zones/y_pack-1)%(y_zones/y_pack)],
        we_comm[i][j],
        we_comm[(i+x_zones/x_pack-1)%(x_zones/x_pack)][j]);
    }
  }
}
```

Listing 15: Spezifikationsprogramm des BT-MZ Benchmarks.

```
// Konstantendefinitionen
const x_zones = 16;          // Anzahl Zonen x-Richtung
const y_zones = 16;          // Anzahl Zonen y-Richtung
const gx_size = 480;         // Gesamtgröße Lösungsgebiet x-Richtung
const gy_size = 320;         // Gesamtgröße Lösungsgebiet y-Richtung
const gz_size = 28;          // Gesamtgröße Lösungsgebiet z-Richtung
const x_pack = 4;            // Zonen pro Modulaufruf in x-Richtung
const y_pack = 2;            // Zonen pro Modulaufruf in y-Richtung
const ratio = 4.5;           // Größenverhältnis der Zonen
// Hilfskonstanten zur Zonengrößenberechnung
const x_smallest = gx_size*(x_r-1.0)/(x_r^Xzones-1.0);
const y_smallest = gy_size*(y_r-1.0)/(y_r^Yzones-1.0);
const x_r = 2^(log(ratio)/(Xzones-1.0));
const y_r = 2^(log(ratio)/(Yzones-1.0));

// Datentypdefinitionen (Bedeutung analog zu SP-MZ Benchmark)
type dvec = array[5] of double;
type ns_border = array[5*(gz_size+x_pack*(gx_size/x_zones))]
  of double;
type we_border = array[5*(gz_size+y_pack*(gy_size/y_zones))]
  of double;
type ns_borders = array[x_zones][y_zones][5*(gz_size+x_pack*
   (gx_size/x_zones))] of double;
type we_borders = array[x_zones][y_zones][5*(gz_size+y_pack*
   (gy_size/y_zones))] of double;

// Basismodul mit Kosten abhängig von xpos,ypos,numx und numy
cmtask bt_compute_zone (numsteps,xpos,ypos,numx,numy:int,
   rsdnrm,errnm:dvec:out, north,sourth:ns_border:comm,
   east,west:we_border:comm)
   runtime ((x_smallest*(x_r^(xpos+xsize)-1.0)/(x_r-1.0))-
     (x_smallest*(x_r^xpos-1.0)/(x_r-1.0)))*
     ((y_smallest*(y_r^(ypos+ysize)-1.0)/(y_r-1.0))-
     (y_smallest*(y_r^ypos-1.0)/(y_r-1.0)))/(p*1.0);

// Hauptmodul
cmmain btmz (numsteps:int, errnm, rsdnrm:dvec) {
  // Deklaration lokaler Variablen
  var ns_comm : ns_borders;     // Randwerte in Nord-Süd-Richtung
  var we_comm : we_borders;     // Randwerte in West-Ost-Richtung
  var i, j : int;

  // Modulausdruck
  cparfor (i=0:x_zones/x_pack-1) {
    cparfor (j=0:y_zones/y_pack-1) {
      bt_compute_zone (numsteps, i*x_pack, j*y_pack, x_pack,
        y_pack, errnm, rsdnrm, ns_comm[i][j],
        ns_comm[i][(j+y_zones/y_pack-1)%(y_zones/y_pack)],
        we_comm[i][j],
        we_comm[(i+x_zones/x_pack-1)%(x_zones/x_pack)][j]);
    }
  }
}
```

Literatur

[1] ALEXANDROV, A., M.F. IONESCU, K.E. SCHAUSER und C. SCHEIMAN: *LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation*. Journal of Parallel and Distributed Computing, 44(1):71–79, 1997.

[2] AMDAHL, G.M.: *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. In: *Proceedings of the '67 AFIPS Computer Conference*, Seiten 483–485. ACM, 1967.

[3] BANSAL, S., P. KUMAR und K. SINGH: *An Improved Two-step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines*. Parallel Computing, 32(10):759–774, 2006.

[4] BARKER, K., A.N. CHERNIKOV, N. CHRISOCHOIDES und K. PINGALI: *A Load Balancing Framework for Adaptive and Asynchronous Applications*. IEEE Transactions on Parallel and Distributed Systems, 15(2):183–192, 2004.

[5] BEN HASSEN, S., H.E. BAL und C.J.H. JACOBS: *A Task- and Data-parallel Programming Language Based on Shared Objects*. ACM Transactions on Programming Languages and Systems (TOPLAS), 20(6):1131–1170, 1998.

[6] BENKNER, S.: *Vienna Fortran 90 - An Advanced Data Parallel Language*. In: *Proc. of the 3rd International Conference on Parallel Computing Technologies (PaCT '95)*, Band 964 der Reihe *LNCS*, Seiten 142–156, London, UK, 1995. Springer-Verlag.

