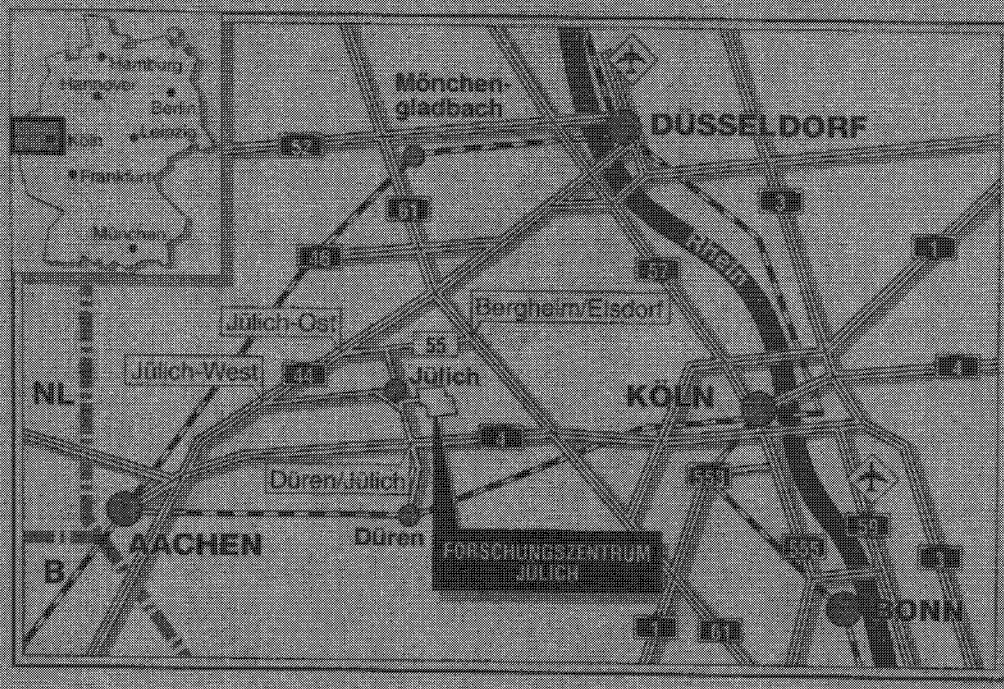
Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Partitionierung unstrukturierter Gitter für Shared-Virtual-Memory-Rechner

Oliver Weiß

Jül-3636



Berichte des Forschungszentrums Jülich ; 3336
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3336

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461/61-6102 · Telefax: 02461/61-6103 · e-mail: zb-publikation@fz-juelich.de

Kurzfassung

Simulationsrechnungen sind ein unverzichtbares Werkzeug zur Lösung von wissenschaftlichen und technischen Problemstellungen. Sie basieren auf Systemen von Partiellen Differentialgleichungen, die auf regulären oder unstrukturierten Gittern diskretisiert werden. Die Parallelisierung der Lösungsverfahren erfolgt über eine Zerlegung des Gitters und die Verteilung der Teilgitter auf die Prozessoren.

Zur Unterstützung der Programmierung von Parallelrechnern wurde die Programmierumgebung von SVM-Fortran entwickelt, die auf dem Konzept des virtuell gemeinsamen Speichers *(Shared Virtual Memory)* beruht.

SVM-Fortran stellt Möglichkeiten für die Zerlegung regulärer Gitter bereits zur Verfügung, d. h. die Gitterpunkte können blockweise oder zyklisch verteilt werden. Unstrukturierte Gitter lassen sich so nur unzureichend abbilden. Stattdessen müssen Partitionierungsalgorithmen eingesetzt werden, die bei der Verteilung der Gitterpunkte Nachbarschaftsverhältnisse und/oder räumliche Gegebenheiten berücksichtigen und so zu adäquateren Verteilungen führen.

Diese Arbeit gibt einen Überblick über die heutzutage eingesetzten Partitionierer. Zur Einbindung der Partitionierungsverfahren wird SVM-Fortran um geeignete Anweisungen erweitert. Die Basis für die Realisierung der Anweisungen im SVM-Fortran-Compiler wird durch die in dieser Arbeit entwickelte Laufzeitbibliothek gebildet, die eine effiziente Verwaltung der Gitterinformation und eine Schnittstelle zur Einbindung externer Partitionierungsalgorithmen bereitstellt.

Abstract

Simulations are an essential tool for solving scientific and technical problems. They are based on systems of partial differential equations which lead to regular or irregular meshes. The parallelisation of the solution algorithm is done by a mesh decomposition and a distribution of the submeshes to the processors of the parallel computer.

For supporting the programming of parallel computers, the SVM-Fortran programming environment is developed, based on the Shared Virtual Memory concept.

SVM-Fortran already provides possibilities for the decomposition of regular meshes, that means that the mesh points may be distributed blocked or cyclic. Based on these techniques, irregular grids are mapped inadequately. Better mappings can be determined with partitioning algorithms which take the connectivity and/or spatial information of the grid points into account.

This work presents an overview of the most important partitioning algorithms and describes the language features of SVM-Fortran supporting unstructured meshes. It also outlines the concepts of the implemented runtime system that allows to realise the language features efficiently.

Wie stolz Sie auf Ihre Logik sind. Sie sprechen nur noch von logischen Widerlegungen, Ihre Briefe sind strenge Argumentationen, Sie bringen „also“ und „folglich“ hinein, was das Vergnügen , das ich beim Lesen finden könnte, sehr beeinträchtigt. Verzichten Sie doch auf die Logik! Sie hat nie jemanden auch nur einen Schritt weitergebracht. Suchen Sie die Widersprüche soweit wie möglich zu vermeiden, aber wenn Sie mal einen finden, haben Sie keine Angst davor, sie beißen nicht.

Jean-Paul Sartre, Briefe an Simone de Beauvoir Band I (1926 – 1939)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Rechenintensive Problemstellungen, beispielsweise aus den Bereichen der Strömungssimulation, der Molekulardynamik, der Klimaberechnung und des Chipdesigns verlangen nach immer leistungsfähigeren Rechnern. Parallelrechner mit einer großen Anzahl von Prozessoren – heutzutage etwa 100 bis einige 1000 – bieten diese Leistungsfähigkeit.

Zwei grundsätzlich verschiedene Architekturansätze bildeten sich in der Vergangenheit heraus: Rechner mit einem gemeinsamen Speicher und Rechner mit verteiltem Speicher. Bei Rechnern mit gemeinsamem Speicher werden Synchronisation und Kommunikation zwischen Prozessoren durch gemeinsame Variablen im Hauptspeicher realisiert. Ein Problem bei dieser Rechnerart ist, daß die Speicherzugriffe schon bei wenigen Prozessoren zum Engpaß werden: Die Prozessoren greifen über ein Verbindungsnetzwerk auf den Hauptspeicher zu, so daß Zugriffskonflikte entstehen.

Rechner mit verteiltem Speicher arbeiten auf ihrem lokalen Speicher. Die Daten werden durch Nachrichtenaustausch *(message passing)* zwischen den Prozessoren versandt. Da Zugriffe auf entfernt gespeicherte Daten merklich langsamer als Zugriffe auf lokale Daten sind, muß besonders auf die Lokalität der Programme geachtet werden. Der Programmierer muß üblicherweise den Nachrichtenaustausch zwischen den Prozessoren explizit ausprogrammieren. Er muß sich deshalb die Datenverteilung bewußt machen und sie beachten.

Neben dieser Klassifizierung nach der Zuordnung des Speichers zu den Prozessoren, wird auch zwischen gemeinsamem und privatem Adreßraum unterschieden. Parallelrechner mit gemeinsamem Speicher haben natürlicherweise auch einen gemeinsamen, globalen Adreßraum. Speicheradressen sind somit eindeutig vergeben. Aufbauend auf verteiltem Speicher erzeugen Rechner mit verteiltem gemeinsamem Speicher *(distributed shared memory systems)* durch Hardware- oder Softwareunterstützung einen globalen Adreßraum. Eine Klasse solcher Rechner stellen die Rechner mit gemeinsamem

virtuellem Speicher (*shared virtual memory systems,* SVM-Rechner) dar. Bei diesen Rechnern werden die privaten Adreßräume der Prozessoren virtuell und für das Anwendungsprogramm transparent zu einem gemeinsamen Speicher mit globalen Adressen zusammengefügt.

Wesentlich ist hier, daß die Zuordnung von Speicheradressen und physikalischem Speicher nicht mehr statisch ist. Werden im Laufe eines Programms Daten eines anderen Prozessors benötigt, so migrieren sie zwischen den Prozessoren. Aus diesem Grunde kann die Lokalität der Datenzugriffe durch die Wiederverwendung der Daten erreicht werden.

Da nun wieder der einfach zu handhabende globale Adreßraum zur Verfügung steht, kann das dafür gewohnte Programmiermodell verwendet werden. Dadurch wird die Programmierung dieser Architekturen vereinfacht.

Das im Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich entwickelte SVM-Fortran ist eine Sprache für SVM-Rechner. Es handelt sich dabei um eine Erweiterung der Sprache FORTRAN 77. Hinzugekommen sind Direktiven, die Datenparallelismus, grobgranularen funktionalen Parallelismus und die Spezifizierung der Arbeitsverteilung auf die Prozessoren ermöglichen. Anders als bei Ansätzen, die die Daten einer Anwendung explizit verteilen, werden in SVM-Fortran datenparallele oder funktional parallele Teile des Programmes auf die Prozessoren der Anwendung verteilt.

SVM-Fortran unterstützt gezielt die Programmierung von gitterbasierten Anwendungen. Die Aufgabenstellungen, die in den genannten Bereichen untersucht werden, lassen sich häufig mit Partiellen Differentialgleichungen beschreiben. Die Parallelisierung dieser Anwendungen erfolgt durch Gebietszerlegung. Je nach Aufgabenstellung basiert die Diskretisierung der Differentialgleichungen auf regulären Gittern (z. B. zur Simulation des atmosphärischen Flusses, *Shallow Water Equations*) oder irregulären, also unstrukturierten Gittern. In den Gitterknoten werden Berechnungen ausgeführt. Verbindungen stellen die Beziehungen zwischen den Knoten dar.

Unstrukturierte Gitter erhält man durch die Finite-Element- bzw. Finite-Volumen-Methode, z. B. zur Berechnung der Umströmung einer Tragfläche. Es wird versucht, sich unregelmäßigen Formen mit geometrisch leicht faßbaren Figuren (Dreieck, Würfel usw.) anzunähern und die unregelmäßige Form möglichst exakt auszufüllen. Wird das Gitter während der Berechnung an die sich verändernden Anforderungen angepaßt, so spricht man von adaptiven Gittern. Dies ist beispielsweise der Fall, wenn das Gitter bei der Strömungssimulation entsprechend der entstehenden Turbulenzen an der Tragfläche verfeinert wird.

Durch Gebietszerlegungsverfahren werden Teilgitter erzeugt, die den Prozessoren zugeordnet werden. Üblicherweise müssen dazu die Daten über das Gitter aus der Anwendung extrahiert und in einem spezifischen Format abge-

speichert werden, damit ein externes Partitionierungswerkzeug die Zerlegung durchführen kann. Das Ergebnis dieser Partitionierung kann dann wieder eingelesen und eine daran angepaßte Arbeitsverteilung gewählt werden.

In SVM-Fortran wird eine Arbeitsverteilung durch sogenannte Templates beschrieben. Sie dienen dazu, die gewählte Gebietszerlegung zu realisieren. Momentan sind Templates implementiert, die eine Abbildung von rechteckigen Indexbereichen auf die einzelnen Prozessoren ermöglichen. Im Rahmen der vorliegenden Arbeit sollen die Möglichkeiten zur Modellierung in SVM-Fortran um unstrukturierte Templates erweitert werden. Damit wird es möglich, geeignete Partitionierer zur Laufzeit auf das Gitter anzuwenden. Dann können auch dynamische Umverteilung bei adaptiven Gittern durchgeführt und Werkzeuge zur Leistungsanalyse mit diesen Daten versorgt werden, ohne Schnittstellen zu externen Partitionierern bereitstellen zu müssen.

Ziel der Arbeit ist es, eine Laufzeitbibliothek zur Verfügung zu stellen, die als Basis einer Implementierung von Direktiven für die Spezifizierung von unstrukturierten Templates in SVM-Fortran dient. Die Arbeit gliedert sich wie folgt:

In Kapitel 2 werden Grundlagen des Prinzips des verteilten gemeinsamen Speichers und der Programmiersprache SVM-Fortran vorgestellt. Auch auf die werkzeugunterstützte Leistungsanalyse wird eingegangen.

Kapitel 3 schildert die Schritte und Erfahrungen, die bei der Parallelisierung eines sequentiellen Testprogramms (AVL-FIRE-Benchmark) gemacht wurden. Hier wurde die Parallelisierung durch reguläre Templates realisiert.

In Kapitel 4 wird ein Überblick über die verbreiteten Partitionierungs- und Optimierungsalgorithmen gegeben. Ein Abschnitt widmet sich speziell der Partitionierung adaptiver Gitter. Da Partitionierer üblicherweise keine Rücksicht auf Speicherseiten nehmen, werden zwei Ansätze vorgestellt, die dieses Problem zu lösen versuchen.

Die zugrundeliegenden Datenstrukturen und Funktionen der Laufzeitbibliothek werden in Kapitel 5 beschrieben. Ein Abschnitt erläutert die Änderungen, die in der dynamischen Speicherverwaltung des SVM-Fortran-Laufzeitsystems vorgenommen wurden.

Kapitel 6 demonstriert den Einsatz unstrukturierter Gitter. Hier wird der AVL-FIRE-Benchmark mit den neu definierten unstrukturierten Templates parallelisiert.

Kapitel 2

SVM-Rechner und ihre Programmierung

Rechner mit gemeinsamem virtuellem Speicher (*shared virtual memory systems,* SVM-Rechner) stellen eine spezielle Klasse von Parallelrechnern dar. Es handelt sich dabei um Rechner mit physikalisch verteiltem Speicher, der mit Hilfe von Hard- oder Software als gemeinsamer Speicher mit globalem Adreßraum erscheint. Das *Advanced Shared Virtual Memory System* (ASVM-System) der Intel Paragon ist ein Beispiel für eine Softwarelösung.

Die SVM-Fortran-Entwicklungsumgebung wird für diese Systeme entwickelt. Zur Entwicklungsumgebung gehört der Compiler, OPAL, ein Werkzeug zur Leistungsanalyse, und der Monitor SAM, der zur Laufzeit Leistungsinformationen sammelt.

2.1 Architekturen

Die Architekturen von Parallelrechnern lassen sich grob nach der physischen Anordnung des Speichers und der Art des Adreßraums einteilen. Greifen mehrere Prozessoren auf den gleichen Speicher zu, so spricht man von Rechnern mit gemeinsamem Speicher *(shared memory systems)*. Rechner bei denen der Arbeitsspeicher auf die Prozessoren verteilt wird, bilden die Gruppe der *distributed memory systems*.

Der Adreßraum kann gemeinsam oder privat vorgesehen werden. Hat man einen gemeinsamen Adreßraum, so sind die Speicheradressen global. Jeder Prozessor kann auf jede Adresse zugreifen. Hat jeder Prozessor seine eigenen privaten Adressen, so kann der Austausch von Daten nur über das Versenden von Nachrichten *(message passing)* erfolgen. Ein direkter Zugriff auf den Speicher anderer Prozessoren ist nicht möglich.

2.1.1 Rechner mit verteiltem gemeinsamem Speicher

Die hier weiter verfolgte Klasse sind Rechner mit verteiltem gemeinsamem Speicher *(distributed shared memory systems)*, also Systeme mit verteiltem Speicher und gemeinsamem Adreßraum. Sie läßt sich weiter unterteilen. Systeme, bei denen die Zuordnung von Adressen zu Speicherbereichen statisch bleibt, ordnen einer virtuellen Adresse einen festen Speicherbereich zu, welchen diese bis zur Beendigung der Bearbeitung behält. Der zusammenhängende Adreßraum ist auf die Prozessoren aufgeteilt, und durch die Adresse ergibt sich implizit der Prozessor, in dessen Speicher die Adresse abgebildet wird. Durch einen Cache kann erreicht werden, daß die Anzahl der langsamen entfernten Zugriffe verringert wird.

In *Rechnern mit gemeinsamem virtuellem Speicher* (*shared virtual memory systems,* SVM-Rechner) ist die Zuordnung der Adressen dynamisch. Werden von einem Prozessor Daten angefordert, so wird die gesamte Speicherseite, die diese enthält, verschoben. Unter der Annahme, daß das Programm ein gewisses Lokalitätsverhalten aufweist, können so häufige Zugriffe auf entfernt gespeicherte Daten vermieden werden. Sowohl Implementierungen in Hardware (Kendall Square Research KSR1 und KSR2) als auch in Software sind möglich.

Generell wird zwischen Schreib- und Lesewünschen auf entfernte Daten unterschieden, die entsprechend dem Kohärenzprotokoll behandelt werden. Will ein Prozessor lesend auf ein Datum zugreifen *(Lese-Seitenfehler),* so wird eine Kopie der gesamten Speicherseite angefertigt und in den privaten Speicher des Prozessors übertragen. Auf diese Kopie kann der Prozessor *nur lesend* zugreifen. Auch der Ursprungsprozessor darf nur noch lesend auf seine Kopie zugreifen. Versucht er auf die Seite zu schreiben, erzeugt er einen Schreib-Seitenfehler. Ohne weiteres können mehrere Lesekopien gleichzeitig nebeneinander existieren.

Will ein Prozessor allerdings schreibend auf eine Seite zugreifen *(Schreib-Seitenfehler),* so müssen alle Lesekopien invalidiert werden, d. h. auch ein lesender Zugriff ist auf diese Kopien nicht mehr erlaubt und führt beim nächsten Zugriffsversuch zu einem Lese-Seitenfehler. Die Kopie im vorher besitzenden Prozessor wird an den anfordernden Prozessor übertragen und im Ursprungsprozessor gelöscht. Es existiert also nur noch die Schreibkopie im anfordernden Prozessor.

Da die Seiten keinem festen Speicherbereich mehr zugeordnet sind, müssen Strategien zum Auffinden der gewünschten Seiten implementiert werden. Diese Aufgabe übernehmen *Mangageralgorithmen.* Der *Manager* verwaltet die Daten, die zum Lokalisieren einer Seite benötigt werden.

Einfach zu realisieren ist ein zentraler Manager *(Centralized Manager).*

Ein Prozessor übernimmt die Managerrolle und führt die Verteilung der Seiten nach. Benötigt ein Prozessor eine Seite, so fordert er diese vom zentralen Managerprozessor an. Dieser schickt an den momentanen Besitzer der gewünschten Seite eine Nachricht, diese zu versenden. Der Manager wird bei größeren Rechnern schnell zum Engpaß, so daß diese Methode nicht skaliert.

Um diesen Engpaß zu umgehen, kann man versuchen, die Manageraufgabe auf alle Prozessoren zu verteilten. Beim *Broadcast Distributed Manager* (BDM) ist der Besitzer einer Seite auch immer ihr Manager. Der Manager wird hier durch eine Nachricht an alle Prozessoren (Broadcast) gefunden. Der große Kommunikationsaufwand ist ein Nachteil dieses Algorithmus.

Die globale Kommunikation wird umgangen, wenn die Zuständigkeit für die Seiten nach einem festen Schema verteilt wird (*Fixed Distributed Manager*, FDM). Verteilt man sie z. B. blockweise auf die Prozessoren, so läßt sich der Manager zum einen durch eine einfache Formel bestimmen, und zum anderen ist die Belastung besser auf alle Prozessoren verteilt, als beim zentralen Management.

Da bei diesem Algorithmus der Manager nicht notwendigerweise der Besitzer einer Seite ist, sind für einen Seitenaustausch drei Nachrichten zu versenden. Da in SVM-Rechnern während der Ausführung die Seiten migrieren, sollte auch der Manager entsprechend wechseln.

Der *Dynamic-Distributed-Manager*-Algorithmus (DDM-Algorithmus) erreicht durch Verzeigerung, daß der Manager einer Seite aufgefunden wird. Jeder Prozessor hat für jede Seite einen Zeiger, der auf den wahrscheinlichen Besitzer zeigt (Abbildung 2.1): Bei Programmbeginn ist der Besitzer einer Seite auch gleichzusetzen mit ihrem Manager (a). Wechselt eine Seite aufgrund eines Schreib-Seitenfehlers den Besitzer, so trägt der ursprüngliche Besitzer einen Verweis auf den neuen Besitzer ein (b). Bei einem erneuten Schreib-Seitenfehler in einem anderen Prozessor verlängert sich diese Kette, da nun wiederum der nächste Besitzer im zweiten Prozessor eingetragen wird (c) usw.

Der Eintrag im ersten Prozessor kann aktualisiert werden, wenn er lesend auf die Seite zugreifen will (d): Zunächst wird die Kette durchlaufen und eine Lesekopie vom dritten Prozessor angefordert. Danach wird der Verweis auf den wahrscheinlichen Besitzer aktualisiert.

2.1.2 Das Advanced Shared Virtual Memory System (ASVM) der Intel Paragon

Als Beispiel für eine Softwarelösung eines gemeinsamen virtuellen Speichers wird im folgenden das ASVM-System (*Advanced Shared Virtual Memory Sy-*

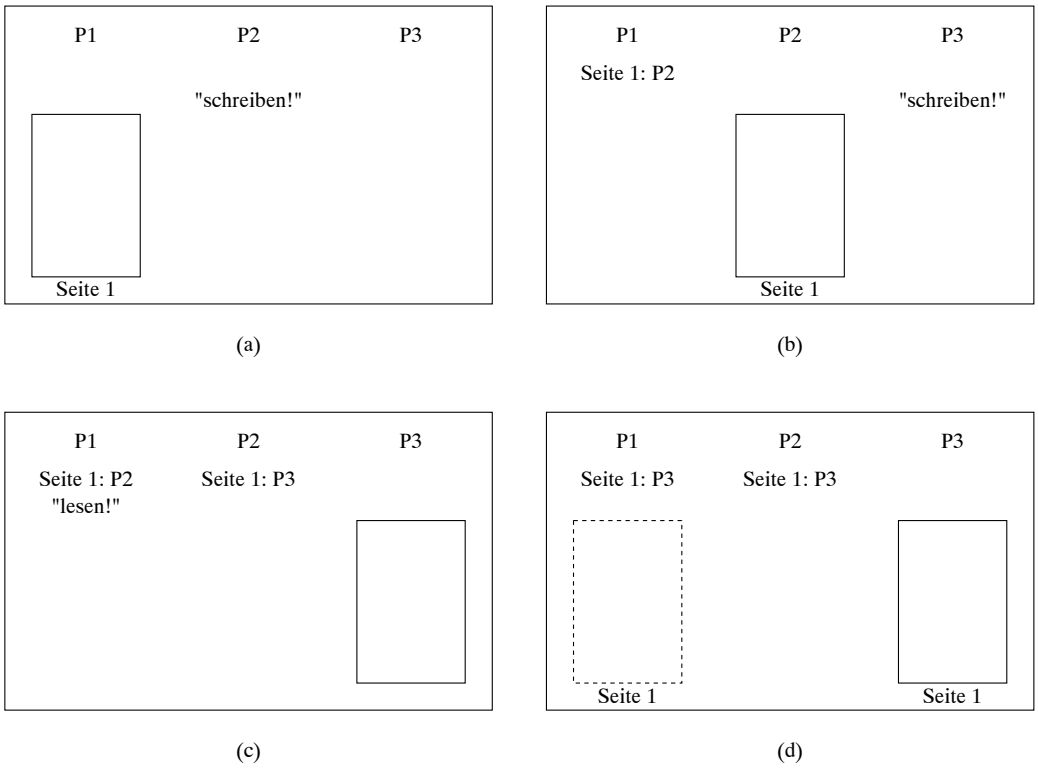


Abbildung 2.1: Wahrscheinlicher Besitzer im Dynamic-Distributed-Manager-Algorithmus

stem, [32]) der Intel Paragon vorgestellt, da die implementierte Laufzeitbibliothek auf diesem Rechner entwickelt wurde.