[7] BOUDET, V., F. DESPREZ und F. SUTER: *One-Step Algorithm for Mixed Data and Task Parallel Scheduling without Data Replication*. In: *Proc. of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. IEEE, 2003.

[8] BRANDES, T.: *Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library*. In: *Proc. of the 5th International Euro-Par Conference on Parallel Processing (Euro-Par '99)*, Band 1685 der Reihe *LNCS*, Seiten 833–844. Springer-Verlag, 1999.

[9] CARPENTER, B., G. ZHANG, G. FOX, X. LI und Y. WEN: *HPJava: Data Parallel Extensions to Java*. Concurrency: Practice and Experience, 10(11-13):873–877, 1998.

[10] CHANDY, M., I. FOSTER, K. KENNEDY, C. KOELBEL und C.-W. TSENG: *Integrated Support for Task and Data Parallelism*. The International Journal of Supercomputer Applications, 8(2):80–98, 1994.

[11] CHAPMAN, B.M., M. HAINES, P. MEHROTA, H.P. ZIMA und J. VAN ROSENDALE: *Opus: A Coordination Language for Multidisciplinary Applications*. Scientific Programming, 6(4):345–362, 1997.

[12] CHAPMAN, B.M., P. MEHROTRA, J. VAN ROSENDALE und H.P. ZIMA: *A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism*. In: *Proc. of the 3rd Joint International Conference on Vector and Parallel Processing (CONPAR '94)*, Band 854 der Reihe *LNCS*, Seiten 664–676. Springer-Verlag, 1994.

[13] CHAVARRIÍA-MIRANDA, D.G.: *Advanced Data-parallel Compilation*. PdD thesis, Rice University, Houston, TX, USA, 2004.

[14] CIARPAGLINI, S., L. FOLCHI, S. ORLANDO, S. PELAGATTI und R. PEREGO: *Integrating Task and Data Parallelism with taskHPF*. In: *Proc. of the International Conference on Parallel and Distributed Techniques and Applications (PDPTA '00)*, Seiten 2485–2491. CSREA Press, 2000.

[15] CULLER, D.E., R. KARP, D. PATTERSON, A. SAHAY, K.E. SCHAUSER, E. SANTOS, R. SUBRAMONIAN und T. VON EICKEN: *LogP: Towards a Realistic Model of Parallel Computation*. ACM SIGPLAN Notices, 28(7):1–12, 1993.

[16] DARTE, A., J. MELLOR-CRUMMEY, R. FOWLER und D. CHAVARRÍA-MIRANDA: *Generalized Multipartitioning of Multi-dimensional Arrays for Parallelizing Line-sweep Computations*. Journal of Parallel and Distributed Computing, 63(9):887–911, 2003.

[17] DEVINE, K., E. BOMAN, R. HEAPHY, B. HENDRICKSON, J. TERESCO, J. FAIK, J. FLAHERTY und L. GERVASIO: *New challenges in dynamic load balancing*. Applied Numerical Mathematics, 52(2-3):133–152, 2005.

[18] DEVINE, K., B. HENDRICKSON, E. BOMAN, M. ST. JOHN und C. VAUGHAN: *Design of Dynamic Load-balancing Tools for Parallel Applications*. In: *Proceedings of the 14th International Conference on Supercomputing (ICS '00)*, Seiten 110–118. ACM, 2000.

[19] DÍAZ, M., B. RUBIO, E. SOLER und J.M. TROYA: *A Border-based Coordination Language for Integrating Task and Data Parallelism*. Journal of Parallel Distributed Computing, 62(4):715–740, 2002.

[20] DÍAZ, M., B. RUBIO, E. SOLER und J.M. TROYA: *Domain Interaction Patterns to Coordinate HPF Tasks*. Parallel Computing, 29(7):925–951, 2003.

[21] DÍAZ, M., B. RUBIO, E. SOLER und J.M. TROYA: *SBASCO: Skeleton-Based Scientific Components*. In: *Proc. of the 12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP '04)*, Seiten 318–325. IEEE, 2004.

[22] DORTA, A.J., J.A. GONZÁLEZ, C. RODRÍGUEZ und F. DE SANDE: *llc: A Parallel Skeletal Language*. Parallel Processing Letters, 13(3):437–448, 2003.

[23] DORTA, A.J., P. LÓPEZ und F. DE SANDE: *Basic Skeletons in llc*. Parallel Computing, 32(7-8):491–506, 2006.

[24] DU, J. und J.Y.-T. LEUNG: *Complexity of Scheduling Parallel Task Systems*. SIAM Journal on Discrete Mathematics, 2(4):473–487, 1989.

[25] DÜMMLER, J., R. KUNIS und G. RÜNGER: *A Comparison of Scheduling Algorithms for Multiprocessortasks with Precedence Constraints*. In: *Proc. of the 2007 High Performance Computing & Simulation Conference (HPCS'07)*, Seiten 663–669. ECMS, 2007.

[26] DÜMMLER, J., R. KUNIS und G. RÜNGER: *A Scheduling Toolkit for Multiprocessor-Task Programming with Dependencies*. In: *Proc. of the 13th International Euro-Par Conference (Euro-Par '07)*, Band 4641 der Reihe *LNCS*, Seiten 23–32. Springer-Verlag, 2007.

[27] DÜMMLER, J., R. KUNIS und G. RÜNGER: *Layer-Based Scheduling Algorithms for Multiprocessor-Tasks with Precedence Constraints*. In: *Proc. of the 12th International Conference on Parallel Computing (ParCo '07)*, Band 15 der Reihe *Advances in Parallel Computing*, Seiten 321–328. IOS Press, 2007.

[28] DÜMMLER, J., T. RAUBER und G. RÜNGER: *Communicating Multiprocessor-Tasks*. In: *Proc. of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC '07)*, Band 5234 der Reihe *LNCS*, Seiten 292–307. Springer-Verlag, 2007.

[29] DÜMMLER, J., T. RAUBER und G. RÜNGER: *A Transformation Framework for Communicating Multiprocessor-Tasks*. In: *Proc. of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '08)*, Seiten 64–71. IEEE, 2008.

[30] DÜMMLER, J., T. RAUBER und G. RÜNGER: *Mapping Algorithms for Multiprocessor Tasks on Multi-Core Clusters*. In: *Proc. of the 37th International Conference on Parallel Processing (ICPP '08)*, Seiten 141–148. IEEE, 2008.

[31] DÜMMLER, J., T. RAUBER und G. RÜNGER: *Mixed Programming Models using Parallel Tasks*. In: LI, K.-C., C.-H. HSU, L. T. YANG, J. DONGARRA und H. ZIMA (Herausgeber): *Handbook of Research on Scalable Computing Technologies*, Seiten 246–275. Information Science Reference, 2009.

[32] DÜMMLER, J., T. RAUBER und G. RÜNGER: *Scalable Computing with Parallel Tasks*. In: *Proc. of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS '09)*. ACM, 2009.

[33] DUTOT, P.-F., T. N'TAKPE, F. SUTER und H. CASANOVA: *Scheduling Parallel Task Graphs on (Almost) Homogeneous Multicluster Platforms*. IEEE Transactions on Parallel and Distributed Systems, 20(7):940–952, 2009.

[34] F. SUTER, F. DESPREZ und H. CASANOVA: *From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling*. In: *Proc. of the 10th International Euro-Par Conference (Euro-Par '04)*, Band 3149 der Reihe *LNCS*, Seiten 230–237. Springer-Verlag, 2004.

[35] FINK, S.J.: *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. PhD thesis, University of California, San Diego, 1998.

[36] FISSGUS, U.: *Scheduling Using Genetic Algorithms*. In: *Proc. of the 20th International Conference on Distributed Computing Systems (ICDCS '00)*, Seiten 662–669. IEEE, 2000.

[37] FISSGUS, U.: *A Tool for Generating Programs with Mixed Task and Data Parallelism*. Doktorarbeit, Martin-Luther-Universität Halle-Wittenberg, 2001.

[38] FONLUPT, C., P. MARQUET und J.-L. DEKEYSER: *Data-Parallel Load Balancing Strategies*. Parallel Computing, 24(11):1665–1684, 1998.

[39] FOSTER, I., D.R. KOHR, R. KRISHNAIYER und A. CHOUDHARY: *Double Standards: Bringing Task Parallelism to HPF Via the Message Passing Interface*. In: *Proc. of the 1996 ACM/IEEE Conference on Supercomputing (SC '96)*. IEEE, 1996.

[40] FOSTER, I.T. und K.M. CHANDY: *Fortran M: A Language for Modular Parallel Programming*. Journal of Parallel and Distributed Computing, 26(1):24–35, 1995.

[41] FOX, G.C., S. HIRANANDANI, K. KENNEDY, C. KOELBEL, U. KREMER, C.-W. TSENG und M. WU: *The Fortran D Language Specification*. Technischer Bericht CRPC-TR90079, Center for Research on Parallel Computation, Rice University, 1991.

[42] GONZÁLEZ-ESCRIBANO, A., A.J.C. VAN GEMUND und V. CARDEÑOSO-PAYO: *Mapping Unstructured Applications into Nested Parallelism*. In: *Proc. of the International Conference on Vector and Parallel Processing (VECPAR '02)*, Band 2565 der Reihe *LNCS*, Seiten 407–420. Springer, 2002.

[43] GUO, M. und I. NAKATA: *A Framework for Efficient Data Redistribution on Distributed Memory Multicomputers*. The Journal of Supercomputing, 20(3):243–265, 2001.

[44] HAIRER, E., S.P. NØRSETT und G. WANNER: *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, Berlin Heidelberg New York, 1993.