Das ASVM-System wurde im Rahmen des SVM-Fortran-Projektes im *Intel European Supercomputer Development Center* (EDSC) in München entwickelt. ASVM ist direkt in das Betriebssystem der Intel Paragon integriert und stellt deshalb eine effiziente Implementierung von gemeinsamem virtuellem Speicher dar. Es wird im TeraFlop-Rechner im ASCI(-*Accelerated Strategic Computing Initiative*)-Projekt als Basis für das Betriebssystem eingesetzt werden und die bisherige Mach-Implementierung ersetzen.

ASVM realisiert die im vorigen Abschnitt beschriebene Migration von Seiten bei Seitenfehlern über eine Hierarchie der skizzierten Manageralgorithmen. Zunächst wird versucht mittels des DDM-Algorithmus die gewünschte Seite zu finden. Es werden allerdings nur die letzten Verwaltungsinformationen in einem Cache gehalten. Wird der Besitzer anhand der dort gespeicherten Angaben nicht gefunden, so versucht man mittels FDM- und zuletzt durch den BDM-Algorithmus die gewünschte Seite zu finden. Kann auch

damit die Seite nicht gefunden werden, so ist sie nicht in den Prozessoren vorhanden und muß vom Hintergrundspeicher geladen werden.

Auch eine Schnittstelle zur Leistungsanalyse in Form von Anfragen wird zur Verfügung gestellt, auf der die Werkzeuge der SVM-Fortran-Umgebung aufbauen.

2.2 SVM-Fortran

SVM-Fortran [7, 8] ist eine Programmiersprache, die auf der Intel Paragon die Programmierung von gemeinsamem verteiltem Speicher ermöglicht. Anders als bei der Programmierung durch expliziten Nachrichtenaustausch kann der Programmierer im SVM-Fortran die natürliche, datenparallele Sichtweise beibehalten. Er wird nicht gezwungen, die Sicht eines einzelnen Prozessors einzunehmen. Damit entfällt auch die explizite Ausprogrammierung von Datenzugriffen, so daß man sich auf einer abstrakteren Ebene bewegt.

Erreicht man bezüglich der Beschreibung von Algorithmen mit problemorientierten Programmiersprachen einen höheren Abstraktionsgrad als mit der Assemblersprache, so erreicht im Vergleich dazu SVM-Fortran einen höheren Abstraktionsgrad als *Message-Passing*-Ansätze, bezogen auf die Beschreibung von Kommunikation und den Zugriff auf Daten.

Auch wenn die Plazierung der Variablen im physikalischen Speicher für den Programmierer transparent bleibt, ist weiterhin auf Datenlokalität zu achten. Zur Umgehung der Latenz, die bei Zugriffen auf Daten im Speicher anderer Prozessoren auftritt, muß berücksichtigt werden, daß Prozessoren möglichst wenige solcher entfernter Zugriffe vornehmen müssen. Bei der Intel Paragon muß beispielsweise mit Latenzen von $1,5\,\mathrm{ms}$ gerechnet werden. Die Datenlokalität kann erhöht werden, wenn eine an die Aufgabe angepaßte Arbeitsverteilung gewählt wird, d. h. die Arbeit wird so auf die Prozessoren verteilt, daß die Zugriffe möglichst immer auf die gleichen, dann lokalen Daten erfolgen. Da das Problem der Datenlokalität kein Problem auf Schleifenebene ist, sorgen sogenannte Templates für eine globale Arbeitsverteilung.

Die Entwicklungsumgebung für SVM-Fortran (Abbildung 2.2) besteht aus dem

- SVM-Fortran-Compiler, dem
- Optimierungswerkzeug OPAL *(Optimizer and Locality Analyzer)* und dem
- Applikations-Monitor SAM *(SVM-Fortran Application Monitor).*

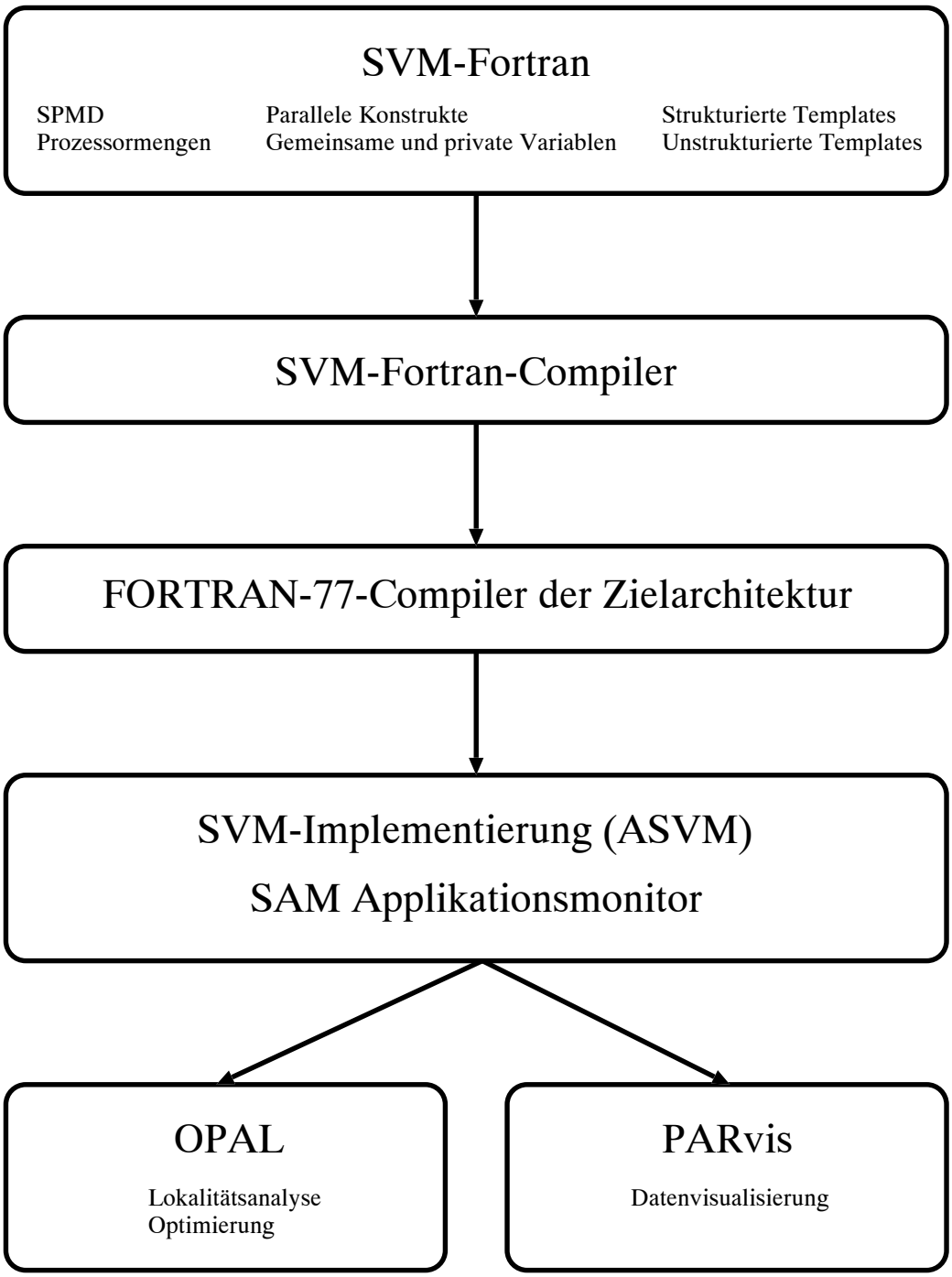


Abbildung 2.2: Integrierte Entwicklungsumgebung für SVM-Fortran

Der Compiler ist ein *Source-to-Source*-Compiler, der ähnlich einem Präprozessor aus dem SVM-Fortran-Quelltext ein FORTRAN-77-Programm erzeugt. Dieses kann dann mit dem FORTRAN-77-Compiler der Zielarchitektur übersetzt und mit der Laufzeitbibliothek gebunden werden, so daß ein Programm entsteht, welches auf der entsprechenden SVM-Implementierung lauffähig ist.

Die Programmoptimierung wird durch das Optimierungswerkzeug *OPAL* [24] unterstützt. Damit können Seitenfehler aufgedeckt und verfolgt werden (Kapitel 2.3). Auch das Visualisierungswerkzeug PARvis [2, 6] kann bei der Entwicklung eingesetzt werden. Beide Werkzeuge setzen auf *SAM* [42] auf.

Man beachte, daß die Beispiele in den folgenden Unterkapiteln nicht in sich geschlossen sind. So wurde beispielsweise auf die Deklaration von Variablen verzichtet, wenn der Datentyp für den erläuterten Sachverhalt unwesentlich ist.

2.2.1 Programmiermodell

Die Ausführung von SVM-Fortran-Programmen entspricht dem *Single-Program-Multiple-Data*-Ansatz (SPMD-Ansatz, [12]), der um funktionalen Parallelismus erweitert wird.

Ein SPMD-Programm wird von einer Anzahl von abstrakten Prozessoren[1] ausgeführt, die dasselbe Programm bearbeiten. Durch den funktionalen Parallelismus in SVM-Fortran ist es möglich, grobgranulare parallele Aufträge (Programmteile) zu spezifizieren, die selber wieder datenparallelen Code enthalten, welcher entsprechend dem SPMD-Modell ausgeführt wird.

Der Vorteil dieses Ansatzes ist, daß das serielle Programm in seiner Struktur erhalten bleibt. Zusätzliche Direktiven, die die parallele Ausführung steuern, werden eingefügt. Die Parallelisierung kann somit auch schrittweise oder nur für einen Teil von Regionen durchgeführt werden.

Charakterisierend für den SPMD-Ansatz ist, daß alle Prozessoren zu Beginn der Ausführung herangezogen werden und bis zur Beendigung des Programms belegt bleiben. Die Menge der Prozessoren bleibt also statisch. Somit ist es leichter für eine gleichmäßige Arbeitsverteilung zu sorgen, als in einem dynamischen *fork-/join*-Konzept, in dem Prozesse während des Programmlaufes abhängig von Programmzuständen erzeugt werden und sich wieder mit dem erzeugenden Prozeß vereinigen können.

[1] In [12] spricht man in diesem Zusammenhang von *Prozessen*. Durch die Wortwahl soll hier aber ausgedrückt werden, daß von SVM-Fortran eine bijektive Zuordnung von abstrakten zu physikalischen Prozessoren angenommen wird.

CSVM$ **PROCESSORS**:: $P1(NUMPROC() - 4)$
CSVM$ **PROCESSORS**:: $P2(4)$

Abbildung 2.3: Prozessormengen

Neben parallelen Schleifen enthält ein Programm üblicherweise auch serielle und replizierte Regionen. Serielle Regionen werden nur von einem Prozessor ausgeführt, während die restlichen Prozessoren pausieren. Zu Beginn der nächsten parallelen Anweisung nehmen sie wieder an der Programmabarbeitung teil. In einer replizierten Region wird der serielle Code von allen Prozessoren redundant durchlaufen. Replizierte Regionen sind z. B. bei Initialisierungen sinnvoll, die von allen Prozessoren ausgeführt werden müssen.

2.2.2 Prozessormengen

Zu Beginn wird das gesamte Programm als ein *Auftrag* (task) angesehen. Durch parallele Anweisungen wird es in *Teilaufträge* (subtasks) aufgeteilt. Teilaufträge sind:

- Abschnitte einer parallelen Sektion *(parallel section)* und
- Blöcke von Iterationen paralleler Schleifen.

Jeder Auftrag wird einer Menge von Prozessoren zugeteilt. Die Prozessoren, die an der Ausführung eines Teilauftrages beteiligt sind, werden *aktive Prozessormenge (active processor set)* genannt. Zu Beginn der Programmausführung besteht die aktive Prozessormenge aus allen der Anwendung zugewiesenen Prozessoren. Sind mehr als ein Prozessor an der Ausführung eines Auftrages beteiligt, so kann wiederum eine erneute Aufteilung in weitere Teilaufträge erfolgen.

Durch die PROCESSORS-Direktive werden Prozessoranordnungen definiert, die zur Arbeitsverteilung eingesetzt werden (Abbildung 2.3). Bei der Deklaration kann die Funktion NUMPROC() für die Anzahl der Prozessoren verwendet werden, die dem Programm zugewiesen wurden *(„number of processors“)*.

Im Beispiel definiert $P1$ eine eindimensionale Prozessoranordnung. Da für Prozessormenge $P2$ vier Prozessoren reserviert werden, sollen $P1$ die restlichen der Anwendung zur Verfügung stehenden Prozessoren zugeordnet werden. Damit für $P1$ ein zulässiger Wert errechnet werden kann, muß das Programm mit mindestens 5 Prozessoren gestartet werden.

```
CSVM$   PRIVATE:: I, SKALAR
CSVM$   SHARED:: FELD

C       einfache parallele Schleife
CSVM$   PDO(LOOPS(I), PROCESSORS(P1), STRATEGY(BLOCK))
        DO I = 1,100
              FELD(I) = FELD(I) + 1
        END DO

C       Reduktion
CSVM$   PDO(LOOPS(I), PROCESSORS(P1),
       >REDUCTION(SKALAR))
        DO I = 1,100
              SKALAR = SKALAR + FELD(I)
        END DO
```

Abbildung 2.4: Parallele Schleifen und Reduktionen

2.2.3 Parallele Anweisungen

SVM-Fortran kennt zwei parallele Anweisungen:

- parallele Schleifen und
- parallele Sektionen.

Abbildung 2.4 zeigt im oberen Teil eine parallele Schleife. Die DO-Schleife wird auf die Prozessoren der Prozessormenge $P1$ verteilt. Dafür wird die Strategie BLOCK verwendet, d. h. jeder der Prozessoren bekommt einen Block von etwa $\frac{100}{|P1|}$ aufeinanderfolgenden Iterationen zugeteilt.

Parallele Schleifen werden einer bestimmten Strategie folgend auf die aktive Prozessormenge verteilt. Die Strategien werden in *statisches, dynamisches, vordefiniertes* (predefined) und *halbdynamisches Scheduling* unterschieden.

Statisches Scheduling sind die Strategien *BLOCK, CYCLIC* und *ALIGNED.* Bei der Blockverteilung werden die Iterationen in zusammenhängenden, gleichgroßen Blöcken auf die Prozessoren der Prozessormenge verteilt. Bei der zyklischen Verteilung werden die Iterationen reihum in Blöcken angegebener Größe verteilt. Die *ALIGNED*-Strategie sorgt dafür, daß die Iterationen den Prozessoren entsprechend der referenzierten Daten – beispielsweise bezogen auf die Seitenverteilung eines Feldes im gemeinsamen Speicher – zugeordnet werden.

```
CSVM$   PSECTION
CSVM$   SECTION ON P1
                ...
CSVM$   SECTION ON P2
                ...
CSVM$   PSECTION_END
```

Abbildung 2.5: Parallele Sektion

Bei *dynamischem Scheduling* werden Prozessoren neue Arbeitseinheiten zugeteilt, sobald vorher zugewiesene abgearbeitet wurden. Dynamisches Scheduling sorgt für eine Optimierung der Lastbalance, vernachlässigt jedoch die Datenlokalität. Die anderen Strategien beziehen sich auf das Scheduling von Template-Elementen. Deshalb wird das vordefinierte Scheduling erst in Kapitel 2.2.5 beschrieben.[2]

In der zweiten parallelen Schleife in Abbildung 2.4 wird eine *Reduktion* ausgeführt: Die Elemente eines Feldes werden in einem Skalar aufaddiert. Neben der hier verwendeten Addition kann als Reduktionsoperation auch eine Subtraktion oder eine Multiplikation dienen. Weitere Reduktionsoperationen weisen einer skalaren Variable das Minimum bzw. Maximum von Feldelementen zu. Intern werden die Feldelemente in den beteiligten Prozessoren in privaten Kopien der Variable $SKALAR$ summiert. Die Kopien werden anschließend durch die *gdsum*-Operation zu Teilergebnissen aufsummiert und in die privaten Kopien aller Prozessoren geschrieben.

Auch die zweite Schleife des Beispiels wird blockverteilt, da diese Strategie die Voreinstellung ist, die verwendet wird, wenn keine spezielle Strategie vorgegeben wurde.

Durch die Einführung von *parallelen Sektionen* (Abbildung 2.5) wird dem Programmierer die Möglichkeit gegeben, funktionalen Parallelismus in SVM-Fortran-Programme einfließen zu lassen. Die einzelnen Sektionen können datenparallelen Code enthalten. Eine Zuordnung der Abschnitte einer parallelen Sektion auf Prozessormengen ist durch entsprechende Notation möglich. Im Beispiel wird die erste Sektion Prozessormenge $P1$ und die zweite $P2$ zugewiesen.

Vor und nach jeder parallelen Anweisung erfolgt eine globale Synchronisation aller Prozessoren der aktiven Prozessormenge, sofern sie im Programm nicht durch die Option NOBARRIER explizit unterdrückt wird.

[2]Dynamisches und halbdynamisches Scheduling soll hier nicht weiter ausgeführt werden, da diese Strategien im Rahmen der vorliegenden Arbeit nicht verwendet werden. Es sei aber auf [7] und speziell für das dynamische Scheduling auf [39] verwiesen.

2.2.4 Gemeinsame und private Variablen

SVM-Fortran kennt gemeinsame (shared) und private Variablen bzw. COMMON-Blöcke. Ist ein Datum privat, so existiert eine Kopie unabhängig in jedem Prozessor. Die privaten Variablen werden durch den Kontrollfluß des jeweiligen Prozessors verändert, haben also im allgemeinen verschiedene Werte. Grundsätzlich werden alle Variablen eines Programms als SHARED angenommen. Bei Variablen in COMMON-Blöcken ist darauf zu achten, daß diese in allen Programmeinheiten gleich deklariert werden (gemeinsam oder privat).

Bereits in Abbildung 2.4 tauchen gemeinsame und private Variablen auf: Jeder Prozessor hat seinen eigenen, privaten Laufindex I für die DO-Schleifen. Die Variable $FELD$ ist im gemeinsamen Speicher abgelegt, d. h. jeder Prozessor arbeitet auf dem gleichen Speicherbereich im gemeinsamen Speicher, wenn dieses Feld referenziert wird.

2.2.5 Reguläre Templates

Da die Optimierung der Datenlokalität keine Aufgabe ist, die auf Schleifenebene, sondern auf globaler Ebene zu lösen ist, existieren in SVM-Fortran Templates. *Templates* sind rechteckige Indexbereiche, die auf Prozessoren verteilt werden.

Wie bereits weiter oben erwähnt, werden Templates unter anderem zur Realisierung von vordefiniertem Scheduling verwendet. Bei dem *vordefinierten Scheduling* wird die Verteilung von Schleifeniterationen gemäß der Verteilung der entsprechenden Template-Elemente vorgenommen (Abbildung 2.6).

Zu den Strategien zur Verteilung gehören Blockverteilung, zyklische Verteilung und die generalisierte Blockverteilung, bei der sich die Größe der Blöcke festlegen läßt. Insbesondere wird dadurch die Berücksichtigung von Seitengrenzen erleichtert; bei der Blockverteilung ist dies nicht möglich. Auch die Ausrichtung der Verteilung an einem Datenfeld oder einem anderen Template ist möglich.

In Abbildung 2.6 werden die drei Templates $T1, T2$ und $T3$ definiert und verteilt. $T1$ bietet Platz für 5000 Elemente und wird blockverteilt auf die Prozessoren der Prozessormenge $P1$. Die Prozessormengen wurden bereits in Abbildung 2.3 definiert.

Mit $T2$ wird eine indirekte Verteilung demonstriert: Die Elemente des Template $T2$ werden entsprechend den Elementen des Vektors MAP auf $P2$ verteilt, d. h. das $I-$te Element des Template $T2$ gelangt auf den Prozessor, dessen Index in $MAP(I)$ gespeichert ist.

Als drittes Beispiel ist eine generalisierte Blockverteilung angeführt.

```
CSVM$   TEMPLATE:: T1(5000)
CSVM$   TEMPLATE:: T2(10000)
CSVM$   TEMPLATE:: T3(15000)

C       Blockverteilung
CSVM$   DISTRIBUTE(BLOCK) ONTO P1:: T1

C       Indirekte Verteilung
        READ(*,*) (MAP(I), I = 1, 10000
CSVM$   REDISTRIBUTE(I) ONTO P2(MAP(I)):: T2

C       generalisierte Blockverteilung
        WORK(1) = 100
        WORK(2) = 1000
        WORK(3) = 10000
CSVM$   REDISTRIBUTE(GENERAL_BLOCK(WORK))
     >      ONTO P2:: T3
```

Abbildung 2.6: Reguläre Templates

Hier werden die 15000 Elemente des Template $T3$ entsprechend dem Feld $WORK$ auf die vier Prozessoren von $P2$ verteilt: $WORK(1), WORK(2)$ und $WORK(3)$ geben an, wieviele Elemente des Template den ersten drei Prozessoren der Prozessormenge zugeteilt werden. Die restlichen Elemente – hier sind es 3900 – gelangen auf Prozessor vier.

Templates machen die Verteilung von Iterationen einfacher und flexibler: Eine Änderung der Verteilung des Template reicht aus, um die Arbeitsverteilung aller parallelen Schleifen, die sich auf dieses Template beziehen, zu ändern. Außerdem müssen die Berechnungen zur Verteilung nur einmal beim Anlegen des Template und nicht erneut vor jeder Schleife ausgeführt werden.

Templates können dynamisch erzeugt, verteilt und während des Programmlaufes auch umverteilt werden. Weiterhin können sie auch als Argumente an Unterprogramme übergeben werden. Somit kann man eine Verteilung im Hauptprogramm definieren und Schleifen im Unterprogramm entsprechend verteilen.

2.2.6 Unstrukturierte Templates

Anwendungen, die auf unstrukturierten Gittern arbeiten, können mit den zuvor aufgeführten Sprachmitteln nicht adäquat modelliert werden. Das Git-

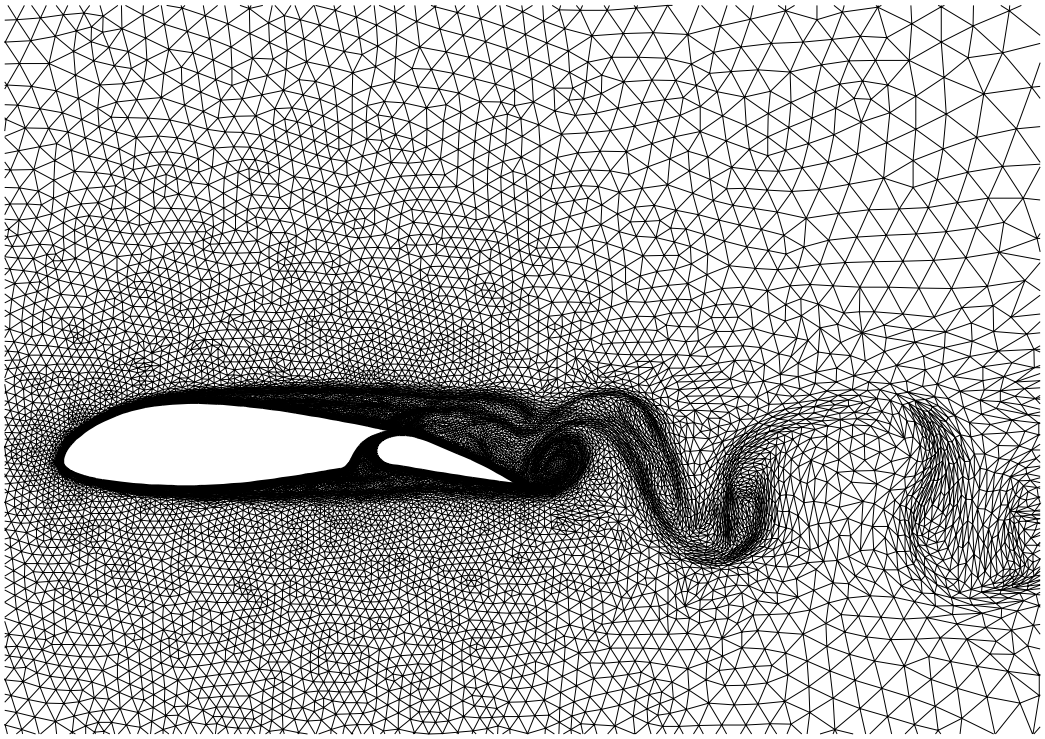


Abbildung 2.7: Unstrukturiertes Gitter: umströmte Tragfläche (Querschnitt) in einer Strömungssimulation

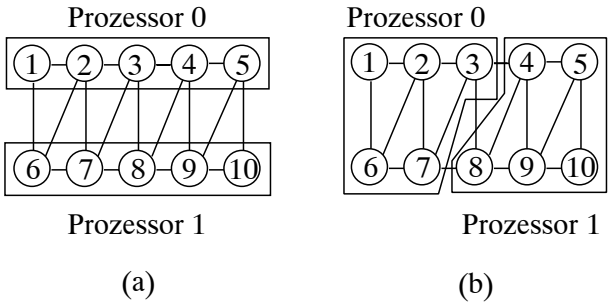


Abbildung 2.8: Numerierung eines unstrukturierten Gitters: (a) Zerlegung gemäß der Knotennumerierung; (b) Zerlegung gemäß der Nachbarschaftsbeziehungen

ter ist ungleichmäßig dicht. In Abbildung 2.7 [26] sieht man, daß die Dichte des Gitters von der Tragfläche nach außen hin abnimmt; hinzu kommt noch, daß dies nicht gleichmäßig geschieht. Um beispielsweise die Verwirbelungen hinter der Tragfläche studieren zu können, ist in diesem Bereich das Gitter ebenfalls recht dicht gehalten. Dieses Gitter mit einer Blockverteilung oder einer anderen regelmäßigen Verteilung zu zerlegen, wird der Aufgabe nicht gerecht. Die Gitterpunkte unstrukturierter Gitter sind üblicherweise durchnumeriert (Abbildung 2.8). Eine Blockzerlegung der Numerierung berücksichtigt die Nachbarschaftsbeziehungen nicht (a). Berücksichtigt man bei der Zerlegung diese Beziehungen, so erhält man die Zerlegung aus (b). Sie zeichnet sich durch eine geringere Anzahl von Verbindungen zwischen den beiden Teilgittern aus. Dies bedeutet, daß weniger Kommunikation stattfindet.