[45] HIGH PERFORMANCE FORTRAN FORUM: *High Performance Fortran Language Specification 1.0*. Technischer Bericht CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993.

[46] HIGH PERFORMANCE FORTRAN FORUM: *High Performance Fortran Language Specification 2.0*. Technischer Bericht, Center for Research on Parallel Computation, Rice University, Houston, TX, 1997.

[47] HIRANANDANI, S., K. KENNEDY und C.-W. TSENG: *Compiling Fortran D for MIMD distributed-memory Machines*. Communications of the ACM, 35(8):66–80, 1992.

[48] HSU, C.-H., S.-W. BAI, Y.-C. CHUNG und C.-S. YANG: *A Generalized Basic-Cycle Calculation Method for Efficient Array Redistribution*. IEEE Transactions on Parallel and Distributed Systems, 11(12):1201–1216, 2000.

[49] HUNOLD, S., T. RAUBER und G. RÜNGER: *Dynamic Scheduling of Multi-processor Tasks on Clusters of Clusters*. In: *Proc. of the 2007 IEEE International Conference on Cluster Computing (CLUSTER '07)*, Seiten 507–514, Washington, DC, USA, 2007. IEEE Computer Society.

[50] HUNOLD, S., T. RAUBER und F. SUTER: *Redistribution Aware Two-step Scheduling for Mixed-parallel Applications*. In: *Proc. of the 2008 IEEE International Conference on Cluster Computing (CLUSTER '08)*, Seiten 50–58. IEEE, 2008.

[51] HUNOLD, S., T. RAUBER und F. SUTER: *Scheduling Dynamic Workflows onto Clusters of Clusters using Postponing*. In: *Proc. of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '08)*, Seiten 669–674, Washington, DC, USA, 2008. IEEE Computer Society.

[52] *Intel MPI Benchmark*. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[53] JANSEN, K. und H. ZHANG: *Scheduling Malleable Tasks with Precedence Constraints*. In: *Proc. of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '05)*, Seiten 86–95. ACM, 2005.

[54] JANSEN, K. und H. ZHANG: *An Approximation Algorithm for Scheduling Malleable Tasks under General Precedence Constraints*. ACM Transactions on Algorithms, 2(3):416–434, 2006.

[55] JIN, H., und R.F. VAN DER WIJNGAART: *Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks*. Journal of Parallel Distributed Computing, 66(5):674–685, 2006.

[56] JOISHA, P.G. und P. BANERJEE: *PARADIGM (version 2.0): A New HPF Compilation System*. In: *Proc. of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing (IPPS '99/SPDP '99)*, Seiten 609–615. IEEE, 1999.

[57] KALNS, E.T. und L.M. NI: *DaReL: a Portable Data Redistribution Library for Distributed-memory Machines*. In: *Proc. of the 1994 Scalable Parallel Libraries Conference (SPLC '94)*, Seiten 78–87. IEEE, 1994.

[58] KESSLER, C.W. und W. LÖWE: *A Framework for Performance-Aware Composition of Explicitly Parallel Components*. In: *Proc. of the 12th International Conference on Parallel Computing (ParCo '07)*, Band 15 der Reihe *Advances in Parallel Computing*, Seiten 227–234. IOS Press, 2007.

[59] KÜHNEMANN, M., T. RAUBER und G. RÜNGER: *A Source Code Analyzer for Performance Prediction*. In: *Proc. of the IPDPS '04 Workshop on Massively Parallel Processing (WMPP'04)*. IEEE, 2004.

[60] KÜHNEMANN, M., T. RAUBER und G. RÜNGER: *Performance Modelling for Task-Parallel Programs*. In: GERNDT, M., V. GETOV, A. HOISIE, A. MALONY und B. MILLER (Herausgeber): *Performance Analysis and Grid Computing*, Seiten 77–91. Kluwer, 2004.

[61] LAURE, E.: *OpusJava: a Java Framework for Distributed High Performance Computing*. Future Generation Computer Systems, 18(2):235–251, 2001.

[62] LAURE, E., P. MEHROTRA und H.P. ZIMA: *Opus: Heterogeneous Computing with Data Parallel Tasks*. Parallel Processing Letters, 9(2), 1999.

[63] LEPERE, R., G. MOUNIE und D. TRYSTRAM: *An Approximation Algorithm for Scheduling Trees of Malleable Tasks*. European Journal of Operational Research, 142:242–249, 2002.

[64] LEPERE, R., D. TRYSTRAM und G.J. WOEGINGER: *Approximation Algorithms for Scheduling Malleable Tasks Under Precedence Constraints.* International Journal of Foundation of Computer Science, 13(4):613–627, 2002.

[65] LUDWIG, W. und P. TIWARI: *Scheduling Malleable and Nonmalleable Parallel Tasks*. In: *Proc. of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '94)*, Seiten 167–176. SIAM, 1994.

[66] MELANCON, G., I. DUTOUR und M. BOUSQUET-MELOU: *Random Generation of Dags for Graph Drawing*. Technischer Bericht INS-R0005, CWI (Centre for Mathematics and Computer Science), 2000.

[67] MERLIN, J.H., S.B. BADEN, S. FINK und B.M. CHAPMAN: *Multiple Data Parallelism with HPF and KeLP*. Future Generation Computer Systems, 15(3):393–405, 1999.

[68] MESSAGE PASSING INTERFACE FORUM: *MPI: A Message-Passing Interface Standard*, 1994. `http://www.mpi-forum.org/docs/docs.html`.

[69] MOUNIE, G., C. RAPINE und D. TRYSTRAM: *Efficient Approximation Algorithms for Scheduling Malleable Tasks*. In: *Proc. of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*, Seiten 23–32. ACM, 1999.

[70] MOUNIE, G., C. RAPINE und D. TRYSTRAM: *A $\frac{3}{2}$-Approximation Algorithm for Scheduling Independent Monotonic Malleable Tasks*. SIAM Journal on Computing, 37(2):401–412, 2007.

[71] N'TAKPÉ, T. und F. SUTER: *Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms*. In: *Proc. of the 12th International Conference on Parallel and Distributed Systems (ICPADS '06)*, Seiten 3–10, 2006.

[72] N'TAKPÉ, T., F. SUTER und H. CASANOVA: *A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms*. In: *6th International Symposium on Parallel and Distributed Computing*, Hagenberg, Austria, Juli 2007. IEEE.

[73] O'DONNELL, J., T. RAUBER und G. RÜNGER: *Functional Realization of Coordination Environments for Mixed Parallelism*. In: *Proc. of the IPDPS'04 Workshop on Advances in Parallel and Distributed Computational Models (APDCM '04)*, CD-ROM. IEEE, 2004.

[74] ORLANDO, S., P. PALMERINI und R. PEREGO: *Coordinating HPF programs to Mix Task and Data Parallelism*. In: *Proc. of the 2000 ACM Symposium on Applied Computing (SAC '00)*, Seiten 240–247. ACM, 2000.

[75] ORLANDO, S. und R. PEREGO: *COLTHPF, a Run-time Support for the High-level Co-ordination of HPF tasks*. Concurrency - Practice and Experience, 11(8):407–434, 1999.

[76] OSMAN, A. und H. AMMAR: *Dynamic Load Balancing Strategies for Parallel Computers*. In: *Proc. of the International Symposium on Parallel and Distributed Computing (ISPDC '02)*, 2002.

[77] PARK, N., V.K. PRASANNA und C.S. RAGHAVENDRA: *Efficient Algorithms for Block-Cyclic Array Redistribution Between Processor Sets*. IEEE Transactions on Parallel and Distributed Systems, 10(12):1217–1240, 1999.

[78] PELAGATTI, S.: *Task and Data Parallelism in P3L*. In: RABHI, F.A. und S. GORLATCH (Herausgeber): *Patterns and Skeletons for Parallel and Distributed Computing*, Seiten 155–186. Springer-Verlag, 2003.

[79] PELAGATTI, S. und D.B. SKILLICORN: *Coordinating Programs in the Network of Tasks Model*. Journal of Systems Integration, 10(2):107–126, 2001.

[80] PLASTINO, A., C.C. RIBEIRO und N. RODRIGUEZ: *Developing SPMD Applications with Load Balancing*. Parallel Computing, 29(6):743–766, 2003.

[81] PRYLLI, L. und B. TOURANCHEAU: *Fast Runtime Block Cyclic Data Redistribution on Multiprocessors*. Journal of Parallel and Distributed Computing, 45(1):63–72, 1997.

[82] RADULESCU, A., C. NICOLESCU, A.J.C. VAN GEMUND und P. JONKER: *CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems*. In: *Proc. of the 15th International Parallel & Distributed Processing Symposium (IPDPS '01)*, Seiten 39–46. IEEE, 2001.

[83] RADULESCU, A. und A.J.C. VAN GEMUND: *A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling*. In: *Proc. of the International Conference on Parallel Processing (ICPP '01)*, Seiten 69–76. IEEE, 2001.

[84] RAMASWAMY, S.: *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Computations*. PhD thesis, University of Illinois at Urbana–Champaign, 1996.

[85] RAMASWAMY, S., S. SAPATNEKAR und P. BANERJEE: *A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers*. IEEE Transactions on Parallel Distributed Systems, 8(11):1098–1116, 1997.

[86] RAMASWAMY, S., B. SIMONS und P. BANERJEE: *Optimizations for Efficient Array Redistribution on Distributed Memory Multicomputers*. Journal of Parallel and Distributed Computing, 38(2):217–228, 1996.