Wie im vorigen Abschnitt gezeigt wurde, kann mittels indirekter Verteilung über einen Indirektionsvektor eine gewünschte Verteilung erreicht werden. Die Einträge in diesem Vektor könnten beispielsweise die Ergebnisse des Laufes eines externen Partitionierers widerspiegeln. Die im folgenden beschriebenen unstrukturierten Templates sollen eine Möglichkeit bieten, diesen Partitionierungsschritt im SVM-Fortran-Programm auszuführen. Somit ist es für den Anwendungsprogrammierer leichter, mit verschiedenen Partitionierern zu experimentieren und die beste Verteilung für die jeweilige Anwendung zu finden.[3]

Unstrukturierte Templates übernehmen deshalb zusätzlich zur Aufgabe die Datenverteilung zu steuern die Rolle eines Speichers für Geometrie- und/oder Verbindungsinformationen, so daß die gesamte Information zur Partitionierung der Gitter den entsprechenden Algorithmen zur Verfügung steht. Zusätzlich können auch Informationen über den Berechnungsaufwand einzelner Punkte bzw. das Zugriffsverhalten als Attribute der Template-Elemente gespeichert werden.

Da auch Verteilungen für adaptive Gitter, also sich während der Laufzeit verändernde Gitter, über unstrukturierte Templates bestimmt werden sollen, ist eine dynamische Anpassung der Templates zu ermöglichen.

Somit sind neben Direktiven für die Definition unstrukturierter Templates auch solche für Einfügen und Löschen von Gitterknoten und -kanten, für Schreiben und Lesen von Knoten- und Verbindungsattributen und für die Anwendung von Partitionierern auf das unstrukturierte Template vorhanden.

In Abbildung 2.9 werden im Template T Informationen gespeichert, die für den Partitionierer *RCB* (Recursive Coordinate Bisection, Kapitel 4.1.1), ein koordinatenbasiertes Verfahren, relevant sind.

[3]Ein weiterer Vorteil ist, daß sich alle in SVM-Fortran integrierten Partitionierer der gleichen Datenschnittstelle bedienen; dies ist naturgemäß bei der Vielzahl der verfügbaren externen Partitionierungswerkzeuge nicht der Fall.

```
        REAL x, y, z
        REAL Koordinaten(3, 10000)
CSVM$   TEMPLATE:: T{(10000), ATTRIBUTES(x, y, z)}

C       Partitionierungsphase
CSVM$   INSERT:: T(1 : 700)
        DO i = 1, 700
CSVM$       SET:: T(i).x TO Koordinaten(1, i)
CSVM$       SET:: T(i).y TO Koordinaten(2, i)
CSVM$       SET:: T(i).z TO Koordinaten(3, i)
        END DO
CSVM$   REDISTRIBUTE RCB(T, x, y, z) ONTO P

C       Berechnungsphase
        ...

C       Adaptiver Schritt
CSVM$   INSERT:: T(701 : 1000)
        DO i = 701, 1000
CSVM$       SET:: T(i).x TO Koordinaten(1, i)
CSVM$       SET:: T(i).y TO Koordinaten(2, i)
CSVM$       SET:: T(i).z TO Koordinaten(3, i)
        END DO
CSVM$   REDISTRIBUTE RCB(T, x, y, z) ONTO P
```

Abbildung 2.9: Verwendung von unstrukturierten Templates

Durch die Optionen der TEMPLATE-Anweisung – die geschweiften Klammern deuten den Mengencharakter unstrukturierter Templates an – wird Speicherplatz für maximal 10000 Gitterpunkte angelegt. Zusätzlich werden drei Attribute definiert, die die Raumkoordinaten repräsentieren. Attribute können die FORTRAN-Typen INTEGER, REAL oder DOUBLE PRECISION haben. Diese Deklaration ermöglicht es, die definierten Attributnamen in SET- und GET-Direktiven zu verwenden.

In der Partitionierungsphase werden die Knoten 1 bis 700 eingefügt (INSERT) und mit Attributen versehen (SET). Man sieht, daß eine Syntax verwendet wird, wie sie in der Programmiersprache C für Strukturen üblich ist. In unserem Beispiel werden den Attributen Koordinaten zugewiesen, die in einem Feld gespeichert sind; sie können beispielsweise beim Programmstart aus einer Datei eingelesen werden. Nachdem alle Informationen in Form von Attributen zur Verfügung stehen, werden die Knoten mittels eines Partitionierungsalgorithmus auf die Prozessoren verteilt. Auf diese Partitionierungsphase folgt eine Berechnungsphase. Um den Einsatz adaptiver Gitter zu illustrieren, ist eine anschließende Anpassung des Gitters an neue Gegebenheiten angedeutet. Für die Anwendung in Abbildung 2.7 würde dies z. B. bedeuten, daß bei der Berechnung der Umströmung der Tragfläche Gebiete ausgemacht werden, in denen eine Verfeinerung des Gitters notwendig wird. Deshalb werden nun neue Knoten (701 bis 1000) hinzugefügt.

Auch hier wird im Anschluß an das Sammeln von Informationen durch die SET-Direktive eine Neuverteilung der Knoten auf die Prozessoren ausgeführt. In der Praxis wird man allerdings bemüht sein, die Verteilung der bisherigen Knoten nicht aufzugeben, sondern neue Knoten in die bestehende Verteilung einzuarbeiten. Optimierungsalgorithmen, die diese Gegebenheiten berücksichtigen, sollten hier eingesetzt werden.

Entsprechend können Verbindungen zwischen Knoten definiert und Partitionierer eingesetzt werden, die diese Verbindungsinformationen verwenden. Auch Partitionierer, die Gewichte (beispielsweise Rechenlast einzelner Gitterknoten) berücksichtigen, können verwendet werden, da durch die Attribute alle Möglichkeiten offengehalten sind. Partitionierungsalgorithmen werden ausführlich in Kapitel 4 dargestellt.

Eine weitere Möglichkeit, die durch unstrukturierte Templates ermöglicht werden muß, ist die Verknüpfung zwischen Finiten Elementen und den sie umgebenden Knoten. Wie man Informationen über Verbindungen zwischen Gitterknoten oder zwischen Finiten Elementen in den Attributen speichern kann, so kann man auch Informationen über Verknüpfungen speichern. So können die Gitterknoten, die ein Element umgeben als Nachbarn aufgefaßt werden, die durch eine Verbindung mit dem Element verknüpft sind. Damit können auch Partitionierer eingesetzt werden, die auf diese Beziehungen Rücksicht nehmen.

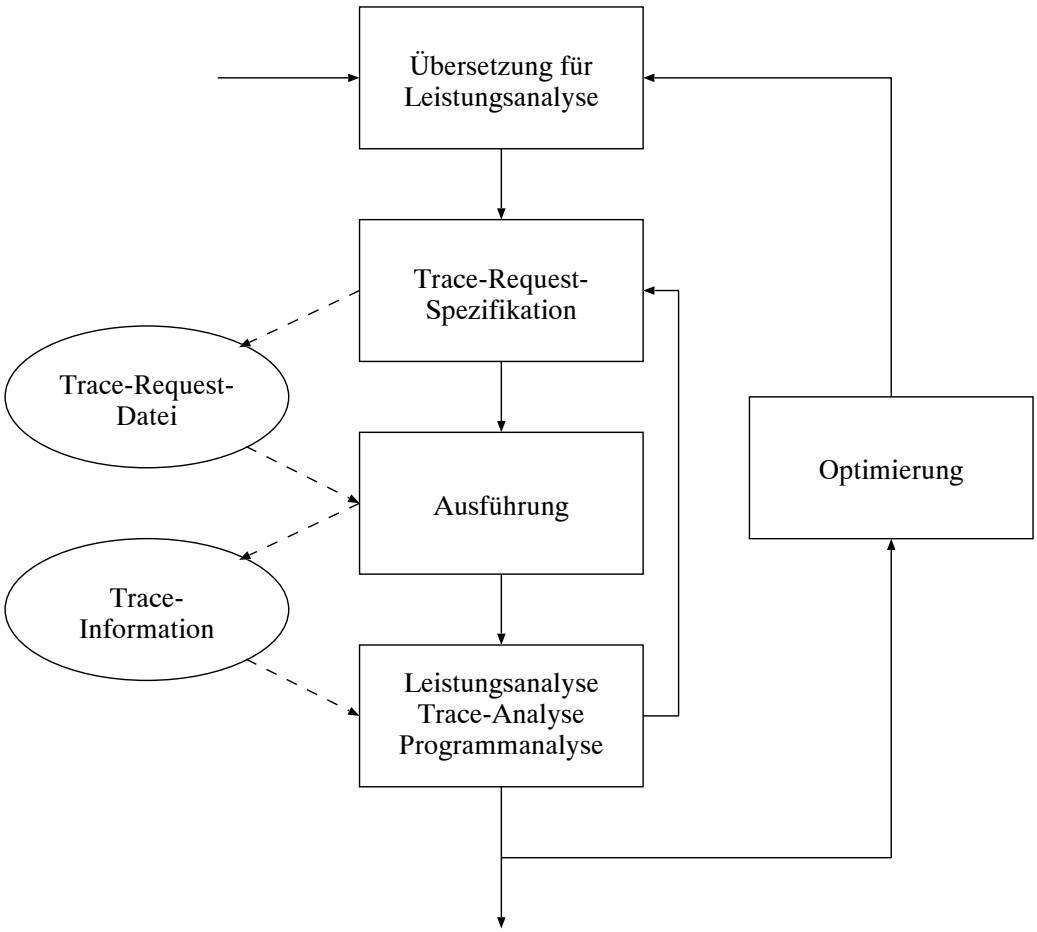


Abbildung 2.10: Zyklus zum Optimieren eines SVM-Fortran-Programmes

Es sei an dieser Stelle angemerkt, daß im Rahmen dieser Arbeit Funktionen zur Verfügung gestellt werden, die über Unterprogrammaufrufe die skizzierte Funktionalität im Quellprogramm zur Verfügung stellen. Die endgültige Integration in den Compiler, d. h. die Einbindung mittels SVM-Fortran-Direktiven, ist nicht Teil dieser Arbeit.

2.3 Werkzeuge zur Leistungsanalyse

Um dem Programmierer Hilfe bei Optimierungen in Hinblick auf die oben erwähnte Datenlokalität zu geben, wurde das Werkzeug OPAL *(Optimizer and Locality Analyzer)* entwickelt [24]. Um Daten über den Lokalitätsgrad einer Anwendung zu bekommen, wird das Programm instrumentiert, d. h.

LOCAL foo.PDO_LABEL(150): RPF A, WPF A

Abbildung 2.11: Beispiel für eine Anfrage an das Monitoring

insbesondere (Schreib- und Lese-)Seitenfehler, als auch Synchronisationszeiten werden protokolliert. Dabei wird ein inkrementeller Analyseansatz verfolgt (Abbildung 2.10). Zu Beginn wird nur ein grober Überblick über die Ausführungszeiten und Seitenfehler der einzelnen Regionen angefordert. Für kritische Teile des Programms, also Regionen, in denen Seitenfehler oder Synchronisationszeiten gehäuft auftreten, wird ein weiterer Lauf gestartet, in dem detailliertere Informationen ermittelt werden, beispielsweise auch die Namen der Variablen, die einen Seitenfehler verursachen.

Man sieht, daß der Optimierungszyklus damit beginnt, daß man das Programm für die Leistungsanalyse übersetzt. Der Compiler erzeugt eine spezielle Version des ausführbaren Codes. In dieser Version wird das Laufzeitverhalten protokolliert.

Anschließend werden in OPAL die gewünschten Daten spezifiziert. Man erhält eine editierbare Textdatei *(trace request file)*, in der die Anforderungen an das Monitoring formuliert sind. Während der Ausführung des Programms werden die ermittelten Daten pro Prozessor in eine Datei geschrieben *(trace data file)*, die mit Hilfe von OPAL ausgewertet werden kann. Weitere Läufe mit detaillierteren Anforderungen werden anschließend initiiert (innere Schleife). Eine Optimierung des Quellprogrammes führt zur erneuten Übersetzung des Programms (äußere Schleife).

Bei jedem Lauf werden jeweils nur die Daten ermittelt, die wirklich von Interesse sind. Die Erhebung der detaillierten Daten beschränkt sich lediglich auf bestimmte Regionen im Programm. Somit wird also das Datenaufkommen möglichst klein gehalten. Auch die Einflüsse auf die Programmlaufzeiten und damit die Verfälschung der Tests wird so eingeschränkt.

Die Monitoring-Informationen werden von *SAM* [42] festgehalten und in die Trace-Datei geschrieben. Er liefert also die Datenbasis für die Werkzeuge OPAL und PARvis. Der Monitor unterstützt das Konzept der inkrementellen Leistungsanalyse, indem Informationen unterschiedlicher Granularität bereitgestellt werden.

OPAL liefert bei der Parallelisierung wertvolle Hinweise auf Leistungsengpässe im Programm. Ohne dieses Werkzeug wäre deren Lokalisierung erheblich schwieriger, da man nur indirekt durch Überlegen auf mögliche Ursachen schließen kann und oftmals auch falsche Schlußfolgerungen zieht.

Auch die Instrumentierung und die Interpretation der generierten Daten müßte von Hand erfolgen. Abbildung 2.11 zeigt eine exemplarische Anforde-

rung an das Monitoring. Für die PDO-Schleife 150 im Unterprogramm foo sollen die Seitenfehler für das Feld A protokolliert werden. Man kann sich vorstellen, daß die Instrumentierung von Hand aufwendig und auch fehleranfällig ist. OPAL unterstützt hier die Arbeit, da der Programmierer die gewünschten Stellen im Programm im Quelltext aufsuchen und dort die gewünschten Daten spezifizieren kann.

Die von SAM erzeugten Daten ohne jegliches Hilfsmittel auszuwerten scheitert bereits an der Menge der Daten. Sehr schnell sind mehrere 100 kBytes pro Prozessor angefallen, die Zeile für Zeile untersucht werden müßten. OPAL liefert hier Übersichten und Summenwerte, so daß man sich zu den kritischen Stellen des Programms vorarbeiten kann.

Kapitel 3

Parallelisierung des AVL-FIRE-Benchmarks

Der AVL-FIRE-Benchmark [10] ist ein Fragment aus einem CFD-Programmpaket der Grazer Firma AVL. Es erlaubt strömungsmechanische Vorgänge in komplexen, unregelmäßigen Geometrien (Abbildung 3.1) mit Hilfe eines Finite-Volumen-Ansatzes zu simulieren. Hierbei wird der Versuchsraum in eine feste Menge von Kontrollvolumina eingeteilt. Bei dem Programmausschnitt handelt es sich um den iterativen Löser der linearen Gleichungssysteme[1]. Die Hauptschleife umfaßt etwa 150 Zeilen, ist also recht überschaubar.

Die Matrix, die das lineare Gleichungssystem beschreibt, ist dünn besetzt. Sie tritt nicht explizit in dem Fragment auf, sondern wird implizit in der Berechnung des Gleichungslösers berücksichtigt. Die Daten für die einzelnen Knoten werden in eindimensionalen Feldern abgespeichert, und entsprechend der Matrix wird über indirekte Adressierung darauf zugegriffen. Die Nichtnulleinträge der Matrix stellen die Verbindungsstruktur des verwendeten Gitters dar.

Die Daten, die die Gitter beschreiben, liegen in Dateien vor. Dort sind die Größe des Gitters, sowie die Koeffizienten und die Nummern der sechs Nachbarn eines jeden Knotens abgespeichert. Drei Datensätze wurden für die Messungen verwendet: Der kleine Datensatz *tjunc* (13845 Knoten), der mittlere *drall* (47312 Knoten) und *pent,* mit 108000 Knoten der größte der verwendeten Datensätze.

Die Messungen wurden auf einer Intel Paragon XP/S mit 20 Prozessoren durchgeführt. 19 dieser Knoten haben jeweils 16 MByte lokalen Speicher. Ein Knoten ist mit 32 MByte ausgestattet. Um Verfälschungen der Messungen durch Paging aufgrund von Speicherknappheit zu vermeiden, wurde bei den Läufen auf nur einem Prozessor dieser 32-MByte-Knoten verwendet.

[1] Krylov-Unterraum-Verfahren ORTHOMIN [54]

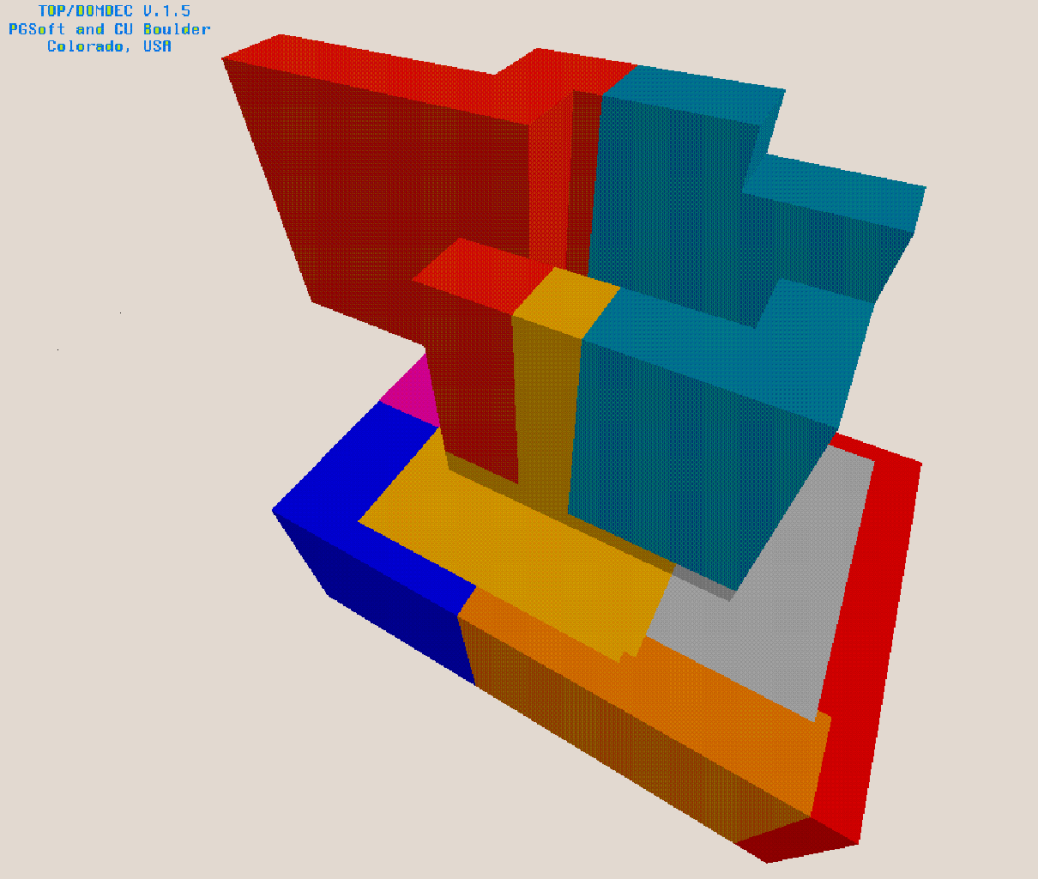


Abbildung 3.1: Beispiel für ein unstrukturiertes Gebiet (drall) im AVL-FIRE-Benchmark

Für die Optimierungsschritte sind Leistungsdaten angegeben, anhand derer man die Fortrschritte der einzelnen Stufen verfolgen kann. Auch für die vollständig optimierte Version sind Meßwerte für alle betrachteten Datensätze und verschiedene Prozessorzahlen angefügt.

3.1 Ausgangspunkt: Die serielle Version

Der Benchmark GCCG besteht im wesentlichen aus den drei Programmteilen:

- Einlesen der Gitterdaten
- Gleichungen des Systems iterativ lösen (Hauptschleife)

```
          RESREF = ...
          ...
100       CONTINUE
          ...
C         Berechnungsschritt
          DO NC = NINTCI, NINTCF
               DIREC2(NC) = BP(NC) * DIREC1(NC)
     >                     -BS(NC) * DIREC1(LCC(1,NC))
     >                     -BW(NC) * DIREC1(LCC(4,NC))
                           ...
     >                     -BH(NC) * DIREC1(LCC(6,NC))
          END DO
C---
          ...
          RESNEW = ...
C---
          RATIO = RESNEW/RESREF
          IF (RATIO .LE. ε) GOTO 9999
C---
          ...

          IF (ITER .LT. 10000) GOTO 100
C---
9999      CONTINUE
```

Abbildung 3.2: Iterativer Gleichungslöser

- Ausgabe der Rechenergebnisse und der Leistungsdaten (Laufzeit und Rechenleistung der Hauptschleife)

Eine Parallelisierung von Eingabe und Ausgabe ist wegen des *Shared-virtual-Memory*-Ansatzes nicht notwendig. Daten werden in den globalen Adreßraum geschrieben oder aus diesem gelesen. Die Verteilung auf die physikalischen Speicher wird von der SVM-Implementierung übernommen und bleibt auf der Programmebene verborgen. Im Rahmen dieser Arbeit wird der rechenintensive Teil parallelisiert. Dies ist die Hauptschleife zur Lösung des Gleichungssystems (Abbildung 3.2).