[87] RAUBER, T., R. REILEIN-RUSS und G. RÜNGER: *Group-SPMD Programming with Orthogonal Processor Groups*. Concurrency and Computation: Practice and Experience, Special Issue on Compilers for Parallel Computers, 16(2-3):173–195, 2004.

[88] RAUBER, T., R. REILEIN-RUSS und G. RÜNGER: *On Compiler Support for Mixed Task and Data Parallelism*. In: *Proc. of the 10th International Conference on Parallel Computing (ParCo'03)*, Parallel Computing: Software Technology, Algorithms, Architectures & Applications, Seiten 23–30. Elsevier, 2004.

[89] RAUBER, T. und G. RÜNGER: *Parallel Solution of a Schrödinger-Poisson System*. In: *Proc. of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe '95)*, Band 919 der Reihe *LNCS*, Seiten 697–702. Springer-Verlag, 1995.

[90] RAUBER, T. und G. RÜNGER: *The Compiler TwoL for the Design of Parallel Implementations*. In: *Proc. of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, Seiten 292–301. IEEE, 1996.

[91] RAUBER, T. und G. RÜNGER: *Compiler Support for Task Scheduling in Hierarchical Execution Models*. Journal of Systems Architecture, 45(6-7):483–503, 1998.

[92] RAUBER, T. und G. RÜNGER: *A Coordination Language for Mixed Task and Data Parallel Programs*. In: *Proc. of the 13th Annual ACM Symposium on Applied Computing (SAC'99)*, Seiten 146–155. ACM, 1999.

[93] RAUBER, T. und G. RÜNGER: *Parallel Execution of Embedded and Iterated Runge-Kutta Methods*. Concurrency - Practice and Experience, 11(7):367–385, 1999.

[94] RAUBER, T. und G. RÜNGER: *Scheduling of Data Parallel Modules for Scientific Computing.* In: *Proc. of the 9th SIAM Conference on Parallel Processing for Scientific Computing (PPSC '99)*. SIAM, 1999.

[95] RAUBER, T. und G. RÜNGER: *A Transformation Approach to Derive Efficient Parallel Implementations*. IEEE Transactions on Software Engineering, 26(4):315–339, 2000.

[96] RAUBER, T. und G. RÜNGER: *Deriving Array Distributions by Optimization Techniques*. Journal of Supercomputing, 15(3):271–293, 2000.

[97] RAUBER, T. und G. RÜNGER: *Parallele und verteilte Programmierung*. Springer-Verlag, Berlin Heidelberg New York, 2000.

[98] RAUBER, T. und G. RÜNGER: *Tlib - A Library to Support Programming with Hierarchical Multi-processor Tasks*. Journal of Parallel and Distributed Computing, 65(3):347–360, 2005.

[99] RAUBER, T. und G. RÜNGER: *A Data Re-Distribution Library for Multi-Processor Task Programming*. International Journal of Foundations of Computer Science, 17(2):251–270, 2006.

[100] RAUBER, T. und G. RÜNGER: *Mixed Task and Data Parallel Executions in General Linear Methods*. Scientific Programming, 15(3):137–155, 2007.

[101] RAUBER, T., G. RÜNGER und R. WILHELM: *Deriving Optimal Data Distributions for Group Parallel Numerical Algorithms*. In: *Proc. of the Conference on Programming Models for Massively Parallel Computers (PMMP '95)*, Seiten 33–41. IEEE, 1995.

[102] REILEIN-RUSS, R.: *Eine komponentenbasierte Realisierung der TwoL Spracharchitektur*. Doktorarbeit, TU Chemnitz, Fakultät für Informatik, 2005.

[103] SEDGEWICK, R.: *Algorithmen*. Addison-Wesley, München, 2002. 2. Auflage.

[104] SHIH, K.-P.: *Efficient Computation and Communication Sets Generations for Data-Parallel Programs on Multicomputers*. PhD thesis, Department of Computer Science and Information Engineering, National Central University, Taiwan, 1998.

[105] SKILLICORN, D.B.: *The Network of Tasks Model*. Technischer Bericht TR1999-427, Queen's University, 1999.

[106] SUBHLOK, J. und G. VONDRAN: *Optimal Mapping of Sequences of Data Parallel Tasks*. ACM SIGPLAN Notices, 30(8):134–143, 1995.

[107] SUBHLOK, J. und B. YANG: *A new Model for Integrated Nested Task and Data Parallel Programming*. In: *Proc. of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '97)*, Seiten 1–12. ACM, 1997.

[108] SUTER, F., H. CASANOVA, F. DESPREZ und V. BOUDET: *From Heterogeneous Task Scheduling to Heterogeneous Mixed Data and Task Parallel Scheduling*. Technischer Bericht RR2003-52, Laboratoire de l'Informatique du Parallélisme (LIP), 2003.