Das zweidimensionale Indirektionsfeld $LCC(Richtung, Knotennummer)$ enthält die Nummern der sechs Nachbarknoten, wobei die Ziffern 1 bis 4 für die Himmelsrichtungen *Süden, Osten, Norden* und *Westen,* 5 für *„oberer*

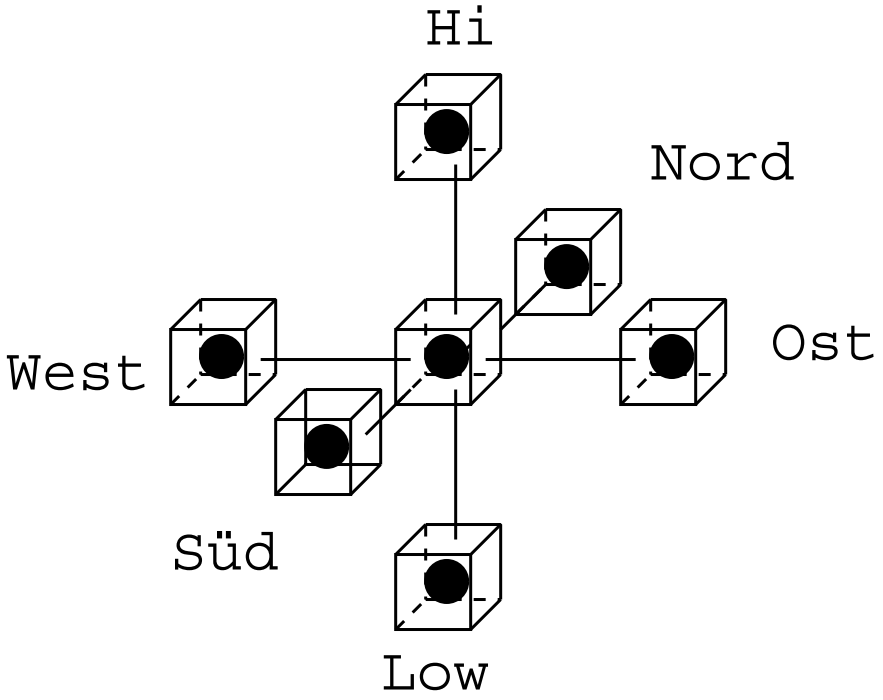


Abbildung 3.3: Bezeichnungen der Nachbarknoten

Nachbar“ und 6 für *„unterer Nachbar“* stehen (Abbildung 3.3). Ein Knoten repräsentiert also ein Finites Volumen in Würfelform. Die Knoten des Gebietes („interne Zellen“) sind durch eine weitere Schicht von Knoten umgeben, die wegen zu erfüllender Randbedingungen hinzugefügt werden („externe Zellen“). Die Berechnung läuft dabei lediglich über die internen Zellen; die interen Zellen, die am Rande des Gebietes liegen, referenzieren allerdings auch die externen Zellen. Bei der Numerierung der Knoten werden zunächst die inneren Zellen durchnumeriert; die externen Zellen schließen sich daran an.

In jedem Durchlauf der Hauptschleife werden die Werte für die Knoten ($DIREC2(NC)$) unter gewichteter Berücksichtigung der Nachbarn ermittelt. Anschließend wird die Norm des neuen Residuums $RESNEW$ berechnet und mit einer Referenznorm $RESREF$ verglichen. Bei genügend hoher Genauigkeit wird die Berechnung abgebrochen. Ansonsten wird das Verfahren fortgesetzt und spätestens nach der 10000sten Iteration beendet.

Die Die Leistungskenndaten der ursprünglichen seriellen Version sind in Tabelle 3.1 für die drei betrachteten Datensätze *tjunc*, *drall* und *pent* aufgeführt. Die Rechenleistungen bewegen sich zwischen $6,84$ MFlops und $7,19$ MFlops, weichen also erwartungsgemäß kaum voneinander ab.

Datensatz	Laufzeit [s]	Rechenleistung [MFlops]
tjunc	20,16	7,19
drall	71,65	6,85
pent	267,31	6,84

Tabelle 3.1: Rechenleistung der sequentiellen Version des AVL-FIRE-Benchmarks

Diese serielle Version wurde zunächst mit dem SVM-Fortran-Compiler übersetzt, ohne daß Änderungen im Code eingefügt wurden. Sie benötigte beispielsweise für die Berechnung des *drall*-Datensatzes nun 135,5 Sekunden (3,6 MFlops).

Da Verschlechterungen in diesem Maße nicht zu erwarten sind, wurde der erzeugte Code genauer untersucht. Es stellte sich heraus, daß im Hauptprogramm einige Felder definiert sind, die den Prozessoren gemeinsam sind. Diese sind in SVM-Fortran als zeigerbasierte Felder implementiert, d. h. der Zugriff auf die Feldelemente erfolgt indirekt. Dadurch ergibt sich, daß sich die Ausführungszeit verdoppelt.

Dies ist nicht der Fall, wenn Felder als Argumente an ein Unterprogramm übergeben werden. Diese werden nicht zeigerbasiert realisiert und werden somit direkt referenziert. Dann verursacht der Compiler auch nur einen Mehraufwand von wenigen Prozent, verglichen mit dem FORTRAN-Compiler.

3.2 Parallelisierung mit regulären Templates

Die Parallelisierung des Benchmarks wurde schrittweise durchgeführt: Zunächst wurden durch Einfügen der PDO-Direktive die Iterationen der Schleifen auf die Prozessoren aufgeteilt. Durch Ausrichten der Datenfelder auf Seitengrenzen und Einführung von Templates in einem weiteren Schritt wurden erste Zeitgewinne erzielt, da nun auch die Hardware und die Topologie der Gebiete berücksichtigt wurde. Letzte erhebliche Steigerungen der Ausführungsgeschwindigkeit ergaben sich durch Replizieren der Hauptschleife und eine generalisierte Blockverteilung des Template.

Die Parallelisierung der Schleifen (Abbildung 3.4) führt dazu, daß die Iterationen blockweise auf die Prozessoren der Anwendung verteilt werden. Die Verteilung der zugehörigen Daten ergibt sich daraus automatisch. Bei der verwendeten Blockverteilung (Standard) werden Seitengrenzen nicht berücksichtigt. Dadurch entsteht *false sharing*. Die hierdurch erzielten Leistungen sind für die verschiedenen Datensätze in Tabelle 3.2 aufgeführt. Der Ein-

```
C       Berechnungsschritt
CSVM$   PDO(LOOPS(NC), STRATEGY(BLOCK))
        DO NC = NINTCI, NINTCF
           ...
        END DO
```

Abbildung 3.4: Parallele DO-Schleife

Prozessoranzahl	1	2	4	8	16
Datensatz *tjunc*					
Laufzeit [s]	32,39	49,73	81,09	107,65	722,15
Rechenleistung [MFlops]	4,48	2,92	1,79	1,35	0,20
Datensatz *drall*					
Laufzeit [s]	122,58	97,19	151,45	146,66	350,27
Rechenleistung [MFlops]	4,01	5,05	3,24	3,35	1,40
Datensatz *pent*					
Laufzeit [s]	523,03	388,21	236,82	195,19	325,00
Rechenleistung [MFlops]	3,50	4,71	7,72	9,37	5,63

Tabelle 3.2: Rechenleistung der Version mit parallelen Schleifen

```
CSVM$   PDO(LOOPS(NC), REDUCTION(OCC),
CSVM$  >        STRATEGY(BLOCK))
        DO NC = NINTCI, NINTCF
              OCC = OCC + ADXOR(NC)...
        END DO
```

Abbildung 3.5: Reduktion einer Variablen

```
CSVM$   PROCESSORS:: P(numproc())
CSVM$   TEMPLATE:: T(47312)
CSVM$   DISTRIBUTE (CYCLIC(1024)) ONTO P:: T

CSVM$   SHARED, ALIGN:: DIREC2, ...
```

Abbildung 3.6: Ausrichten verteilter Felder und Templates

satz von mehreren Prozessoren verlangsamt bei den kleineren Datensätzen *tjunc* und *drall* eher die Ausführungszeit, nur bei dem *pent*-Datensatz gibt es Verbesserungen. Diese schlechten Werte lassen sich auf das *false sharing* zurückführen. Die Seiten werden permanent zwischen den Prozessoren ausgetauscht.

Es sei angemerkt, daß in einem Teil der DO-Schleifen Reduktionen vorkamen (Abbildung 3.5). Das Problem ist hier, daß die Summe von Feldelementen auf eine skalare Variable addiert werden und somit alle Prozessoren an dieser Summenbildung beteiligt sind. Der Compiler kann Reduktionen nicht erkennen, deshalb müssen sie als solche ausgewiesen werden.

Im nächsten Schritt wurde versucht, durch Ausrichten der verteilten Felder auf Seitengrenzen (Alignment) und Einführen von Templates Verbesserungen zu erreichen (Abbildung 3.6). Die Feldelemente der gemeinsamen Felder werden dann nicht mehr kontinuierlich auf den Seiten abgespeichert, sondern der Beginn eines Feldes wird auf eine Seitengrenze gelegt. Templates ermöglichen eine exaktere Steuerung der Verteilung. Durch das Ausrichten erreicht man, daß die durch die Templates spezifizierte Arbeitsverteilung für alle gemeinsamen Felder paßt, die ausgerichtet sind. Das Ausrichten auf Seitengrenzen ist nur für die Felder notwendig, auf die schreibend zugegriffen wird, die also auf der linken Seite einer Zuweisung stehen, da Seiten im lesenden Zugriff dupliziert in mehreren Prozessoren gehalten werden können.

Bei blockweiser Verteilung wird die Seitengröße nicht berücksichtigt. Die Prozessoren bekommen jeweils die gleiche Anzahl Iterationen. Im

Prozessoranzahl	1	2	4	8	16
Datensatz *tjunc*, BLOCK					
Laufzeit [s]	37,02	39,38	31,06	79,13	1387,69
Rechenleistung [MFlops]	3,92	3,68	4,67	1,83	0,10
Datensatz *tjunc*, CYCLIC					
Laufzeit [s]	37,41	47,75	32,50	50,72	221,75
Rechenleistung [MFlops]	3,88	3,04	4,46	2,86	0,65
Datensatz *drall*, BLOCK					
Laufzeit [s]	123,02	119,22	58,19	93,34	257,69
Rechenleistung [MFlops]	3,99	4,12	8,44	5,26	1,91
Datensatz *drall*, CYCLIC					
Laufzeit [s]	124,38	134,78	79,38	105,81	119,28
Rechenleistung [MFlops]	3,95	3,64	6,19	4,64	4,12
Datensatz *pent*, BLOCK					
Laufzeit [s]	585,99	409,19	148,25	251,94	254,00
Rechenleistung [MFlops]	3,12	4,47	12,33	7,26	7,20
Datensatz *pent*, CYCLIC					
Laufzeit [s]	590,81	908,88	232,34	499,16	213,78
Rechenleistung [MFlops]	3,09	2,01	7,87	3,66	8,55

Tabelle 3.3: Rechenleistung nach Alignment

vorliegenden Beispiel werden DOUBLE-PRECISION-Felder verteilt, d. h. jeweils 1024 Elemente passen auf eine Seite. Beim *drall*-Datensatz werden 47312 Knoten verteilt. Demnach werden jedem der vier Prozessoren 11828 Knoten, d. h. 11,5 Seiten zugewiesen.

Um die Berücksichtigung der Seitengrenzen zu ermöglichen, wurden die Template-Elemente versuchsweise zyklisch auf die Prozessoren verteilt (DISTRIBUTE(CYCLIC(1024)) ...), da es möglich ist, die Anzahl der Elemente festzulegen, die auf einen Prozessor gelangen.

Die Messungen haben ergeben, daß je nach Prozessoranzahl und Größe des Gebiets einmal die blockweise Verteilung, ein anderes Mal die zyklische Verteilung bessere Ergebnisse bringt (Abbildung 3.3).

Der letzte durchgeführte Optimierungsschritt – generalisierte Blockverteilung (Abbildung 3.7) und Replizieren der Hauptschleife – brachte noch einmal erhebliche Verbesserungen. Bei der generalisierten Blockverteilung wird das Template wieder blockverteilt, allerdings unter Berücksichtigung der Seitengrenzen. Durch einen Vektor kann SVM-Fortran veranlaßt werden, genau festgelegte Anzahlen von Iterationen auf die Prozessoren zu verteilen.

```
CSVM$   PROCESSORS:: P(numproc())
CSVM$   TEMPLATE:: T(:)
        ...
        INTEGER ITPROZ(16)
        ...
CSVM$   CREATE:: T(NINTCI : NINTCF)
        CALL ITVERT(NINTCI, NINTCF, 16)
CSVM$   REDISTRIBUTE (GENERAL_BLOCK(ITPROZ)) ONTO P:: T
```

Abbildung 3.7: Generalisierte Blockverteilung

Mit TEMPLATE:: $T(:)$ wird lediglich ein eindimensionales Template erzeugt, dessen Größe dynamisch zur Laufzeit mit der Direktive CREATE:: $T(NINTCI : NINTCF)$ kreiert wird.

Die Verteilung der Iterationen wird in dem Unterprogramm ITVERT berechnet. Übergeben werden Anfangs- und Enditeration der zuzuteilenden Iterationen – bei *drall* Iterationen 1 bis 47312 – und die Größe des Feldes ($ITPROZ$). Man verfährt dabei so, daß man den Prozessoren nur ganze Seiten zuteilt. Dazu muß man die Iterationen, wie bereits bei Einführung der zyklischen Verteilung erwähnt, jeweils in Vielfachen von 1024 zuteilen. In der Regel wird der letzte Prozessor weniger Arbeit erhalten, da die Anzahl der Iterationen im allgemeinen nicht ein Vielfaches von 1024 sein wird. Bei *drall* auf vier Prozessoren ergibt sich somit folgende Verteilung der Iterationen:

Prozessor	1	2	3	4
Anzahl der Iterationen	12088	12088	12088	10448

Da zwischen den Prozessoren keine Abhängigkeiten bestehen, ist es ohne weiteres möglich, den Kontrollfluß der Hauptschleife zu replizieren (Abbildung 3.8). Alle Anweisungen werden zwischen den globalen Synchronisationspunkten nun unabhängig von allen Prozessoren durchgeführt. Diese Barrieren können bis auf die Synchronisation vor dem Berechnungsschritt alle entfernt werden (Abbildung 3.9). Als Folge der Replizierung sind alle skalaren Variablen, die in einer replizierten Region auftreten, als PRIVATE zu deklarieren.

Die Tabelle 3.4, sowie die Abbildung 3.10 zeigen die Ergebnisse der Parallelisierungsbemühungen mit regulären Templates. Die parallele Version des Benchmarks läuft erst auf vier Prozessoren schneller als die serielle Version. Man sieht, daß der kleine Datensatz *tjunc* 16 Prozessoren nicht auslastet (Tabelle 3.4). Dies ist auch klar, wenn man sich vergegenwärtigt, daß nur Daten

```
CSVM$   PRIVATE:: OCC, ...
...
CSVM$   REPLICATED_REGION
100     CONTINUE
        ...
C       Berechnungsschritt
        DO NC = NINTCI, NINTCF
             DIREC2(NC) = BP(NC) * DIREC1(NC)
     >                    -BS(NC) * DIREC1(LCC(1,NC))
     >                    -BW(NC) * DIREC1(LCC(4,NC))
                          ...
     >                    -BH(NC) * DIREC1(LCC(6,NC))
        END DO
C--
        ...
9999    CONTINUE
CSVM$   REPLICATED_REGION_END
```

Abbildung 3.8: Replizierung der Hautpschleife

```
CSVM$   PDO (LOOPS(NC), REDUCTION(OCC), STRATEGY(BLOCK),
     >NOBARRIER)
        DO NC = NINTCI, NINTCF
             ...
        END DO
```

Abbildung 3.9: Entfernung überflüssiger Barrieren

Prozessoranzahl	1	2	4	8	16
Datensatz *tjunc*					
Laufzeit [s]	36,34	24,15	17,12	12,42	9,69
Rechenleistung [MFlops]	3,99	6,00	8,47	11,67	14,97
Datensatz *drall*					
Laufzeit [s]	123,72	74,17	42,74	28,38	20,25
Rechenleistung [MFlops]	3,97	6,62	11,49	17,31	24,13
Datensatz *pent*					
Laufzeit [s]	601,93	345,35	222,34	135,65	70,31
Rechenleistung [MFlops]	3,04	5,29	8,23	13,48	26,05

Tabelle 3.4: Rechenleistung der parallelisierten Version des AVL-FIRE-Benchmarks

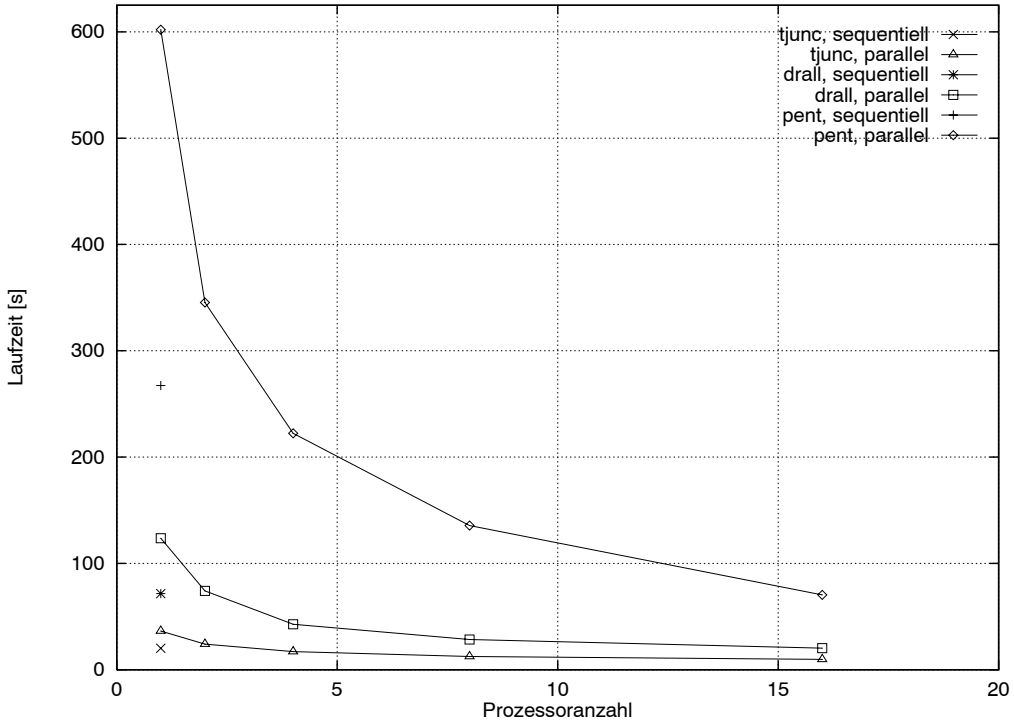


Abbildung 3.10: Laufzeiten des AVL-FIRE-Benchmarks

für etwas mehr als 13 Seiten zu verteilen sind und somit $2\frac{1}{2}$ Prozessoren quasi ungenutzt bleiben. Erst die beiden größeren Datensätze haben genügend Datenmaterial, um auch 16 Prozessoren auszulasten; allerdings geht auch schon *drall* in die Sättigung, da pro Prozessor zu wenig Daten anfallen, als daß der Kommunikationsaufwand durch die Rechenzeit aufgewogen wird.

Kapitel 4

Partitionierungsalgorithmen

Viele Algorithmen zur Untersuchung der Problemstellungen auf Parallelrechnern basieren auf Gittern (z. B. Finite-Element-Methode). Sie können durch Graphen $G = (V, E)$ dargestellt werden. Die Knotenmenge V repräsentiert die Gitterpunkte, auf denen Berechnungen ausgeführt werden. Die Menge der Verbindungen E sind die Beziehungen zwischen den Gitterpunkten. Zusätzliche Informationen, wie etwa der geometrische Ort eines Gitterpunktes, Rechenzeit für einen Punkt (also sein Gewicht) oder die Kosten der Kommunikation (das Gewicht der Verbindungen) können in die Partitionierung einfließen. Da die Zuordnung von Gitter und Graph eineindeutig ist, wird im folgenden von Gittern gesprochen, wenn Knoten und ihre Koordinaten betrachtet werden. In den Fällen, wo Bezug auf die Verknüpfungsinformationen genommen wird, soll von Graphen die Rede sein.

Für einen Parallelrechner mit N Prozessoren werden diese Gitter in N knotendisjunkte Teilgitter $G_1, \ldots, G_p, \ldots, G_N$ aufgeteilt. Jedes Teilgitter wird einem Prozessor zugeordnet. Diese Aufteilung in Teilgitter heißt *Partition*. Ziel des Partitionierens ist es, eine möglichst optimale parallele Ausführung, also durch die parallele Berechnung einen optimalen Speedup[1] zu erreichen.

Bei Parallelrechnern mit verteiltem Speicher wird die Güte einer Partition im wesentlichen durch zwei Punkte bestimmt:

- Eine gleichmäßige Lastverteilung soll erreicht werden, d. h. die Prozessoren haben etwa die gleiche Rechenleistung zu erbringen und benötigen somit gleiche Zeit für die Berechnung ihres Teilgitters.

- Die Kommunikation zwischen den Prozessoren soll minimal sein, da diese einen zusätzlichen Aufwand darstellt.

[1]Speedup S_N: Verhältnis der Ausführungszeit des sequentiellen Algorithmus, T_1, zur Ausführungszeit des parallelen Algorithmus auf N Prozessoren, T_N: $S_N = \frac{T_1}{T_N}$.

Das Problem, einen ungerichteten Graphen in mehrere Komponenten so aufzuteilen, daß jede Komponente möglichst die gleiche Anzahl von Knoten besitzt und dabei außerdem möglichst wenige Kanten geschnitten werden, ist schon für zwei Komponenten NP-vollständig [21].

Für hinreichend große Graphen ist daher kein effizientes Verfahren zur Konstruktion einer in diesem Sinne optimalen Partitionierung bekannt. Es ist darüber hinaus nach heutigem Wissensstand aussichtslos, nach einem derartigen Verfahren überhaupt zu suchen. In der Literatur werden eine Fülle von Heuristiken diskutiert, die die optimale Partitionierung mit geeignetem Aufwand approximieren.

In bezug auf Parallelrechner mit hohen Startup-Zeiten wird man versuchen solche Heuristiken anzuwenden, die die Anzahl benachbarter Komponenten minimiert unabhängig davon, wieviele Kanten zwischen zwei Komponenten geschnitten werden. Die Schnittstelle[2] sollte dagegen möglichst klein gehalten werden, wenn der begrenzende Faktor die Kommunikationsbandbreite ist.

Die hier dargestellten Verfahren sind nur in einer einfachen Form beschrieben, d. h. es wurde keine Gewichtung der Knoten bzw. Verbindungen eingeführt. Viele der Algorithmen lassen ein Knotengewicht entsprechend den Kosten der Berechnungen in einem Knoten und/oder ein Verbindungsgewicht entsprechend den Kommunikationskosten der Verbindung bzw. entsprechend der Kommunikationshäufigkeit zwischen zwei Knoten zu.[3] Um einen Überblick gewinnen zu können, soll im folgenden eine Einteilung und eine kurze Beschreibung der Verfahren versucht werden. In der Literatur wird üblicherweise eine Unterscheidung zwischen geometriebasierten und Graphinformation ausnutzenden Algorithmen getroffen (Saltz et al. [46]). Teilweise betrachtet man unter den geometriebasierten Algorithmen diejenigen gesondert, die zusätzlich eine Gewichtung zulassen (Ponnusamy [43], Bresany und Sipkova [9]). Alternativ findet man auch eine Unterscheidung zwischen Partitionierungs- und Optimierungsalgorithmen (Farhat et al. [16]).

In der Praxis scheinen sich neben relativ langsamen Partitionierern auch zweistufige Verfahren zu etablieren: Ein schnelles Verfahren wird durchgeführt, um eine initiale Partition zu kreieren; durch einen anschließenden Optimierungsschritt versucht man dann, die Kosten der Partition zu verringern, d. h. man versucht eine Partition zu erzeugen, die weniger Verbindungen zwischen den Teilgittern hat, als die Ausgangspartition.

[2]Schnittstelle eines Teilgraphen: Menge der Knoten einer Komponente, die eine Verbindung zu einem Knoten in einer benachbarten Komponente haben.

[3]In der Literatur war kein Hinweis auf Partitionierer zu finden, die Kommunikationskosten berücksichtigen. Diese Kategorie von Algorithmen scheint nicht zu existieren oder zumindest nicht üblich zu sein.

Da Partitionierungsalgorithmen sich nur auf heuristische Weise beliebig gut an eine optimale Lösung annähern können, existieren auch beliebig viele Verfahren. Deswegen kann das Thema in der vorliegenden Darstellung nicht erschöpfend behandelt werden. Die Artikel [14], [37] und [16] seien dem Leser als Startpunkt für eigene Erkundungen empfohlen.

Adaptive Gitter fordern nach effizienten Verfahren, die bei einer Neuverteilung die bestehende Partition berücksichtigen und die Zuordnung der Knoten zu Prozessoren weitestgehend erhalten. Einige iterative Verfahren sind verfügbar, die ein inkrementelles Vorgehen erlauben.

Die bekannten Verfahren berücksichtigen in der Regel keine Seitengrenzen. Deshalb können die erzeugten Partitionen zu einem erhöhten Seitenaustausch führen, der aber durch die in [60] beschriebenen Maßnahmen verhindert werden kann.

Für die nachfolgende Beschreibung bietet sich eine Einordnung nach algorithmischen Aspekten an. Somit ergeben sich die Unterkapitel *Rekursive Bisektionsalgorithmen*, *Direkte Partitionierungsalgorithmen* und *Optimierungsalgorithmen*. Anschließend werden die vorgestellten Algorithmen den in der Literatur gängigen Kategorien zugeordnet. Den Abschluß bildet ein Überblick der Algorithmen zur *Partitionierung von adaptiven Gittern* und die Beschreibung zweier Ansätze, um Cache- bzw. Seitengrenzen zu wahren.

4.1 Rekursive Bisektionsalgorithmen

Zu den rekursiven Bisektionsalgorithmen gehören Rekursive Koordinatenbisektion [5, 48], Rekursive Inertialbisektion [17, 58, 38], Rekursive Graphbisektion [48] und das Verfahren der Rekursiven Spektralbisektion [4, 48, 44, 3]. All diese Algorithmen basieren auf dem gleichen algorithmischen Gerüst [3] (Abbildung 4.1).

Als Eingabe erhält der Algorithmus das zu partitionierende Gitter G und die Anzahl der Prozessoren N. Die rekursiven Bisektionsalgorithmen bedienen sich alle des *Divide-and-Conquer*-Paradigmas: In bisect() wird einer der im folgenden beschriebenen Bisektionsalgorithmen ausgeführt und das Gitter in Teilgitter G_0 und G_1 zerlegt. Daraufhin werden wiederum die beiden Teilgitter partitioniert (partitionieren()). Abschließend werden die zwei Partitionen zu einer Gesamtpartition zusammengesetzt (zusammensetzen()). Es ist offensichtlich, daß sich die Bisektionsverfahren in dieser Form nur für die Zerlegung in 2^k Teilgitter verwenden lassen. Trisektion usw. lassen sich aber dennoch sehr einfach herleiten, indem in bisect() nicht in zwei, sondern in drei oder mehr Teilgitter zerlegt wird[4] und partitionieren() nicht nur zweimal, sondern entsprechend öfter aufgerufen wird.

[4]Man nennt es dann besser multisect(G, n, $G_1, \ldots, G_n$) o.ä., wobei n angibt, wieviele Teilgitter im Multisektionsschritt erzeugt werden.

```
partition bisektionsalgorithmus(gitter G, int N)
{
    gitter G_0, G_1;
    partition p_0, p_1; /* partition: Menge von Teilgittern */

    if (N > 1)
    {
        bisect(G, G_0, G_1);
        p_0 = partitionieren(G_0, N/2);
        p_1 = partitionieren(G_1, N/2);
        return (zusammensetzen(p_0, p_1));
    }
    else
    {
        return ({G});
    }
}
```

Abbildung 4.1: Pseudocode für Bisektionsalgorithmen

4.1.1 Rekursive Koordinatenbisektion (RCB)

RCB ist ein einfaches, geometrisches Verfahren, bei dem die Menge der Knoten des Gitters entlang der Koordinatenachse mit der längsten Ausdehnung angeordnet und dann halbiert wird (Abbildung 4.2). Dieser Vorgang wird rekursiv auf jedem Teilgitter wiederholt, bis die gewünschte Anzahl von Teilgittern erreicht ist. Dies führt zu langen Teilgittern mit großen Schnittstellen. Da die Verbindungsinformationen nicht in den Algorithmus eingehen, kann man im allgemeinen nicht erwarten, daß die Teilgitter zusammenhängend sind.

Eine Variante der RCB ist die Orthogonale Koordinatenbisektion (OCB) [36], bei der nach jeder Rekursion die Sortierachse gewechselt wird. Es wird also nicht jeweils die für das Teilgitter längste Ausdehnung gesucht. Die mit OCB erzeugten Partitionen haben folglich ein schachbrettartiges Muster.

Eine zweite, nichtrekursive Variation der RCB ist Geometrisches Sortieren [36]. Damit läßt sich eine Abbildung auf ein $(p \times q)$-Prozessorfeld erreichen: Die Knoten werden zunächst entlang der längsten Achse angeordnet und dann in $\frac{|V|}{p}$ Teilgitter zerteilt. Die Knoten jedes Teilgitters werden dann längs der orthogonalen Achse geordnet und in $\frac{|V|}{pq}$ Teilgitter geteilt, wobei $|V|$ der Anzahl der Gitterknoten entspricht.

```
void bisect(gitter G, G_0, G_1)
{
    Dimension mit längster Ausdehnung bestimmen;
    Knoten entlang der ausgewählten Dimension sortieren;
    G_0 = erste Hälfte der Knoten;
    G_1 = zweite Hälfte der Knoten;
}
```

Abbildung 4.2: Pseudocode für RCB

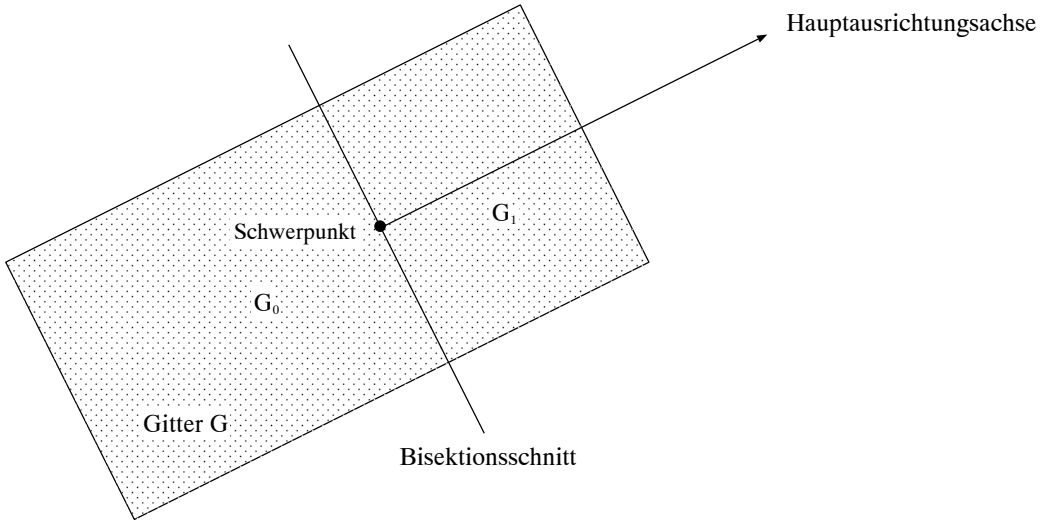


Abbildung 4.3: Bisektionsschritt der RIB

4.1.2 Rekursive Inertialbisektion (RIB)

Bei der RIB (auch *Recursive Principal Inertia, (RPI)*) bemüht man ein physikalisches Analogon: Es wird angenommen, daß jeder Knoten eine Masse hat. Man bestimmt den Schwerpunkt des Gitters und teilt dort durch einen geraden Schnitt orthogonal zur Hauptausrichtungsachse des Gitters (Abbildung 4.3). Die Hauptausrichtungsachse ist i. a. nicht identisch mit einer der drei Raumrichtungen. In die Berechnung der Hauptausrichtungsachse fließen die Euklidischen Abstände der Knoten vom Schwerpunkt des Gitters ein (Abbildung 4.4). Da zur Bestimmung ein Eigenwertproblem zu lösen ist, ist dieses Verfahren zwar langsamer als RCB und OCB, gehört aber immer noch zu den schnellen Methoden und führt zu besseren Ergebnissen bei konkaven Geometrien [36].

```
void bisect(gitter G, G_0, G_1)
{
    Hauptaussrichtungsachse bestimmen;
    Gitter orthogonal zur Hauptausrichtungsachse zerschneiden;
    G_0 = erste Hälfte der Knoten;
    G_1 = zweite Hälfte der Knoten;
}
```

Abbildung 4.4: Pseudocode für RIB

```
void bisect(gitter G, G_0, G_1)
{
    zwei Knoten mit maximalem Abstand bestimmen;
    restliche Knoten nach Abstand von diesen Knoten ordnen;
    G_0 = erste Hälfte der Knoten;
    G_1 = zweite Hälfte der Knoten;
}
```

Abbildung 4.5: Pseudocode für RGB

4.1.3 Rekursive Graphbisektion (RGB)

Die RGB ähnelt vom Konzept her der RIB. Während jedoch die RIB den Euklidischen Abstand zum Sortieren der Knoten benutzt, werden im RGB-Algorithmus die Abstände im Graphen verwendet (Abbildung 4.5): Zu Beginn werden zwei Knoten bestimmt, die maximalen Abstand haben. Dann werden die restlichen Knoten in Schichten gleicher Entfernung von diesen Knoten angeordnet (Abbildung 4.6). Die Knoten werden dem jeweils nächsten der beiden Startknoten zugeordnet. Somit nutzt der RGB-Partitionierungsalgorithmus die Verbindungsinformation des Gitters.

4.1.4 Rekursive Spektralbisektion (RSB)

Die Heuristik RSB (Abbildung 4.7) nutzt die Informationen, die im sogenannten Fiedler-Vektor enthalten sind. Der Fiedler-Vektor ist ein Eigenvektor zum zweitkleinsten Eigenwert der Laplace-Matrix L eines Graphen $G = (V, E)$. Die Laplace-Matrix ist eine $(|V| \times |V|)$-Matrix und ist durch

$$L = D - A$$

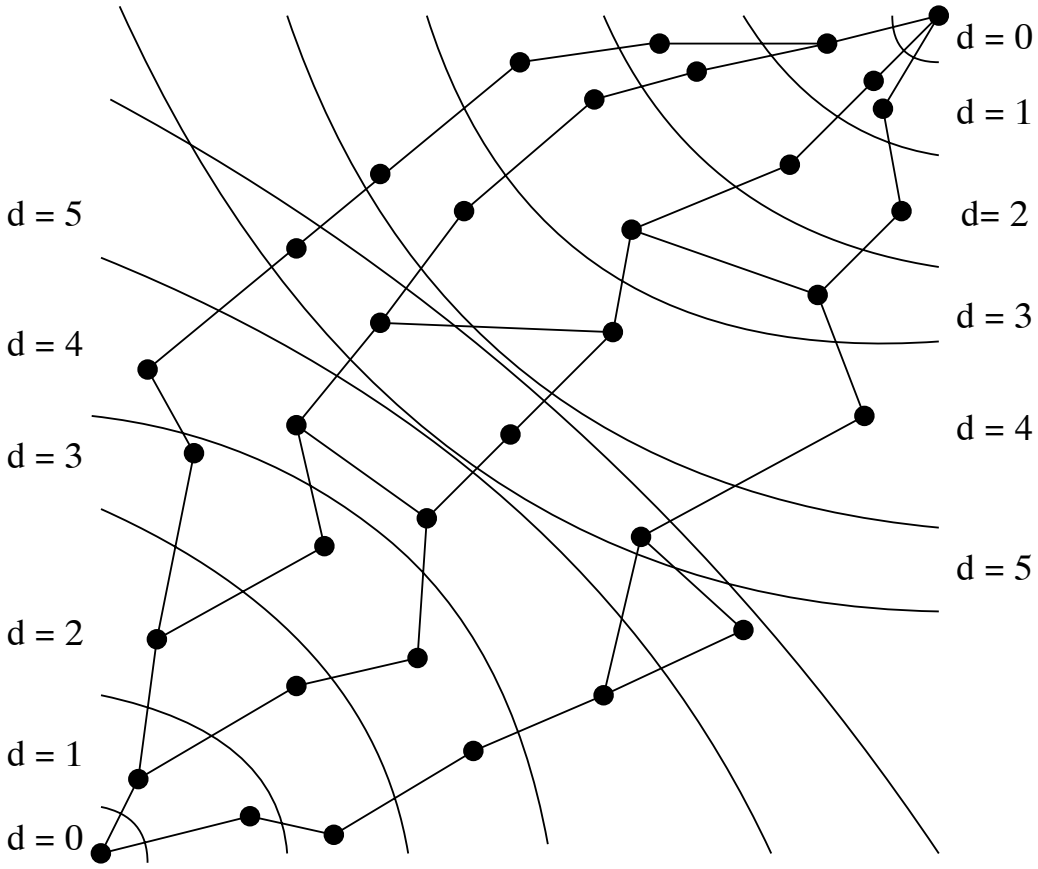


Abbildung 4.6: Schichten gleichen Abstandes in der RGB: Der Abstand d zum nächstliegenden Startknoten wird durch die eingezeichneten Kreisbögen verdeutlicht.

```
void bisect(gitter G, G_0, G_1)
{
    Fiedler-Vektor von G berechnen;
    Knoten gemäß Eintrag im Fiedler-Vektor ordnen;
    G_0 = erste Hälfte der Knoten;
    G_1 = zweite Hälfte der Knoten;
}
```

Abbildung 4.7: Pseudocode für RSB

$$\begin{pmatrix} -0,348 & [1] \\ -0,234 & [2] \\ -0,098 & [3] \\ 0,047 & [4] \\ -0,431 & [5] \\ 0,288 & [6] \\ 0,257 & [7] \\ 0,187 & [8] \end{pmatrix} \qquad \begin{pmatrix} -0,431 & [5] \\ -0,348 & [1] \\ -0,234 & [2] \\ -0,098 & [3] \\ 0,047 & [4] \\ 0,187 & [8] \\ 0,257 & [7] \\ 0,288 & [6] \end{pmatrix}$$

(a) (b)

Abbildung 4.8: (a) Einträge des Fiedlervektors; (b) Fiedlervektor und Einträge nach dem Umsortieren

definiert. Dabei bezeichnet

$$d = \mathrm{diag}(\deg r), v \in V$$

eine Matrix, die den Grad jedes Knotens $v \in V$ als Diagonalelemente enthält und A die Adjazenzmatrix des Graphen G, d. h.

$$A = (a_{vw})_{v,w \in V} \text{ mit } a_{vw} = \begin{cases} 1 & \text{es gibt in } E \text{ eine Kante zwischen } v, w \\ 0 & \text{sonst} \end{cases}$$

Wählt man den speziellen Graphen mit $|V| = 8$ Knoten aus [53] als Beispiel, dann ist der zugehörige Fiedler-Vektor in Abbildung 4.8 (a) angegeben. Im Bisektionsschritt werden die Einträge des Fiedlervektors der Größe nach angeordnet (Abbildung 4.8 (b)); die Knoten, welche nun der oberen Hälfte der Vektoreinträge zugeordnet sind (5, 1, 2, 3), bilden das erste, die restlichen Knoten (4, 8, 7, 6) das zweite Teilgebiet der Partition.

Heute verwendet man aufgrund der Größe der Probleme und des Rechenaufwandes für die Berechnung des Fiedlervektors ($\mathcal{O}(N\sqrt{N})$; diese Berechnung dominiert im wesentlichen die Laufzeit der RSB) eine *Multilevel*-Version der RSB, genannt MRSB. Der (zu große) Graph wird in mehrere kleine Graphen aufgeteilt, die bezüglich der Partitionierung die gleichen Eigenschaften wie der Gesamtgraph haben. Diese werden mit RSB partitioniert, und eine Partition des Gesamtgraphen wird dann extrapoliert. Die kleineren Graphen kann man z. B. durch Verschmelzen von einzelnen Knoten oder durch Zusammenfassen dichter Gebiete erzeugen.

In [3] wird eine parallele Version rekursiver Bisektionsalgorithmen am Beispiel der RSB beschrieben[5], welche auf der Rekursivität der Methode aufbaut. Der Partitionierungsalgorithmus (Abbildung 4.1) kann mit einem *task-team* – einer Gruppe von Prozessoren, die gemeinsam, unabhängig von den restlichen Prozessoren einen Teil der Bisektion ausführen – effizient umgesetzt werden, jeder Prozessor besitzt einige Reihen der Laplace-Matrix. Zu Beginn sind alle Prozessoren in die Bisektion einbezogen. Dann werden sie in zwei halb so große *task-teams* aufgeteilt, die unabhängig voneinander die beiden Untermatrizen partitionieren usw., bis zuletzt jeder einzelne Prozessor ein *task-team* bildet.

4.2 Direkte Partitionierungsalgorithmen

Direkte Partitionierungsalgorithmen führen die Partitionierung in iterativer Weise durch. Zu diesen Verfahren gehören die Greedy-Algorithmen [15, 17], das Reverse-Cuthill-McKee-basierte Verfahren [33] und One-dimensional Topology [52]. Weiterhin existieren auch iterative Variationen der Rekursiven Partitionierungsalgorithmen (Principal Inertia (PI) [17], Koordinatenbisektion [16], Geometrisches Sortieren [36]).

4.2.1 Greedy-Algorithmen (GR)

Die Algorithmen dieser Klasse werden *greedy* (engl. gierig) genannt, weil sie im Prinzip in das Gitter „hineinbeißen“, um die Teilgitter zu konstruieren. Sie starten mit einem Knoten minimalen Grades und vereinigen $\frac{|V|}{N}$ Knoten zu einem Teilgitter. Dieser Vorgang wird auf dem Restgraph wiederholt, bis der ganze Graph verteilt ist. Da GR jeden Knoten nur einmal besucht, ist er ein schneller Algorithmus ($\mathcal{O}(N)$). Die Vorgehensweise des Algorithums kann zu unzusammenhängenden Teilgittern führen. Man kann dies verhindern, indem die Partitionierung in solchen Fällen mit einem anderen Startpunkt wiederholt wird.

Die Greedy-Algorithmen liefern meist – in Verbindung mit einer anschließenden lokalen Optimierung – eine gute Partition. Die Teilgitter sind durch Kompaktheit und eine kleine Schnittstelle charakterisiert.

Ein einfacher Greedy-Algorithmus (Abbildung 4.9) nach [17][6], der für die Partitionierung von Finite-Elemente-Gitter eingesetzt wird, beruht nur auf Verbindungsinformationen. In [1] wird eine Variation präsentiert, die auch

[5] An der Stelle der Rekursiven Spektralbisektion kann hier auch jedes beliebige andere Bisektionsverfahren eingesetzt werden.

[6] Hier sind die Kosten als Anzahl der Knoten des Gitters definiert.

```
partition greedyAlgorithmus(gitter G, int N)
{
    gitter G_1, ..., G_N;

    kosten = |G|; /* kosten: Berechnungskosten von G */
    for (i = 1; i ≤ N; i++)
    {
        if (i == 1)
        {
            G_1 = {beliebiger Knoten mit minimalem Grad};
        }
        else
        {
            Startknoten = beliebiger Knoten minimalen Grades aus der
                Schnittstelle von G_{i-1};
            G_i = ∅;
        }
        while (kosten_i < kosten/N)
        {
            G_i mit unmarkierten Nachbarknoten vereinigen;
            kosten_i = |G_i|; /* kosten_i: Berechnungskosten von G_i */
        }
    }
    return (zusammensetzen(G_1, G_2, ..., G_N));
}
```

Abbildung 4.9: Pseudocode für GR

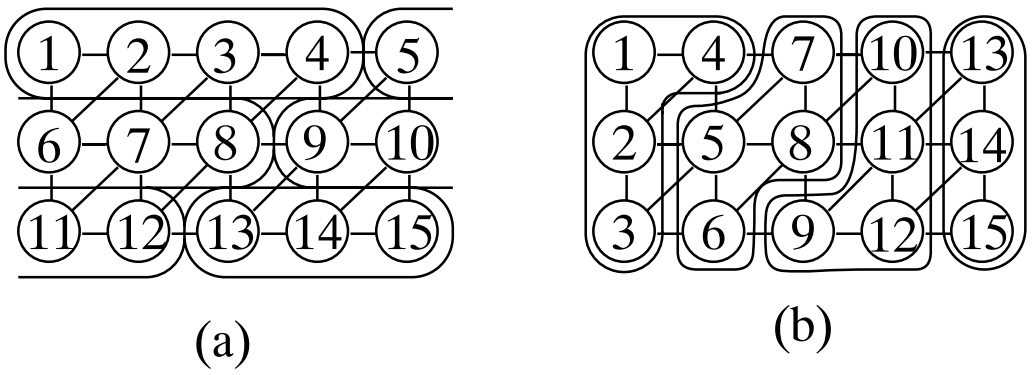


Abbildung 4.10: Partitionierung eines Gitters gemäß der Knotennummern: (a) bei horizontaler Numerierung der Knoten; (b) bei vertikaler Numerierung

die geometrische Information berücksichtigt und dadurch zu besseren Ergebnissen führt.

4.2.2 Reverse-Cuthill-McKee-basierter Algorithmus (RCM)

Der hier beschriebenen Partitionierungsalgorithmus RCM wurde speziell für Finite-Elemente-Gitter entwickelt und basiert auf dem Reverse-Cuthill-McKee-Verfahren zur Bandbreitenminimierung [11].

Das RCM numeriert die Knoten des Gitters so um, daß bei einer Blockverteilung miteinander kommunizierende Knoten möglichst auf dem gleichen Prozessor landen. Eine naive Verteilung der Knoten ohne diese Umnumerierung berücksichtigt die Verbindungen und damit die Kommunikation zwischen den Knoten nicht explizit und führt deshalb i.a. in bezug auf die Kommunikation nicht zu einer optimalen Partition. In Abbildung 4.10 (a) ist die Schnittstelle der vier Teilgitter größer als in Abbildung 4.10 (b).

Die Kommunikationskosten stehen mit der *Bandbreite*[7] der Adjazenzmatrix A des Gitters (bereits in Kapitel 4.1.4 definiert) in Beziehung. Eine Vorgehensweise, die Kommunikationskosten zu reduzieren, ist eine Minimierung der Bandbreite. Denn bei geringer Bandbreite darf man hoffen, Lokalität besser ausnutzen zu können. Nimmt man eine Blockverteilung der Matrix auf die Prozessoren an, so erwartet man geringere Kommunikationskosten, wenn die Bandbreite der Matrix minimiert wird.

[7]Bandbreite: $2 \cdot \max_{a_{ij} \neq 0}(|i-j|)$.

```
partition reverseCuthillMcKee(gitter G)
{
    Knoten in der Adjazenzmatrix umsortieren (RCM);
    Finite Elemente in absteigender Reihenfolge sortieren;
    Elemente blockweise den Prozessoren zuordnen;
    Knoten auf Prozessoren verteilen;
}
```

Abbildung 4.11: Pseudocode für RCM

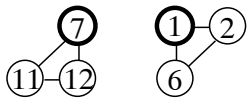


Abbildung 4.12: Sortierkriterium der Elemente am Beispiel zweier Elemente des Beispielgitters aus Abbildung 4.10

Der gesamte Partitionierungsalgorithmus besteht aus vier Schritten (Abbildung 4.11). Der erste Schritt des Algorithmus ist das beschriebene Umsortieren der Zeilen der Adjazenzmatrix. Das Reverse-Cuthill-McKee-Verfahren eignet sich für ein solches Umsortieren. Dies führt zu einer Umnumerierung der Knoten. Auch andere Verfahren zur Bandbreitenreduktion sind geeignet.

Da der Algorithmus von Malone ursprünglich für die Partitionierung von Finite-Element-Gittern gedacht war, wurde besonders darauf geachtet, daß die Knoten, die zu einem Element gehören, möglichst auf dem gleichen Prozessor zu liegen kommen. Dazu werden die Elemente nach dem RCM-Schritt entsprechend umsortiert. Der Knoten mit der kleinsten Nummer ist das Sortierkriterium: Die Elemente werden gemäß ihres kleinsten Knotens absteigend sortiert (Abbildung 4.12).

Daraufhin werden die Elemente den Prozessoren blockweise zugeordnet. Die Knoten werden im letzten Schritt entsprechend den Elementen auf die zugehörigen Prozessoren verteilt. Knoten, die zwei oder mehr Elemente auf verschiedenen Prozessoren angehören, erfahren eine Sonderbehandlung. Zunächst wird der Durchschnitt der Prozessornummern der involvierten Prozessoren gebildet. Dann wird derjenige Knoten dem Prozessor zugewiesen, dessen Nummer diesem Durchschnitt am nächsten kommt.