[109] VALDES, J., R.E. TARJAN und E.L. LAWLER: *The Recognition of Series Parallel Digraphs*. In: *Proc. of the 11th Annual ACM Symposium on Theory of Computing (STOC '79)*, Seiten 1–12. ACM, 1979.

[110] VALIANT, L.G.: *A Bridging Model for Parallel Computation*. Communications of the ACM, 33(8):103–111, 1990.

[111] VAN DER HOUWEN, P.J. und E. MESSINA: *Parallel Adams Methods*. Journal of Computational and Applied Mathematics, 101:153–165, 1999.

[112] VAN DER HOUWEN, P.J. und B.P. SOMMEIJER: *Iterated Runge-Kutta Methods on Parallel Computers*. SIAM Journal on Scientific and Statistical Computing, 12(5):1000–1028, 1991.

[113] VAN DER WIJNGAART, R.F. und H. JIN: *The NAS Parallel Benchmarks, Multi-Zone Versions*. Technischer Bericht NAS-03-010, NASA Ames Research Center, 2003.

[114] VAN DONGEN, V., C. BONELLO und G.R. GAO: *Data Parallelism with High Performance C*. In: *Proc. of the 1994 conference of the Centre for Advanced Studies on Collaborative Research (CASCON '94)*. IBM Press, 1994.

[115] VANNESCHI, M.: *The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications*. Parallel Computing, 28(12):1709–1732, 2002.

[116] VANNESCHI, M. und L. VERALDI: *Dynamicity in Distributed Applications: Issues, Problems and the ASSIST Approach*. Parallel Computing, 33(12):822–845, 2007.

[117] VYDYANATHAN, N., S. KRISHNAMOORTHY, G. SABIN, Ü.V. ÇATALYÜREK, T.M. KURÇ, P. SADAYAPPAN und J.H. SALTZ: *An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications*. In: *Proc. of the 2006 International Conference on Parallel Processing (ICPP '06)*, Seiten 443–450. IEEE, 2006.

[118] VYDYANATHAN, N., S. KRISHNAMOORTHY, G. SABIN, Ü.V. ÇATALYÜREK, T.M. KURÇ, P. SADAYAPPAN und J.H. SALTZ: *An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications*. IEEE Transactions on Parallel and Distributed Systems, 20(8):1158–1172, 2009.

[119] WALKER, D.W. und S.W. OTTO: *Redistribution of Block-cyclic Data Distributions using MPI*. Concurrency: Practice and Experience, 8(9):707–728, 1996.

[120] WEST, E.A. und A.S. GRIMSHAW: *Braid: Integrating Task and Data Parallelism*. In: *Proc. of the 5th Symposium on the Frontiers of Massively Parallel Computation (FRONTIERS '95)*, Seiten 211–219. IEEE, 1995.

[121] ZIMMERMANN, W. und W. LÖWE: *Foundations for the Integration of Scheduling Techniques into Compilers for Parallel Languages*. International Journal of Computational Science and Engineering, 1(2-4):99–109, 2005.

Index

Abarbeitungsplan, 23
Abstrakter Syntaxbaum, 81
Äquivalenz der Typen, 41
Allokations- und Schedulingalgorithmen, 34
Analysephase, 10, 81–90
Arithmetischer Ausdruck, 151
Ausgabedatenverteilung, 104
Ausgabeparameter, 6, 40

Basisdatentyp, 38
Basismodul, 9
  Definition, 39–40, 153
  Implementierung, 76–77
Bezeichner, 151
Bibliotheksbasierter Ansatz, 17
Blockstruktur, 111
BT-MZ Benchmark, 125

C-Relation, 7
CHiC Cluster, 119
CM-task, 6
CM-task Compiler, 9
CM-task Compilerframework, 8–12
CM-task Graph, 7
CM-task Modell, 5–8
CM-task Programm, 6
CM-task Schedule, 24
  gültig, 25
Codegenerierungsphase, 11, 110–113
cpar-Konstruktor, 41, 42, 57
cparfor-Konstruktor, 41, 42, 57

Datenabhängigkeit, 85
  Annotation, 55
Datenparallelität, 1
Datentyp, 6
Datentypdefinition, 38, 153
Datenumverteilungsbibliothek, 12, 70–74, 113
Datenumverteilungsoperation, 7
  dynamisch, 73
  statisch, 72
Datenverteilungsphase, 11, 104–110
Datenverteilungstyp, 6
Datenverteilungstypdefinition, 38–39, 153

Eingabedatenverteilung, 104
Eingabeparameter, 6, 40
Empfangsphase, 70
Endknoten, 90
Erweiterte Spezifikationssprache, 53–55, 154