Ein Vorteil des RCM ist, daß die Anzahl der Prozessoren beliebig sein kann, d. h. keine Zweierpotenz sein muß. Nachteilig am RCM ist, daß er zur Erzeugung sehr langer Teilgitter mit großen Schnittstellen neigt und deshalb

```
partition oneDimensionalTopology(gitter G)
{
gitter G_p, G_1, ... , G_N;

    G_p = G;
    Durchmesser des Graphen G_p berechnen;
    L = {n_1}; /* n_1: erster Startknoten */
    for (i = 1; i < N; i++)
    {
        n_2 = Knoten in G_p mit größtem Abstand zu L;
        restliche Knoten nach Abstand von L bzw. n_2 ordnen;
        while ((G_p ≠ ∅) ∧(|V_i| < |V|/N))
        {
            wähle ein k aus der Nähe von L;
            G_i = G_i ∪ {k};
            G_p = G_p \ {k};
        }
        G_{i+1} = {k|k ist Nachbar von G_i, k ∈ G_p};
        L = G_{i+1};
        }
    }
    return (zusammensetzen(G_1, ... , G_N));
}
```

Abbildung 4.13: Pseudocode für 1DT

nicht geeignet ist für Parallelrechner mit relativ kleinen Startup und großen Übertragungskosten.

4.2.3 One-dimensional topology (1DT)

1DT ist ein direkter Partitionierungsalgorithmus (Abbildung 4.13), der von der RGB (Kapitel 4.1.3) abgeleitet wurde. Er führt zu *topologischer Linearität*, d. h. jede Komponente hat höchstens zwei Nachbarn. Komponente G_{i+1} wird bestimmt, indem zunächst die unmarkierten Nachbarknoten von Komponente G_i markiert werden; hierdurch wird die topologische Linearität garantiert. Dann werden weitere Knoten zu G_{i+1} hinzugefügt. Dies geschieht nach einem der RGB ähnlichen Verfahren: einer der beiden Startknoten liegt auf der Grenze zwischen G_i und G_{i+1}, der zweite ist, entsprechend RGB, ein Knoten mit maximaler Entfernung. Der Komponente G_{i+1} werden so lange Knoten zugeteilt, bis $|V_{i+1}| = \frac{|V|}{N}$ gilt.

```
partition optimierungsalgorithmus(gitter G)
{
    partitionInitialisieren();
    kosten = bewertePartition();
    while (keine Konvergenz)
    {
        optimierung();
        kosten = bewertePartition();
        if (kosten < kosten_der_alten_Partition)
        {
            akzeptiere Änderung;
        }
        else
        {
            akzeptiere Änderung nur mit bestimmter Wahrscheinlichkeit;
        }
        if (keine signifikanten Veränderungen in den letzten Iterationen)
        {
            Konvergenz = true;
        }
    }
}
```

Abbildung 4.14: Pseudocode für Optimierungsalgorithmen

4.3 Optimierungsalgorithmen

Zu den Optimierungsalgorithmen gehören Simulated Annealing [34, 59, 31], Neuronale Netze [34, 27, 19],ein Hybrider Genetischer Algorithmus [34, 35, 25], Tabu-Suche [52, 36] und Kernighan-Lin [30, 37]. Die Optimierungsalgorithmen werden einerseits als Optimierungsschritt im Anschluß an eine Partitionierung durch einen der anderen Algorithmen (z. B. RCB) angewendet, andererseits werden sie – insbesondere die sogenannten *Physikalischen Optimierungsalgorithmen* Simulated Annealing, Neuronale Netze und Genetischer Algorithmus – auch als eigenständige Partitionierungsalgorithmen angesehen.

Die grundsätzliche Methode der Optimierungsalgorithmen ist in Abbildung 4.14 dargestellt. Nachdem durch partitionInitialisieren() eine initiale Partition erzeugt wurde, wird diese bewertet (bewertePartition()). Läuft der Optimierungsschritt nachfolgend zu einem anderen Partitionierungsalgorithmus

ab, so wird die dort erzeugte Partition als Ausgangspunkt für die Optimierungen verwendet, sonst ist diese Startpartition eine zufällig erzeugte. Solange ein bestimmtes Konvergenzkriterium – keine signifikanten Änderungen der Kosten in den letzten Iterationsschritten – nicht erfüllt ist, wird der Optimierungsschritt ausgeführt und die entstandene Partition bewertet (optimierung() und bewertePartition()). Ist die neue günstiger als die Ausgangspartition, so wird jene weiterverwendet und die alte verworfen. Im anderen Falle wird die neue Partition nur mit einer bestimmten Wahrscheinlichkeit angenommen. Je öfter der Optimierungsschritt ausgeführt wird, je mehr Zeitschritte also durchlaufen wurden, desto geringer wird die Wahrscheinlichkeit, daß eine Verschlechterung der Partition akzeptiert wird.

In [51] sind Kostenfunktionen aufgeführt, die zur Optimierung der Schnittstellengröße, der Lastbalance und der Kompaktheit der Teilgitter herangezogen werden können. Meist ist eine gewichtete Kombination aus diesen einzelnen Kostenfunktionen zu bilden, da ein Kriterium für die Beschreibung des Sachverhaltes nicht ausreicht.

Die sogenannten physikalischen Optimierungsalgorithmen (Simulated Annealing, Neuronale Netze und Genetischer Algorithmus) müssen einen Kompromiß zwischen der Effizienz des Verfahrens und Qualität des Ergebnisses finden: Bei kurzer Laufzeit, sprich großer Zeitschrittweite, ist die Gefahr einer frühzeitigen Konvergenz gegeben, d. h. man verbleibt in einem vermeintlichen, lokalen Minimum. Um diesen ungewünschten Effekt zu vermeiden, müssen die Zeitschritte genügend klein gewählt werden; die Veränderungen von Schritt zu Schritt sind dann nicht zu groß. Man hat dann nur das Problem, einen ineffizienten Algorithmus vorliegen zu haben. Die richtige Schrittweite und eine angemessene Laufzeit kann man nur durch Experimentieren bestimmen. Der Gedanke der Parallelisierung der Optimierungsalgorithmen ist naheliegend.

4.3.1 Simulated Annealing (SA)

Beim SA bedient man sich eines Ansatzes der statistischen Mechanik. Die Vorgänge während des langsamen Erkaltens in einem erstarrenden Körper werden simuliert. Bei hoher Temperatur können sich die Moleküle frei bewegen; mit Sinken der Temperatur nimmt der Bewegungsdrang der Moleküle zunehmend ab, bis der Körper vollständig abgekühlt ist.

Die Bewegung der Moleküle wird durch zufällige Störungen in der Partition simuliert: Eine bestimmte Anzahl von Knoten wird zwischen den Teilgittern verschoben. Ergibt sich dadurch eine bessere Partition, so werden die Änderungen angenommen; eine Verschlechterung des Zustandes wird dagegen nur mit einer bestimmten, temperaturabhängigen Wahrscheinlichkeit ange-

nommen. Je mehr die Temperatur absinkt, desto unwahrscheinlicher wird es, daß Änderungen akzeptiert werden, welche die Kosten erhöhen.

In einer parallelen Version dieses Algorithmus (PSA, [34]) wird der sequentielle SA-Algorithmus auf N Prozessoren gleichzeitig ausgeführt. Jeder Prozessor erhält einen disjunkten Teil der Knoten, auf dem er SA ausführt. Die Ergebnisse von PSA weichen von den Ergebnissen der seriellen Version ab, da globale Informationen, speziell die Anzahl der Knoten auf jedem der Prozessoren, während der Partitionierung nur gelegentlich zwischen den Prozessoren ausgetauscht werden. Hier ergibt sich das für die Physikalischen Optimierungsalgorithmen typische Problem: Häufiges Austauschen dieser Informationen führt zwar zu Lösungen wie beim SA, verhindert aber jeglichen Speedup; seltenes Austauschen hingegen führt zur Degeneration. Gemildert wird dieses Problem allerdings dadurch, daß bei sinkenden Temperaturen die Anzahl der akzeptierten Verschlechterungen der Partition abnimmt und somit der parallele Algorithmus sich dem sequentiellen Simulated Annealing annähert.

Laut [17] eignet sich SA nicht für sehr große Gitter, da es ein sehr langsames Verfahren ist. In [52] wird bemerkt, daß die Kosten für das Verfahren allerdings vertretbar sind, wenn man sich beim Austauschen auf die Knoten an der Schnittstelle eines Teilgitters beschränkt.

4.3.2 Bold Neural Network (BNN)

Ein Neuronales Netz nach Hopfield, genannt *Auto-Assoziator* (Abbildung 4.15), besteht aus einer Anzahl von *Zellen*, die untereinander verbunden sind. Zu dem Netzwerk gehören Verbindungsstärken w_{kl} zwischen Zelle k und Zelle l. Somit ist die *Matrix der Verbindungsstärken* W definiert durch $W = (w_{kl})_{k,l \in \{1,\ldots,|V|\}}$. Jede Zelle kann den Wert $+1$ oder den Wert -1 annehmen: Man spricht dann vom *Spin* $s(k)$ $+1$ oder -1. Eine andere Möglichkeit ist, Spinwerte *zwischen* $+1$ und -1 zuzulassen und somit ein kontinuierliches Modell zu entwerfen. Der *Zustand* s des Netzes ist beschrieben durch $s = (s(1), \ldots, s(|V|))$. Das System durchläuft eine Folge von Zuständen, wobei man von einem Zustand s zum folgenden Zustand s' gelangt, indem eine *Update-Regel* auf die Spins angewendet wird. In die Update-Regel fließen die gewichteten Spinwerte aller Knoten ein.

Ordnet man jedem Zustand eine *Energie* zu, so entsteht über dem Zustandsraum aller möglichen Spinzustände ein *Energierelief*. Durch Updates versucht man, die Energie zu minimieren, d. h. man gelangt in (möglicherweise lokale) Minima des Reliefs.

Hopfields Auto-Assoziator wird für das Bold Neural Network (BNN) (Abbildung 4.16) modifiziert. Die Zellen stellen nun die Knoten dar, die auf die

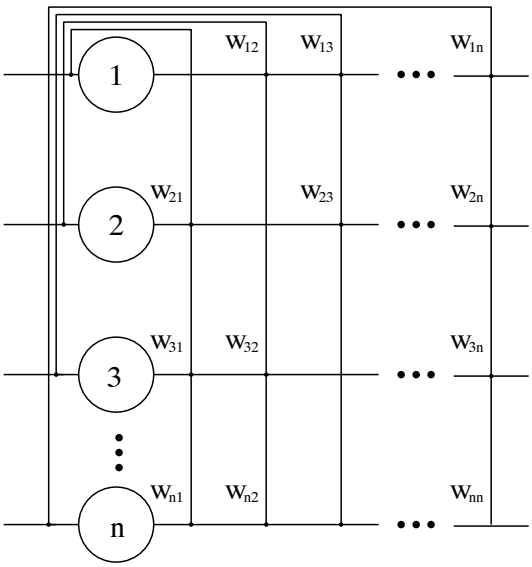


Abbildung 4.15: Auto-Assoziator nach Hopfield

Prozessoren verteilt werden. Der Spin $s(k)$ ist nun ein *Vektor* von Spins: $s(k) = (s(k,1), s(k,2), \ldots, s(k, \log_2(N).\ s(k))$ wird fortan als *Spinvektor* bezeichnet; $s(k,i)$ sei im folgenden ein *Spin*. Jedem Knoten sind $\log_2(N)$ Spins $s(k,i)$ zugeordnet. Dabei ist k ein Gitterknoten ($k \in V$) und i zeigt an, daß der Spin mit dem $i-$ten Bit der Prozessornummer korrespondiert. Die Spins ergeben nach Durchlaufen des Algorithmus die Adresse des Prozessors in binärer Schreibweise. Deshalb haben die Spins sinnvollerweise auch die Werte 0 und 1. Ist beispielsweise $N = 8$, dann existieren 3 Spins, $s(k,0)$, $s(k,1)$ und $s(k,2)$ für jedes $k \in V$. Um die Nummer des Prozessors festzulegen, der einen bestimmten Knoten bearbeitet, muß der Algorithmus bei N Prozessoren $\log_2(N)$-mal durchlaufen werden; in jedem Durchlauf wird ein Bit der Prozessornummer festgelegt.

In jedem Durchlauf wird die Update-Regel auf die Spins angewendet. Die Update-Regel verändert den Wert der Spins entsprechend der Werte der Nachbarspins und der Summe der Spinwerte in einem Teilgitter. Die Update-Regel kann in [19] nachgeschlagen werden.

Die Energie entspricht im BNN den Kosten der Partition. Hat man Konvergenz erreicht, geben die Spins die Aufteilung auf zwei Teilgitter an. Die Knoten, welche den Spinwert 1 haben, bilden das erste, die restlichen Knoten das zweite Teilgitter. Im nächsten Durchlauf wird der Algorithmus nach Art der Bisektionsalgorithmen jeweils getrennt auf die beiden Teilgitter angewendet. Haben die oben aufgeführten Spins z. B. die Werte $s(k,0) = 1$,

```
partition boldNeuralNetwork(gitter G, int N)
{
    gitter G_0, G_1;
    partition p_0, p_1; /* partition: Menge von Teilgittern */

    if (N > 1)
    {
        i = log_2 N - 1
        s(k,i) belegen; /* s(k,i) ∈ {0,1} */
1       do /* Optimierung */
        {
            for (k ∈ G)
            {
                Update(s(k,i));
            }
        }
        while (keine Konvergenz)
        G_0 = {k|s(k,i) = 0};
        G_1 = {k|s(k,i) = 1};
        p_0 = boldNeuralNetwork(G_0, N/2);
        p_1 = boldNeuralNetwork(G_1, N/2);
        return (zusammensetzen(p_0, p_1))
    }
    else
    {
        return (G)
    }
}
```

Abbildung 4.16: Pseudocode für BNN

$s(k,1) = 0$ und $s(k,2) = 1$, so wird Knoten k Prozessor 101_2, also Prozessor 5, zugeordnet.

Wie im PSA wird auch in der parallelen Version dieses Algorithmus (PNN [34]) der sequentielle Algorithmus auf den N Prozessoren, allerdings nur auf einem Teilgitter, ausgeführt. Nach mehreren lokalen Schritten werden in einem globalen Schritt Informationen zwischen den Prozessoren ausgetauscht. Auch hier hat man zu PSA vergleichbare Probleme: Es ist ein Kompromiß zwischen Laufzeit und Exaktheit zu suchen. Ein schneller PNN-Algorithmus hat Abweichungen von den Ergebnissen des BNN-Algorithmus, und tauscht man die Informationen oft zwischen den Prozessoren aus, so erzielt man keine Beschleunigung.

4.3.3 Hybrider Genetischer Algorithmus (HGATA)

Bei HGATA greift man auf den „Algorithmus“ Evolution zurück. Chromosomen (hier: Partitionen) werden über eine Anzahl von Generationen beobachtet. Ein *Chromosom* ist hier ein Vektor der Länge $|V|$. Im Eintrag i ist der Prozessor vermerkt, der Knoten i besitzt. Dieser Eintrag wird *Allel* des Chromosoms genannt. Partner werden entsprechend ihrer *Fitneß* f_i gekreuzt („reproduziert“). Die Fitneß des Chromosoms i ist das Verhältnis der Kosten $kosten_i$ der Partition i zu den durchschnittlichen Kosten $\overline{kosten}$ der Gesamtpopulation:

$$f_i = \frac{kosten_i}{\overline{kosten}}.$$

Zur *Reproduktion* werden die Chromosomen nach einer bestimmten Regel zugelassen: Sie werden $\lfloor f_i \rfloor$-mal in eine Liste geschrieben, aus der Paare zum *Crossing Over* ausgewählt werden (Abbildung 4.17). Zusätzlich werden sie mit einer gewissen Wahrscheinlichkeit, gegeben durch die Nachkommastellen von f_i, in die Liste eingetragen. Diese Liste ist eine Zwischengeneration.

Die anschließende *Rekombination* besteht aus *Crossing Over* und Mutation. Bei dem *Crossing Over* werden die Allele eines Chromosomenpaars von einem zufällig gewählten Rekombinationspunkt an ausgetauscht. Um Gerechtigkeit zwischen den Allelen zu schaffen, führt man ein 2−Punkt-*Crossing-Over* ein, d. h. es werden zwei Punkte auf dem Vektor gewählt, zwischen denen die Allele modulo der Vektorlänge ausgetauscht werden (Abbildung 4.18). Führt man nur ein 1−Punkt-*Crossing-Over* durch, also wählt man einen beliebigen Punkt auf dem Chromosom und tauscht von dort aus die Allele in einer beliebigen Richtung aus, so wären die Allele in Randnähe hin benachteiligt, da deren Austauschwahrscheinlichkeit sinkt.

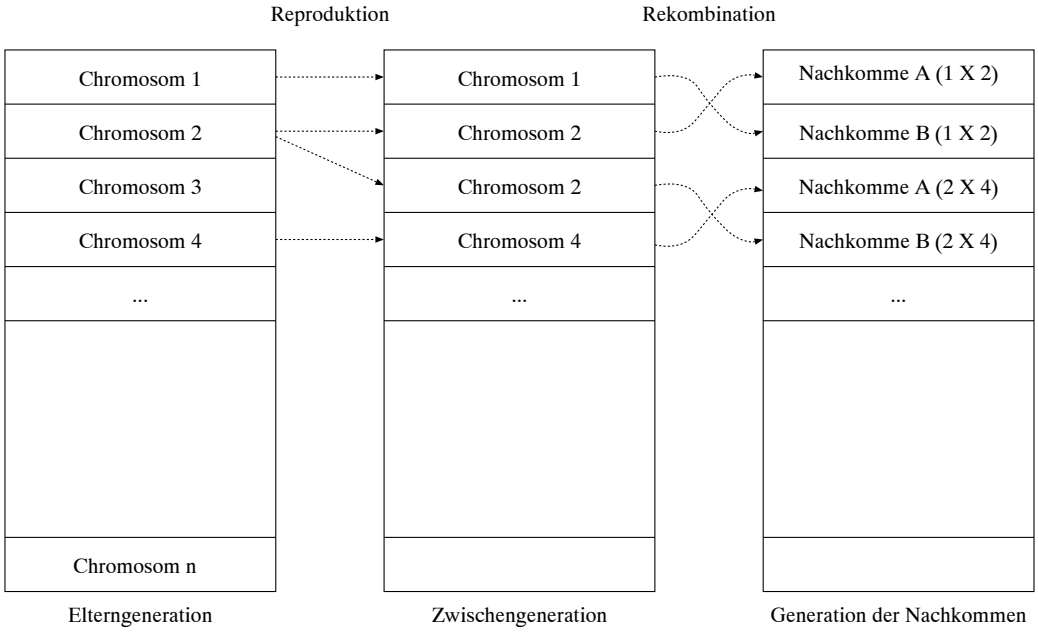


Abbildung 4.17: Reproduktion und Rekombination im Genetischen Algorithmus unter Zuhilfenahme einer Zwischengeneration

11010	\/	0110	\/	0101101	yxyyx0110yyxyxxy
yxyyx	/\	yxxy	/\	yyxyxxy	11010yxxy0101101
	Punkt 2		Punkt 1	→	
	(a)				(b)

Abbildung 4.18: 2−Punkt-*Crossing-Over*: Chromosomenpaar vor (a) und nach (b) dem *Crossing Over*

Auf die zwei Nachkommen wird *Mutation* angewendet: Der einem Allel zugeordnete Prozessor wird zufällig, allerdings nur mit einer sehr kleinen Wahrscheinlichkeit ($< 1\%$), gegen einen anderen Prozessor ausgetauscht.

Der vorliegende Algorithmus HGATA ist ein *Hybrider Genetischer Algorithmus*, d. h. es wurde eine „Lernkomponente", *Hill Climbing*, integriert. Beim *Hill Climbing* wird ein Knoten zwischen Prozessoren ausgetauscht, wenn durch diesen Austausch die Kosten der Partition gesenkt werden. Dies ist nur der Fall, wenn Knoten von überlasteten Prozessoren zu den anderen verschoben werden.

Bei der parallelen Version (PGA) [34] wird die Population, also die Gesamtheit der Chromosomen, in Sippen aufgeteilt (disjunkte Unterpopulationen). Die Sippen werden den Rechenknoten zugeordnet. Die lokale Evolution wird durch einen sequentiellen HGATA über eine bestimmte Anzahl von Generationen nachgebildet. In bestimmten Abständen findet Selektion zwischen den Sippen statt: Sippen mit größerer Fitneß vergrößern sich und ersetzen Teile unterlegener, benachbarter Sippen. Dies wird durch eine Wanderung der besten Chromosomen von den fitteren Sippen zu den weniger fitten Nachbarsippen erreicht. Nur in dieser Phase gibt es eine Kommunikation zwischen den Knoten. PGA ist effizient, da die Kommunikationskosten im Vergleich zur lokalen Berechnungszeit gering gehalten werden.

4.3.4 Kernighan-Lin (KL)

KL beruht auf der Überlegung, daß ein logisches Austauschen von Knoten relativ schnell geht. In einer logischen Austauschsequenz werden alle Knoten zweier Teilgitter so paarweise ausgetauscht, daß jeder der $|V|$ Knoten genau einmal bewegt wird. Es wird jeweils das Knotenpaar miteinander getauscht, welches den höchsten Gewinn bringt; dies werden Knoten sein, die an einem Schnitt liegen und die ein großes Ungleichgewicht zwischen der Anzahl der Verbindungen zu Knoten im eigenen Teilgitter und der Anzahl der Verbindungen zu Knoten in Nachbargittern besitzen. Da alle Knoten einmal in einem Austausch vorkommen, wird man auch Schritte mit negativem Gewinn ausführen müssen (Abbildung 4.19).

Hat man die Vertauschungen logisch durchgeführt, sucht man die Stelle, die ein Kostenminimum erzeugt, danach steigen die Kosten wieder an. In der Abbildung ist das Minimum als punktierte Linie eingezeichnet. Physikalisch werden nun alle Vertauschungen bis dahin ausgeführt.

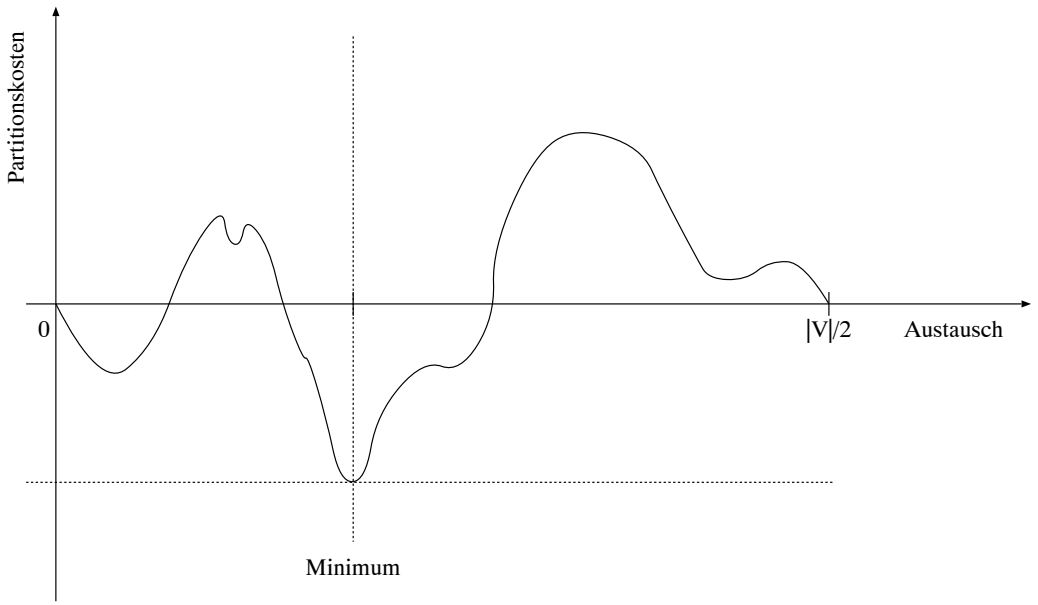


Abbildung 4.19: Eine logische Austauschsequenz in der KL-Heuristik

4.3.5 Tabu-Suche (TS)

TS ist eine iterative Verbesserungstechnik, die auf kombinatorischer Optimierung basiert. Ähnlich wie bei KL werden auch hier zunächst virtuell Vertauschungen durchgeführt. Wo KL aufeinanderfolgende Vertauschungen durchführt, werden in TS verschiedene Variationen des gleichen Schrittes ausprobiert.

In jedem Optimierungsschritt werden mehrere Knoten versuchsweise ausgetauscht. Tatsächlich wird nur die Vertauschung ausgeführt, die die besten Kosten bringt. Diese Änderung kann aber auch zu einer Verschlechterung der Kosten führen, auch dies wird akzeptiert.

Das Steckenbleiben in lokalen Minima versucht man zu vermeiden, indem die bewegten Elemente in eine *Tabuliste* eingetragen werden. Änderungen dieser Knoten sind tabu. Zyklen in der Suche werden werden somit unterbunden.

Da in TS mehrere Varianten des gleichen Schrittes probiert werden, ist auch eine Parallelisierung des Verfahrens einfacher möglich als bei KL. Die verschiedenen Prozessoren können parallel und unabhängig voneinander jeweils eine Variation eines Schrittes berechnen, wovon die beste Lösung weiterverwendet wird. Alternativ ließe sich auch – ähnlich dem genetischen Algorithmus – die Parallelisierung räumlich ansetzen: Für eine bestimmte Anzahl

Partitionierungsalgorithmen	Optimierungsalgorithmen
RCB, OCB, Geometrisches Sortieren	SA, PSA
RIB, PI	BNN, PNN
RGB	HGATA, PGA
RSB	TS
GR	KL
RCM, RRCM	
1DT	
SA, PSA	
BNN, PNN	
HGATA, PGA	

Tabelle 4.1: Einteilung der Algorithmen in Partitionierungs- und Optimierungsalgorithmen

von Schritten agieren die Prozessoren auf knotendisjunkten Ausschnitten des Gitters. In einem globalen Austauschschritt werden Knoten über das gesamte Gitter, auch zwischen verschiedenen Prozessoren, ausgetauscht.

4.4 Einteilungsmöglichkeiten

Die beiden Tabellen 4.1 und 4.2 ordnen die in den vorangegangenen Kapiteln vorgestellten Algorithmen anderen Kategorien zu. In Tabelle 4.1 werden Partitionierungs- und Optimierungsalgorithmen unterschieden. SA, BNN, GA und deren parallele Versionen werden in der Literatur teilweise als Partitionierungs-, teilweise aber auch als Optimierungsalgorithmus geführt und sind deshalb in beiden Kategorien aufgeführt. Tabelle 4.2 ordnet die Algorithmen in Graphinformation und in Geometrieinformation nutzende Algorithmen.

4.5 Partitionierung adaptiver Gitter

Adaptive Gitter dienen zur Darstellungen von zeitabhängigen, unstrukturierten Phänomenen, die den Lösungsraum durchqueren und deshalb eine Verfeinerung an der fokussierten Stelle notwendig werden lassen. Solche Gitter werden nach einigen Schritten neu berechnet und partitioniert. Die Anpassung muß also schnell sein im Vergleich zum Rechenaufwand.

	Geometrieinformation	Graphinformation
RCB, OCB	◇	
RIB, PI	◇	
RGB		◇
RSB		◇
GR	(◇)	◇
RCM, RRCM		◇
1DT		◇
SA, PSA		◇
BNN, PNN		◇
HGATA, PGA		◇
TS		◇
KL		◇

Tabelle 4.2: Informationen, die von Partitionierungsalgorithmen verwendet werden

Neben der Möglichkeit, wiederholt schnelle Partitionierungsalgorithmen zu verwenden, die allerdings nicht so gute Ergebnisse liefern, kann man auch versuchen, die Partitionierungsinformation des alten Gitters weiterzuverwenden und durch vergleichsweise einfache Anpassungsschritte zum neuen Gitter zu gelangen. Dies hat zusätzlich den nicht zu unterschätzenden Vorteil, daß die Verteilung der Knoten größtenteils erhalten bleibt. Würde man diese Information nicht nutzen, würden viele der Knoten zwischen den Prozessoren ausgetauscht. Dieses Kommunikationsaufkommen ist zu vermeiden.

4.5.1 Paralleler Partitionierer (Par^2)

Par^2 [13] besteht aus zwei Phasen. In der ersten Phase wird das zu partitionierende Gitter mittels eines einfachen Partitionierungsalgorithmus (beispielsweise GR, siehe Kapitel 4.2.1) auf die Prozessoren verteilt *(fast initial clustering)*.

Die zweite Phase optimiert die initiale Verteilung der Knoten im Hinblick auf die Lastverteilung. Sie läuft parallel in den einzelnen Prozessoren ab. Zunächst wird für jedes Paar von Teilgittern die Anzahl der Knoten festgestellt, die verschoben werden müßten, um eine optimale Lastverteilung zu erreichen. Daraufhin wird die notwendige Anzahl von Knoten, unter Berücksichtigung von Kommunikationsanforderungen und der Form der Teilgitter, ausgetauscht.

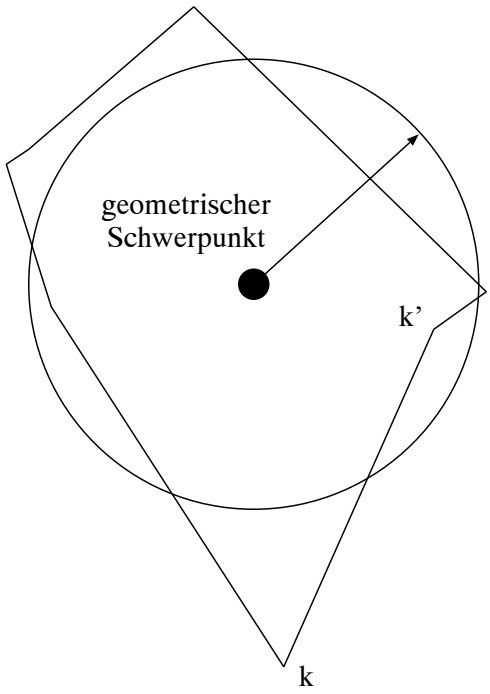


Abbildung 4.20: Optimierung von Teilgebietsformen

Der Austausch der Knoten geschieht einzeln nacheinander. Für einen Knoten k und ein Teilgitter G_p wird der *Nutzen* $b(k, G_p)$ eines Austausches definiert. Ist ein Knoten migriert, wird b erneut für alle Knoten berechnet, um so den nächsten auszutauschenden Knoten zu bestimmen. Der Nutzen b ist definiert als

$$b(k, G_p) = \omega_1 \cdot g(k, G_p) + \omega_2 \cdot f(k, G_p),$$

wobei $g(k, G_p)$ die *Schnittstellenverkleinerung,* also die Änderung in der Anzahl der geschnittenen Verbindungen, wenn k verschoben wird, und $f(k, G_p)$ ein Maß für die *Verbesserung der Form des Teilgitters* G_p ist, falls k verschoben wird. Als optimale Form wird im Zweidimensionalen der Kreis und im Dreidimensionalen die Kugel angestrebt, da diese geometrischen Figuren die kleinste Oberfläche im Verhältnis zum umgebenden Inhalt haben. Die Verbesserung f berücksichtigt die Änderung der Form sowohl in dem Teilgitter, in dem Knoten k entfernt werden soll, als auch die Änderung in dem Teilgitter, in das k verschoben werden soll. Formverbessernd wirkt also der Austausch von „Ausreißern“ (Abbildung 4.20). Ein Entfernen des

Knotens k bringt das Teilgitter eher zur Kreisform als das Austauschen des Knotens k'. ω_1 und ω_2 sind Gewichtungsfaktoren, die eine Anpassung der Nutzenfunktion an die Rechnerarchitektur zulassen.

Abschließend wird versucht, durch Anwendung von *Hill Climbing* die Zerlegung zusätzlich zu optimieren. Nach einer Adaption, d. h. Vergröberung oder Verfeinerung des Gitters, muß lediglich die zweite Phase des Algorithmus ausgeführt werden. Es ist der Literatur ([13]) allerdings nicht zu entnehmen, wie neue Knoten und zugehörige Verbindungen in die bestehende Partition eingearbeitet werden.

Ganz ähnlich verfahren Walshaw, Cross und Everett [56]: Nachdem eine sequentielle initiale Verteilung (mit GR) ausgeführt wurde, wird versucht, die Form der Teilgitter zu optimieren (Kreis- bzw. Kugelform der Teilgitter). Anders als bei Diekmann et al. wird hier nicht gleichzeitig auch auf eine Lastbalance geachtet. Es wird versucht, diese in einem nachfolgende Schritt zu erreichen. Abschließend wird durch eine Optimierung mittels einer parallelen Variante der Kernighan-Lin-Heuristik versucht, die Schnittstellen zwischen den Teilgittern zu minimieren. Die letzten drei Schritte können parallel und/oder nach einem adaptiven Schritt ausgeführt werden.

4.5.2 Dynamische Rekursive Spektralbisektion (DRSB)

In ihrer dynamischen Version des RSB (siehe Kapitel 4.1.4) versuchen Walshaw und Berzins den bewährten RSB-Algorithmus so anzupassen, daß er für Aufgaben mit adaptiven Gittern anwendbar wird [55]. Problematisch an der RSB in Bezug auf adaptive Gitter ist die Zeitkomplexität der sogenannten Lanczos-Methode zur Berechnung des Fiedler-Vektors ($\mathcal{O}(N\sqrt{N})$). Ein weiterer Ansatzpunkt für Verbesserungen bietet die Tatsache, daß nach einem weiteren Lauf des RSB-Algorithmus kleine Veränderungen im Gitter sich nicht notwendigerweise durch nur kleine Änderungen in der Partition widerspiegeln.

Geht man davon aus, daß eine Veränderung im Gitter eine Verfeinerung oder Vergröberung darstellt, so kann eine neue Partition aus der alten interpoliert werden, also die alte Partition als Ausgangspunkt für einen Repartitionierungsalgorithmus verwenden.

In dem dynamischen Algorithmus wird angenommen, daß ein Austausch von Knoten, die nahe der Schnittstelle liegen, in den meisten Fällen ausreicht. Dazu werden die Knoten zum Mittelpunkt der Teilgitter hin in Schichten angeordnet. Die Schnittstellenknoten bilden die Schicht L_1, in der Schicht L_2 befinden sich alle Knoten, die eine Verbindung zu einem Knoten in der

Schicht L_1 haben usw. Als Eingabe für die RSB dienen die Knoten der Schicht L_1 und ein innerer Knoten, der die restlichen Knoten des Teilgitters repräsentiert. Zwischen den Knoten in L_1 und dem „schweren“ Knoten werden die Verbindungen aus dem Originalgraphen eingefügt. Die Knoten von L_1 sind untereinander wie im ursprünglichen Graphen verbunden.

Um zu berücksichtigen, daß in dem inneren Knoten eine gewisse Anzahl von Gitterknoten zusammengefaßt ist, wird der zugehörige Eintrag im Fiedler-Vektor entsprechend oft aufgeführt. Tritt im Bisektionsschritt der Fall auf, daß der Schnitt durch diesen Knoten geht, so gilt die Bisektion als gescheitert. In diesem Falle werden die Knoten und Verbindungen der Schicht L_2 aus dem inneren Knoten entfernt und wieder als Einzelknoten betrachtet. So wird Schicht um Schicht dazugenommen, bis der Graph für die Bisektion genügend groß ist.

4.5.3 Inkrementelle Graphpartitionierung (IGP)

Ou und Ranka stellen in [40] ein inkrementelles Verfahren vor, welches ebenfalls auf der Rekursiven Spektralbisektion aufsetzt. Eine initiale Partition wird durch die RSB erzeugt und in den darauffolgenden Adaptionsschritten an die neuen Anforderungen angepaßt.

Die inkrementelle Phase setzt sich aus vier Schritten zusammen. Im ersten Schritt werden die hinzukommenden Knoten der bestehenden Partition zugeordnet. Man sucht den geometrisch nächstgelegenen Knoten der alten Partition und ordnet den neuen dem zugehörigen Teilgraph zu. Dadurch entstehen Teilgraphen unterschiedlicher Größe. Also müssen Knoten unter der Berücksichtigung der Kommunikationskosten zwischen Teilgraphen ausgetauscht werden. Dazu wird in einem zweiten Schritt für jeden Knoten der nächste Nachbargraph bestimmt. Bei der praktischen Durchführung der Lastbalancierung wird darauf geachtet, daß möglichst wenige Knoten verschoben werden, da die Änderungen an der Partition dann gering sind. Da im Lastbalancierungsschritt nicht explizit auf die Minimierung der Schnitte geachtet wurde, wird die Partition in einem letzten Schritt weiter verbessert. Dabei wird ein Knoten des Teilgitters i am Rand zu Teilgitter j dann ausgetauscht, wenn er mehr Verbindungen zu Knoten in j als zu Knoten in i hat.

Eine *Multilevel*-Version dieses Algorithmus existiert und wird auch in [40] beschrieben. Wie die Teilgitter für diesen Algorithmus zusammengefaßt werden müssen, soll hier nicht ausführlich beschrieben werden. Es sei lediglich erwähnt, daß auf jeder Ebene versucht wird, jeweils zwei Teilgitter zu einem gröberen Gitter zusammenzufassen.

4.5.4 Indexbasierte Partitionierung (IBP)

In einem älteren Bericht von Ou, Ranka und Fox wird ein inkrementelles Verfahren auf der Basis der indexbasierten Partitionierung (IBP) vorgestellt [41]. Dabei werden die mehrdimensionalen Koordinaten der Knoten quasi in den eindimensionalen Raum der Knotenindizes „hineingefaltet". Die Koordinaten eines Knotens werden in der binären Schreibweise betrachtet und in einen eindimensionalen Index überführt. Dazu werden die Koordinaten der Dimensionen nacheinander durchlaufen. Die Bits einer Dimension werden dazu verschränkt (interleaved) – also nicht von links nach rechts – gelesen und an den eindimensionalen Index angehangen. Die Knoten werden dann entsprechend des Index sortiert und in N Teillisten aufgeteilt, die den Prozessoren zugeordnet werden. Dieses Verfahren ist leicht zu parallelisieren.

Neue Knoten können einfach in die bestehende Partition eingearbeitet werden, indem deren eindimensionaler Index bestimmt wird und diese dann in die sortierten Listen der Knoten der einzelnen Teilgitter einsortiert werden. Da man im allgemeinen nicht davon ausgehen kann, daß die neuen Knoten gleichmäßig auf die Partition verteilt werden, muß im Anschluß für einen Lastausgleich gesorgt werden.

4.6 Berücksichtigung von Seiten in Partitionierern

Bei einem seitenorientierten Speicher, wie bei der Intel Paragon, tritt das Problem des *false sharing* von Speicherseiten auf: Zwei Prozessoren greifen schreibend auf Daten zu, die sich auf der gleichen Seite, aber an unterschiedlichen befinden. Ein gleichzeitiges Schreiben der Beteiligten wäre also ohne Konflikt möglich, widerspricht aber der Implementierung des gemeinsamen Speichers, die auf Speichereinheiten der Größe einer Seite aufbaut. Die Prozessoren entziehen sich also überflüssigerweise gegenseitig die Seite. Partitionierungsalgorithmen beachten Seitengrenzen im allgemeinen nicht. Deshalb werden die mit den Knoten korrespondierenden, verteilt gespeicherten Felder in der Regel nicht entsprechend der Seitengrenzen geschnitten. In der Abbildung 4.21 ist ein Feld im gemeinsamen Speicher zweier Prozessoren dargestellt. Das Feld belegt 18 Seiten. Drei der Seiten sind teilweise mit Feldelementen belegt, die Prozessor 1 bzw. Prozessor 2 gehören.

Zeng und Abraham stellen in [60] zwei Techniken vor, die solche Aufteilungen verhindern bzw. beseitigen. Einmal wird ein Präprozeß durchgeführt, der das Zerschneiden von Seiten verhindert (Verschmelzungsalgorithmus). Zum anderen wird ein Algorithmus vorgestellt, der in einem nachgeschalte-

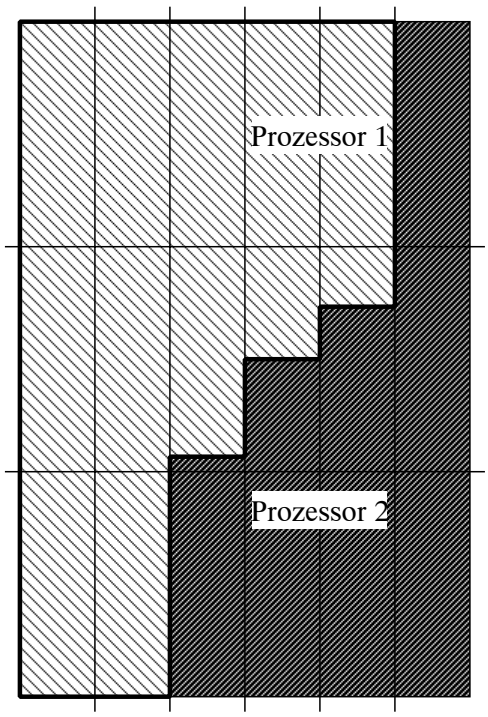


Abbildung 4.21: Partitionierung eines Feldes im gemeinsamen Speicher

ten Schritt solche unerwünschten Verteilungen berichtigt (Kompensationsalgorithmus).

4.6.1 Verschmelzungsalgorithmus

Bei dem *Verschmelzungsalgorithmus* werden einfach alle Feldelemente, die auf einer Seite liegen, zu einem Knoten im Graphen des Partitionierungsalgorithmus zusammengefaßt. Dieser Knoten erhält als Gewicht die Anzahl der zusammengefaßten Knoten. Somit bietet sich dieser Algorithmus an, wenn ein Partitionierer verwendet wird, der Knotengewichte berücksichtigt. Zwei Knoten i und j werden dann verbunden, wenn zur Berechnung einer der Knoten auf Seite i ein oder mehrere Knoten der Seite j benötigt werden.

Bedauerlicherweise berücksichtigen die momentan verfügbaren Partitionierer keine Verbindungsgewichte. Dies würde ermöglichen, daß eine gewichtete Verbindung zwischen i und j eingeführt werden könnte, deren Gewicht angibt, wieviele der Knoten der Seite j zur Berechnung der Knoten der Seite i benötigt werden.

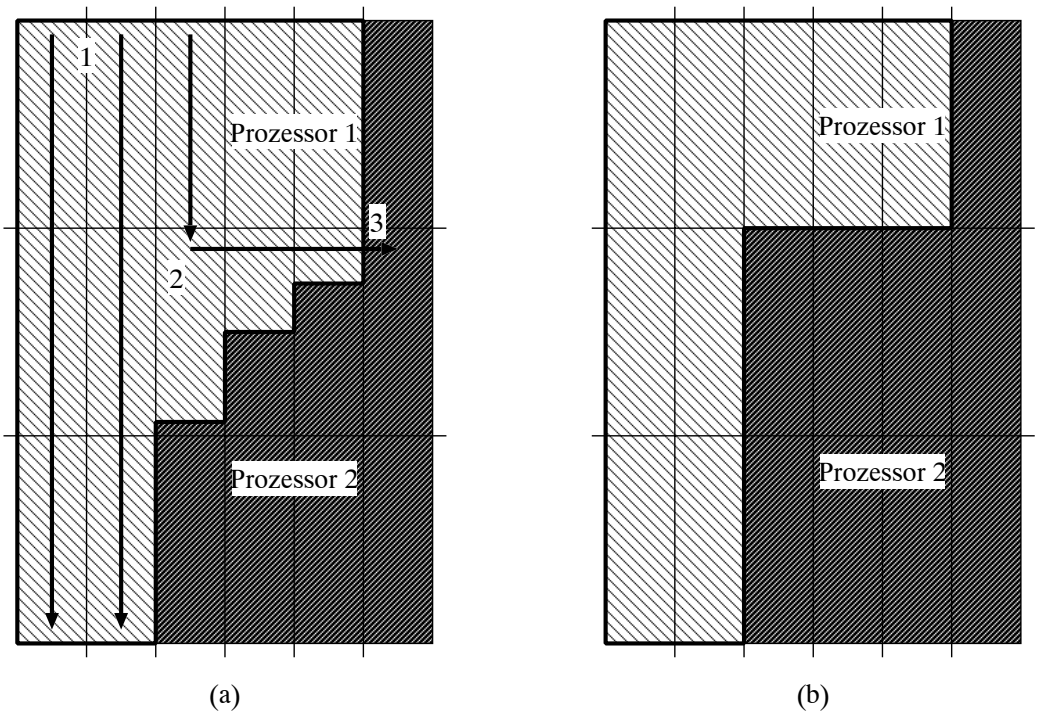


Abbildung 4.22: Kompensationsalgorithmus: (a) Durchlaufen der Seiten im Kompensationsalgorithmus (b) Partition nach Ausführen des Kompensationsalgorithmus

Der Verschmelzungsalgorithmus ist nur anwendbar, wenn die zu verteilenden Felder so groß sind, daß genügend Speicherseiten auf die Prozessoren verteilt werden können.

4.6.2 Kompensationsalgorithmus

Der *Kompensationsalgorithmus* wird auf eine Partition, wie sie von einem Partitionierungsalgorithmus geliefert wird, angewendet. Es handelt sich also um eine Nachbehandlung.

Der Algorithmus sucht nach Feldern, die zwischen zwei oder mehr Prozessoren aufgeteilt sind und behandelt diese gesondert (Abbildung 4.22 (a)). Solange ganze Seiten dem ersten Prozessor gehören, werden die zugehörigen Feldelemente spaltenweise durchlaufen (1); an der Zuteilung zum ersten Prozessor ändert sich nichts. Detektiert man allerdings eine Seite mit Elementen zweier Prozessoren (2), so ändert sich die Laufrichtung ins Waagerechte, bis eine Spalte erreicht wird, die zu dem anderen Prozessor gehört (3). Alle Sei-

ten oberhalb des Pfeiles $\overline{23}$ verbleiben bei Prozessor 1. Die restlichen Seiten werden nun Prozessor 2 zugeordnet (Abbildung 4.22 (b)).

Kapitel 5

Implementierungsaspekte

Da in einem unstrukturierten Template Daten gesammelt werden, die in einem Programm normalerweise nicht mehr direkt zur Verfügung stehen, müssen diese Daten in der Template-Struktur gespeichert werden. Die im vorliegenden Kapitel präsentierten Funktionen ermöglichen das Zusammenstellen dieser Informationen und den Zugriff darauf.

Vor der Implementation war zu klären, ob die Daten im gemeinsamen Speicher oder in den lokalen Speichern der einzelnen Prozessoren abzulegen sind. Für das Abspeichern im privaten Speicher spricht, daß die Daten eines Teilgitters lokal vorhanden sind. Die Schnittstellenknoten werden allerdings dann zum Problem: Um den Wert des Knotens aus dem Nachbargitter zu bekommen, muß der zugehörige Prozessor das Datum senden. Der Ablauf der Kommunikation wäre mit explizitem Nachrichtenaustausch im wesentlichen der folgende: Sei P_1 der anfordernde Knoten. P_2 hat das gewünschte Datum d in seinem lokalen Speicher.

P_1 :	P_2 :
send$(request, P_2)$;	receive$(request, P_1)$;
receive(d, P_2);	send(d, P_1);

Das erste send-/receive-Paar für die Variable $request$ deutet an, daß P_1 d bei P_2 anfordert. Zwei Probleme treten auf: P_1 ist aufgrund der SVM-Philosophie nicht bekannt, auf welchem Prozessor d liegt, der Zielprozessor ist allerdings anzugeben. P_1 könnte eine Nachricht an alle Prozessoren schicken und so das Problem umgehen. Doch hier tritt das zweite Problem auf: Die Nachricht an P_2 würde nicht empfangen werden, da die send-Anweisung keinen Prozessor unterbrechen kann und somit auch nur ein Datenaustausch möglich ist, wenn beide Kommunikationspartner dazu bereit sind. Der Zugriff auf Daten anderer Prozessoren müßte also schon zum Zeitpunkt der

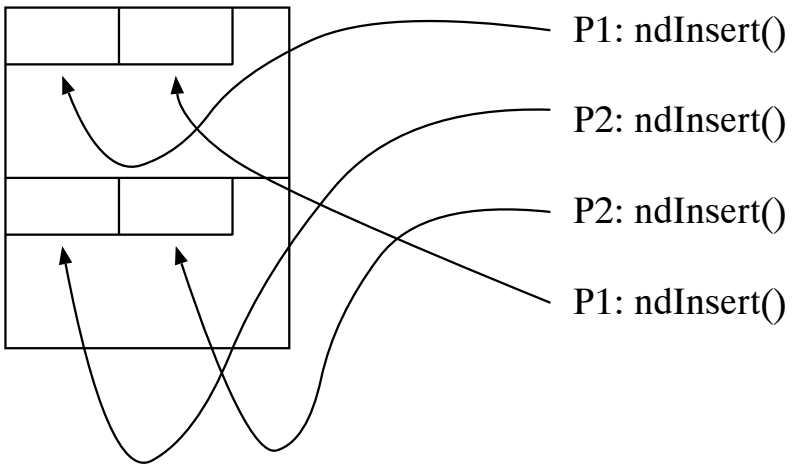


Abbildung 5.1: Berücksichtigung der Seitengrenzen beim Speichern der Knoten

Übersetzung bekannt sein. Dies ist aber wegen der Irregularität der Daten nicht der Fall.