Erweiterter SP-Graph, 42
Erweitertes Rahmenprogramm, 11, 60–63, 156
Erweitertes Spezifikationsprogramm, 10, 53
Externe Kommunikation, 6
Externer Kommunikator, 12, 63

Felddatentyp, 38
Felddatenverteilung, 38
for-Konstruktor, 40, 41, 57

Gültigkeit von Bezeichnern, 152
Gemischte Parallelität, 2
Gemischtes Mapping, 137
Gitterbasierte Kommunikation, 45
Gruppenausgleich, 30
Gruppenkommunikator, 12, 63
Gültiger CM-task Schedule, 25

Heterogener Cluster, 119

Identifikator, 53, 82
if-Konstruktor, 40
Interne Kommunikation, 6
IRK Verfahren, 47, 121
  Analysephase, 85, 88
  Erweitertes Rahmenprogramm, 61
  Erweitertes Spezifikationsprogramm, 55
  Koordinationsprogramm, 65
  Rahmenprogramm, 59
  semi-dynamisches Ausführungsschema, 112
  Spezifikationsprogramm, 49
Iterierte Runge-Kutta Verfahren, *siehe* IRK Verfahren

JuRoPA Cluster, 119

Kommentar, 151
Kommunikationsabhängigkeit, 43, 83
    Annotation, 53
Kommunikationsbeziehung, *siehe* C-Relation
Kommunikationsmenge, 83
Kommunikationsmuster I, 44
Kommunikationsmuster II, 44
Kommunikationsmuster III, 44
Kommunikationsmuster IV, 45
Kommunikationsmuster V, 46
Kommunikationsmuster VI, 46
Kommunikationsparameter, 40, 83
Kommunikationsstruktur, 63
Konsekutives Mapping, 136
Konstante, 151
Konstantendefinition, 37–38, 153
Konstruktoren
    der Spezifikationssprache, 41
    des Rahmenprogrammes, 57
    Syntax, 40
Koordinationsbasierter Ansatz, 19
Koordinationsfunktion, 63
Koordinationsprogramm, 11, 63–70
Koordinationsstruktur, 5
Koordinationsvariable, 66
Kopieannotation, 101
Kostenmodell, 51–52

Lastausgleichsalgorithmus, 75, 114–116
Lastausgleichsannotation, 61
Lastausgleichsbibliothek, 12, 74–76
Lastausgleichsgranularität, 61, 109
Laufzeitformel, 9
Lesemenge, 85
Logischer Ausdruck, 152
LU-MZ Benchmark, 123–124

M-task, *siehe* Paralleler Task
Managementfunktion, 75
Mapping, 136
Master/Slave, 44
Modulausdruck, 9, 40
Modulausführungsteil, 63
    semi-dynamisch, 69
    statisch, 64
MPI Bibliothek, 1

Namensraum, 152
Nutzerdatentyp, 38
Nutzerdatenverteilung, 39

Orthogonale Kommunikation, 46

P-Relation, 7
PAB Methode, 121
PABM Methode, 121
par-Konstruktor, 40, 41, 57
Parallele Adams Verfahren, 121–123
Paralleler Task, 2, 5
Parametrisierter Datenverteilungsvektor, 38
parfor-Konstruktor, 40, 41, 57
Pipeline, 44
Plattformbeschreibung, 52–53
Plattformbeschreibungssprache, 9, 157
Programmblock, 90
Prozessorgruppe
    Annotation, 58
    Darstellung, 58

Rahmenprogramm, 11, 57–60, 155

Schedule, 23
Schedulestruktur, 74, 111
Schedulingphase, 10, 90–104
Schicht, 29
Schichtenbasierte Schedulingalgorithmen, 35
Schichtstruktur, 111, 165
Schlüsselwort, 151
Schlüsselwort
    block, 39
    blockcyclic, 39
    cmgraph, 40
    cmmain, 40
    cmtask, 39
    const, 37
    crels, 55
    cyclic, 39
    distrib, 39
    on, 58, 60
    prels, 55
    rebalance, 61
    redistr, 60
    replic, 39

runtime, 40
type, 38
Schreibmenge, 85
Semi-dynamischer Compileransatz, 10
Sendephase, 70
seq-Konstruktor, 40, 41, 57
SGI Altix 4700, 119
Skeletonbasierter Ansatz, 15
SP-MZ Benchmark, 124–125
Spezifikationssprache, 9, 151
Spracherweiterung, 12
Startknoten, 90
Statischer Compileransatz, 10
Supertask, 25, 26
Supertask Allokation, 27
Supertask Graph, 26
Symbolische Laufzeitformel, 51

Task, 1
Taskparallelität, 1
Taskstruktur, 111, 164

Umverteilungsstruktur, 72

Verbundmodul, 9
Verbundmoduldefinition, 40–42, 153
Verteiltes Mapping, 137
Virtuelle Parameterliste, 86
Vorrangbeziehung, *siehe* P-Relation

while-Konstruktor, 40, 41, 57

Zugriffsdimension, 41
Zugriffstyp, 40