Um diese Probleme zu umgehen, sind die Daten in den gemeinsamen Speicher gelegt worden. Einige Details müssen beachtet werden, damit nicht unnötigerweise permanent Seiten zwischen den Prozessoren verschoben werden. Besonders muß darauf Wert gelegt werden, daß die Aufteilung in Seiten bei der Speicherung berücksichtigt wird. Für jeden Prozessor muß grundsätzlich eine Seite angefordert werden, da seine Knoten auf einer neuen Seite beginnend abgespeichert werden (Abbildung 5.1). Exemplarisch werden hier vier Knoten auf zwei Prozessoren angelegt. Der erste Aufruf der Funktion ndInsert() führt dazu, daß Prozessor 1 eine Speicherseite vom Betriebssystem anfordert und den ersten Knoten darauf unterbringt. Der darauf folgende ndInsert()-Aufruf des Prozessors 2 führt ebenfalls dazu, daß eine Seite angefordert wird. Dieser Knoten wird also nicht auf der von P_1 angeforderten Seite abgespeichert. Die restlichen beiden Knoten werden auf den bereits angelegten Seiten abgespeichert.

Auch die anderen Datenstrukturen werden so abgespeichert, daß bei einem Zugriff kein *false sharing* entsteht. Dies ist sichergestellt, da die dynamische Speicherverwaltung des SVM-Fortran-Compilers so ausgelegt ist, daß ein Prozessor jeweils eine ganze Seite vom Betriebssystem anfordert, die stückweise an das Anwendungsprogramm vergeben wird. Da außerdem der Prozessor, der einen Knoten angelegt hat, auch für den Zugriff auf die zugehörigen Verbindungen und die Attribute zuständig ist, ist sichergestellt,

daß auf diese Daten nur von einem Prozessor zugegriffen wird. Die Speicherverwaltung ist im Kapitel 5.6 näher erläutert.

Die Hilfsfunktionen für die Verwaltung unstrukturierter Templates wurden in C programmiert, sie werden aber aus einem FORTRAN-Programm aufgerufen. Deshalb sind hier die Aufrufe als FORTRAN-CALLS dargestellt.

5.1 Datenstrukturen

In Abbildung 5.2 ist die Datenstruktur in der Übersicht dargestellt. Jeder Block entspricht in der Implementation einer Struktur (struct). Zeiger auf einen anderen Block sind durch Pfeile wiedergegeben.

5.1.1 tList: Tabelle der unstrukturierten Templates

tList ist eine Tabelle von Zeigern auf die unstrukturierten Templates. Durch die Konstante NO_T_MAX ist die maximal mögliche Anzahl von unstrukturierten Templates festgelegt.

5.1.2 tType: Template

Das unstrukturierte Template wird durch die Struktur tType beschrieben (Abbildung 5.3), in der alle Template-Informationen festgehalten werden: In tType werden Informationen über das Template selbst, über Verknüpfungen zu anderen Templates und über Knoten und Verbindungen gespeichert.

Informationen über das Template Über das Template müssen limitierende Werte, wie etwa die maximale Anzahl Knoten $maxNds$ und Verbindungen (pro Knoten) $maxEds$, also der maximale Knotengrad, vermerkt werden. Auch die Größe des Speicherplatzes, der für die Knoten- bzw. Verbindungsattribute ($sizeNdAttrs$ bzw. $sizeEdAttrs$) bereitzuhalten ist, muß gespeichert werden. Momentan wird hier als „Einheit“ der Platz für eine Variable des Typs **double** verwendet.

Informationen über verknüpfte Templates In der Tabelle $linkedTs$ werden die Templates vermerkt, zu denen Verknüpfungen aufgebaut werden sollen. In $noLinkedTs$ ist dazu die Anzahl der vorgesehenen Verknüpfungen zu anderen Templates anzugeben. Diese werden mit Hilfe der Funktion tLinkedTsDef() (siehe Kapitel 5.2) eingetragen. $posLinkedTs$ enthält die Anzahl der tatsächlich eingetragenen verknüpften Templates.

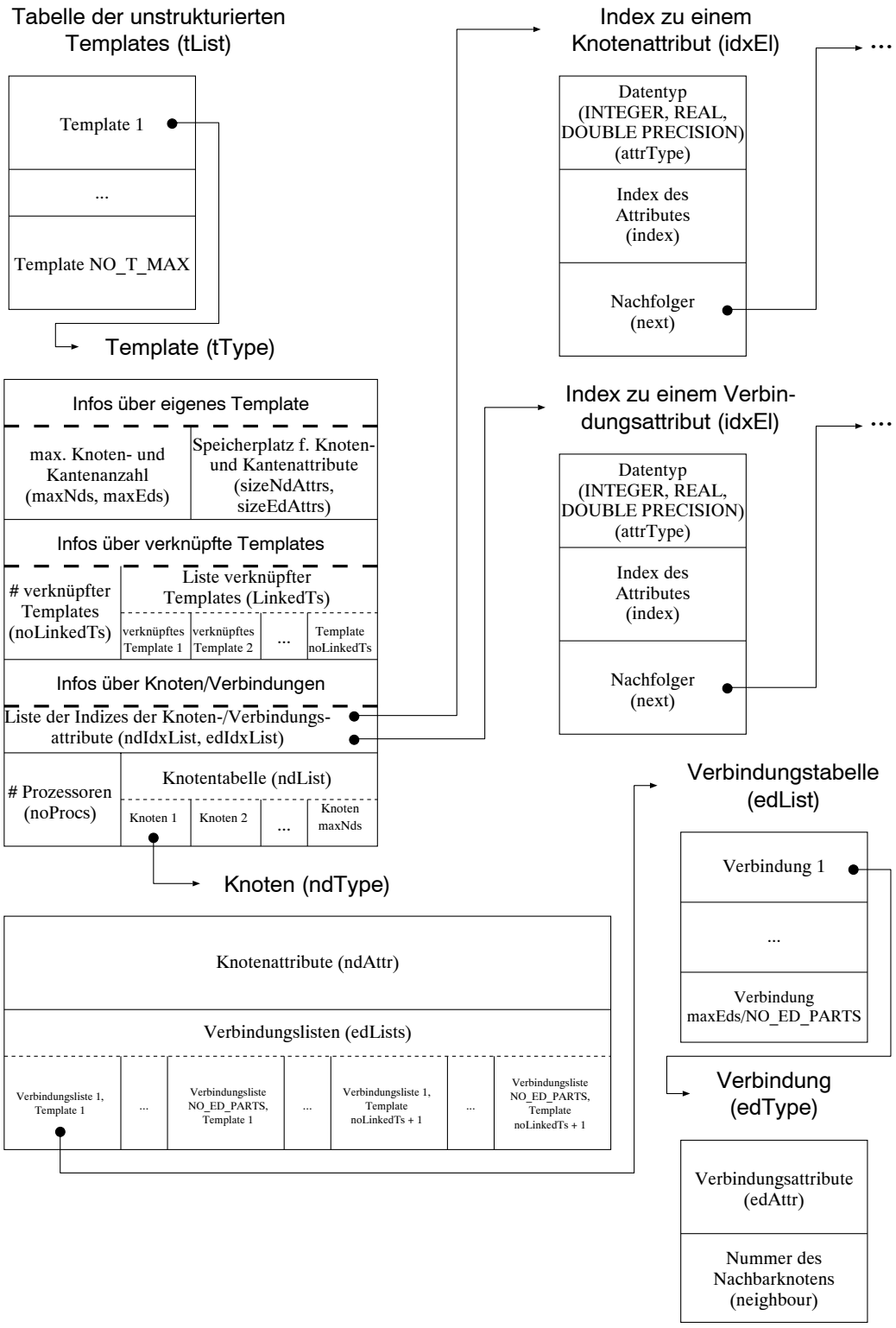


Abbildung 5.2: Datenstruktur der Verwaltung unstrukturierter Templates

```
/* tType: Template */
struct tType
{
        unsigned int maxNds;
        unsigned short maxEds;
        unsigned int sizeNdAttrs;
        unsigned int sizeEdAttrs;

        unsigned short noLinkedTs;
        unsigned short posLinkedTs;
        unsigned int *linkedTs;

        unsigned int noProcs;
        NdType **ndList;
        IdxEl *ndIdxList;

        IdxEl *edIdxList;
};
typedef struct tType TType;
```

Abbildung 5.3: Datenstruktur für ein unstrukturiertes Template

```
/* idxEl: Element einer Indexliste */
struct idxEl
{
        unsigned short attrType;
        unsigned int index;
        IdxEl *next;
};
typedef struct idxEl IdxEl;
```

Abbildung 5.4: Informationen zu einem Knoten- bzw. Verbindungsattribut

Informationen über Knoten und Verbindungen Die Anzahl der Prozessoren der aktiven Prozessormenge wird in $noProcs$ gespeichert. Die Knoten des Template werden in der Tabelle $ndList$ aufgeführt. Sie werden blockweise auf die Prozessoren aufgeteilt. Somit erhält jeder Prozessor etwa $\frac{maxNds}{noProcs}$ Knoten. Prozessor p erhält somit die Knoten $\lceil\frac{maxNds}{noProcs}\rceil \cdot p$ bis $\lceil\frac{maxNds}{noProcs}\rceil \cdot (p+1)$. In den Listen $ndIdxList$ und $edIdxList$ werden die Indizes der Knoten- bzw. Verbindungsattribute gespeichert: Der Speicherplatz für Knoten- und Verbindungsattribute (in den Strukturen `ndType` bzw. `edType`) wird in einem zusammenhängenden Stück angefordert. Zum Zeitpunkt der Allokierung ist die Struktur, d. h. die Reihenfolge der Attribute – und damit die Reihenfolge der zugehörigen Typen (INTEGER, REAL oder DOUBLE PRECISION) – noch nicht bekannt. In der momentanen Version wird allen Attributen, gleich welchen Typs, der Speicherplatz einer DOUBLE-PRECISION-Variable zugewiesen. Ausführlicher wird dieser Aspekt im Kapitel 5.2 behandelt.

5.1.3 idxEl: Information zu einem Knoten- bzw. Verbindungsattribut

Da der Speicherplatz für Attribute als ein Block allokiert wird, muß zum Zugriff auf ein Attribut seine Position in diesem Block – also sein Index – bekannt sein. Die erste Position im Attributblock hat den Index 0, die zweite den Index 1, usw. Diese Positionen werden in einer verketteten Liste gespeichert. Es existieren zwei solche Listen: $ndIdxList$ für die Indizes der Knotenattribute und $edIdxList$ für die Indizes der Verbindungsattribute. In einem Element dieser Liste (Abbildung 5.4) werden neben dem $index$ auch der Typ $attrType$ und ein Zeiger auf den nachfolgenden Eintrag der Liste ($next$) gespeichert. Der Datentyp muß bei einem Zugriff bekannt sein, da sich die Art der Abspeicherung – zum einen in der Größe und zum ande-

```
/* ndType: Knoten */
struct ndType
{
        double *ndAttr;
        EdList **edLists;
}
typedef struct ndType NdType;
```

Abbildung 5.5: Datenstruktur für einen Knoten eines unstrukturierten Template

ren in der Darstellung – bei den einzelnen Typen unterscheidet: INTEGER- und REAL-Werte werden auf der Intel Paragon in vier Bytes gespeichert, DOUBLE-PRECISION-Werte benötigen acht Bytes Speicherplatz. Auch die Darstellung einer Zahl auf Bitebene kann sich bei einer Abspeicherung als INTEGER-, REAL- oder DOUBLE-PRECISION-Wert unterscheiden.

5.1.4 ndType und edType: Knoten und Verbindungen

Durch *ndType* (Abbildung 5.5) und *edType* (Abbildung 5.6) werden die Informationen eines Knotens bzw. einer Verbindung, also die Attribute und die Verbindungen (*ndType*) bzw. der Nachbarknoten (*edType*), beschrieben. Die (bidirektionalen) Verbindungen sind – um Speicherplatz nicht unnötig zu verbrauchen – nicht in beiden Endknoten abgelegt. Sie werden grundsätzlich in dem Knoten mit dem kleineren Index gespeichert.

Die Informationen eines Knotens bestehen aus seinen Attributen *ndAttr* und den Verbindungslisten *edLists*. Wie bereits erwähnt wird auf den Typ der Attribute vorerst keine Rücksicht genommen; für jedes Attribut wird Speicherplatz für eine **double**-Variable reserviert.

In der Regel existieren mehr als eine Verbindungsliste pro Gitterknoten. Auch hier wird wieder eine Strategie zur ökonomischen Nutzung des Speicherplatzes verfolgt. Da, wie bereits weiter oben erwähnt, die Verbindung zweier Knoten nur jeweils in demjenigen Knoten gespeichert wird, der den kleineren Index hat, ist damit zu rechnen, daß diese Knoten viele Verbindungseinträge haben, diejenigen mit einem großen Index wenige oder gar keine. Somit ist es sinnvoll, die Verbindungslisten der Knoten größer oder kleiner zu halten. Dies gänzlich dynamisch zu halten ist problematisch, insbesondere im Hinblick auf adaptive Gitterstrukturen. Denkbar wäre beispielsweise, daß man erst dann Speicher für die Verbindungen anlegt, wenn bekannt ist, wievie-

```
/* edList: Verbindungstabelle */
struct edList
{
      EdType **edPtr;
};
typedef struct edList EdList;

/* edType: Verbindung */
struct edType
{
      double *edAttr;
      unsigned int neighbour;
};
typedef struct edType EdType;
```

Abbildung 5.6: Datenstruktur für Verbindungstabelle und Verbindungsinformationen

le Verbindungen gespeichert werden müssen. Man könnte dann den genau passenden Speicher anfordern und müßte dann auch die Anzahl der Verbindungen protokollieren.

Kommen allerdings in einem adaptiven Schritt weitere Verbindungen dazu, so würde dies bedeuten, daß ein um die weiteren Verbindungen vergrößerter Speicherbereich angelegt, die neuen Verbindungen erzeugt, die alten Verbindungen kopiert und der ursprüngliche Speicherbereich freigegeben werden müßte.

Ebenso widerspricht die gewählte Implementierung der Definition der Verbindungen dem obigen Vorgehen: Eine Verbindung wird vollkommen losgelöst von anderen Verbindungen definiert und eingefügt. Die Größe einer Verbindungsliste genau festzulegen wäre demnach also erst möglich, wenn alle Verbindungen bekannt sind. Ein temporäres Feld mit Platz für alle denkbaren $maxNds \cdot maxEds$ Verbindungen müßte also angelegt werden.

Diese Probleme werden mit dem gewählten und umgesetzten „halbdynamischen“ Ansatz umgangen: Die Verbindungen werden auf eine bestimmte Anzahl von Verbindungslisten, gegeben durch die Konstante `NO_ED_PARTS`, aufgeteilt[1]. Somit erhält man `NO_ED_PARTS` Listen, die jeweils $\lceil \frac{maxEds}{NO\_ED\_PARTS} \rceil$ groß sind. Wird die erste Verbindung eines Knotens definiert, so wird *die erste* dieser Listen erzeugt; weitere Verbindungen werden sukzessive in die-

[1]In der momentanen Implementierung wurde `NO_ED_PARTS` auf zwei festgelegt.

se Liste geschrieben. Die nächste Liste muß erst erzeugt werden, wenn die $(\lceil \frac{maxEds}{NO\_ED\_PARTS} \rceil + 1)$-te Verbindung definiert wird usw.

5.2 Template-Funktionen

5.2.1 irrTInit: Initialisierung der Verwaltung unstrukturierter Templates

Aufruf: CALL irrTInit($result$)

Eingabe: keine

Rückgabe: INTEGER $result$

Mit irrTInit() wird die Tabelle $tList$ der unstrukturierten Templates erzeugt und mit NULL-Zeigern initialisiert. Diese Funktion ist vor allen anderen Funktionen für unstrukturierte Templates aufzurufen.

Der Wert von $result$ ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.2.2 tCreateLinked: Template mit Verknüpfungen erzeugen

Aufruf: CALL tCreateLinked($maxNds, maxEds, sizeNdAttrs,$
$sizeEdAttrs, sizeLinkedTs, tHandle$)

Eingaben: INTEGER $maxNds$
INTEGER $maxEds$
INTEGER $sizeNdAttrs$
INTEGER $sizeEdAttrs$
INTEGER $sizeLinkedTs$

Rückgabe: INTEGER $tHandle$

Mit tCreateLinked() wird ein Template angelegt, d. h. entsprechender Speicherplatz wird angefordert und in der Template-Tabelle $tList$ ein Zeiger auf diese Struktur eingetragen.

Die Maximalwerte für Knoten- bzw. Verbindungsanzahl $maxNds$ und $maxEds$ werden eingetragen. Auch der zu reservierende Platz für Knotenattribute $sizeNdAttrs$ und Verbindungsattribute $sizeNdAttrs$ und die maximale Anzahl von Verbindungen zu anderen Templates $sizeLinkedTs$ werden gesetzt.

Es wird Speicherplatz für die Liste der verknüpften Templates $linkedTs$, ebenso für die Knotenliste $ndList$ und die beiden Indexlisten $ndIdxList$ und $edIdxList$ angelegt und initialisiert.

In $tHandle$ wird ein Handle zurückgegeben, mit dem das Template referenziert werden kann. Im Falle eines Fehlers wird hier -1 zurückgegeben.

Für den Fall, daß ein Template definiert werden soll, das keine Verknüpfungen zu anderen Templates hat, kann die Funktion tCreate() aufgerufen werden. Es wird dann tCreateLinked($maxNds$, $maxEds$, $sizeNdAtrs$, $sizeEdAttrs$, 0, $tHandle$) ausgeführt.

5.2.3 tCreate: Template erzeugen

Aufruf: CALL tCreate($maxNds, maxEds, sizeNdAttrs, sizeEdAttrs,$
$tHandle$)

Eingaben: INTEGER $maxNds$
INTEGER $maxEds$
INTEGER $sizeNdAttrs$
INTEGER $sizeEdAttrs$

Rückgabe: INTEGER $tHandle$

tCreate() arbeitet generell wie tCreateLinked() (ruft dies deshalb auch auf), ist aber nur für Templates gedacht, die keine Verbindungen zu anderen Templates aufbauen müssen. Deshalb wird tCreateLinked($maxNds, maxEds,$ $sizeNdAtrs$, $sizeEdAttrs$, 0, $tHandle$) aufgerufen.

In $tHandle$ wird ein *Handle* zurückgegeben, mit dem das Template referenziert werden kann. Im Falle eines Fehlers wird hier -1 zurückgegeben.

5.2.4 tNdAttrDef: Knotenattribut definieren

Aufruf: CALL tNdAttrDef($tHandle, attrType, idxHandle$)

Eingaben: INTEGER $tHandle$
INTEGER $attrType$

Rückgabe: INTEGER *idxHandle*

Mit tNdAttrDef() wird die Liste der Knotenattributindizes aufgebaut: Es wird ein neuer Eintrag für die Liste erzeugt, der Datentyp (INTEGER, REAL oder DOUBLE PRECISION) und die Indexnummer *(Offset)* im Eintrag vermerkt. Diese Einträge sind intern einfach verkettet.

In *idxHandle* wird ein Handle zurückgegeben, mit dem das Attribut referenziert werden kann. Im Falle eines Fehlers (ungültiger Attributtyp) wird hier -1 zurückgegeben.

5.2.5 tEdAttrDef: Verbindungsattribut definieren

Aufruf: CALL tEdAttrDef(*tHandle*, *attrType*, *idxHandle*)

Eingaben: INTEGER *tHandle*
INTEGER *attrType*

Rückgabe: INTEGER *idxHandle*

Mit tEdAttrDef() wird die Liste der Verbindungsattributindizes aufgebaut: Es wird ein neuer Eintrag für die Liste erzeugt und Datentyp (INTEGER, REAL oder DOUBLE PRECISION) und Indexnummer *(Offset)* im Eintrag vermerkt. Diese Einträge sind intern einfach verkettet.

In *idxHandle* wird ein Handle zurückgegeben, mit dem das Attribut referenziert werden kann. Im Falle eines Fehlers (ungültiger Attributtyp) wird hier -1 zurückgegeben.

5.2.6 tLinkedTsDef: Verknüpfung mit einem Template definieren

Aufruf: CALL tLinkedTsDef(*tHandle*, *linkedTHandle*, *result*)

Eingaben: INTEGER *tHandle*
INTEGER *linkedTHandle*

Rückgabe: INTEGER *result*

tLinkedTsDef() trägt ein Template in die Liste der verknüpften Templates $linkedTs$ ein: Der Handle $linkedTHandle$ wird in der Liste vermerkt.

Der Wert von result ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.2.7 Interne Funktion tShowTList: Template-Zeiger ausgeben

Aufruf: CALL tShowTList()

Eingaben: keine

Rückgabe: keine

tShowTList() gibt für jeden Prozessor (Knoten) die Einträge in der Template-Tabelle $tList$ in Form einer Zeigeradresse auf die Standardausgabe aus. Diese Funktion ist nur für Testzwecke gedacht.

5.2.8 Interne Funktion tShowTNodes: Zuordnung der Gitterknoten zu den Prozessoren

Aufruf: CALL tShowTNodes($tHandle$)

Eingaben: INTEGER $tHandle$

Rückgabe: keine

tShowTNodes() gibt die Zuordnung der Knoten zu den Prozessoren für das Template $tHandle$ als Liste auf die Standardausgabe aus. Diese Funktion ist nur für Testzwecke gedacht.

5.3 Knotenfunktionen

5.3.1 ndInsert: Knoten in Template einfügen

Aufruf: CALL ndInsert($tHandle, ndIdx, result$)

Eingaben: INTEGER $tHandle$

INTEGER *ndIdx*

Rückgabe: INTEGER *result*

Der Knoten wird in das Template *tHandle* an der Stelle *ndIdx* eingefügt, d. h. Speicherplatz für die Knotenattribute und die Zeiger auf die Verbindungslisten des Knotens wird reserviert.

Es ist zu beachten, daß Knoten mit aufeinanderfolgenden Indizes auch nacheinander definiert werden sollten, da nur so gewährleistet ist, daß sie im Speicher auch an benachbarten Positionen abgelegt werden. Dies ist keine Einschränkung, da die Knoten in der Regel in einer Schleife definiert werden, die über die Knotenindizes läuft.

Der Wert von *result* ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.3.2 ndAttrSet: Setzen eines Knotenattributes

Aufruf: CALL ndAttrSet(*tHandle*, *ndIdx*, *attrHandle*, *attrVal*, *result*)

Eingaben: INTEGER *tHandle*
INTEGER *ndIdx*
INTEGER *attrHandle*
DOUBLE PRECISION *attrVal*

Rückgabe: INTEGER *result*

Der Attributwert *attrVal* wird in das Template *tHandle* im *ndIdx*ten Knoten eingefügt. Das zu setzende Attribut wird durch seinen Handle *attrHandle* referenziert.

Der Wert von *result* ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.3.3 ndAttrGet: Auslesen eines Knotenattributes

Aufruf: CALL ndAttrGet(*tHandle*, *ndIdx*, *attrHandle*, *attrVal*, *result*)

Eingaben: INTEGER *tHandle*
INTEGER *ndIdx*
INTEGER *attrHandle*

Rückgabe: DOUBLE PRECISION $attrVal$
INTEGER $result$

Der Attributwert $attrVal$ wird aus dem $ndIdx$ten Knoten des Template $tHandle$ ausgelesen. Das zu lesende Attribut wird durch seinen Handle $attrHandle$ referenziert.

Der Wert von $result$ ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.3.4 Interne Funktion ndExistNode: Test auf Existenz eines Knotens

Aufruf: CALL ndExistNode($tHandle, ndIdx, result$)

Eingaben: INTEGER $tHandle$
INTEGER $ndIdx$

Rückgabe: INTEGER $result$

ndExistNode() überpruft, ob der Knoten $ndIdx$ existiert. Ist dies der Fall, so wird in $result$ die entsprechende Knotennummer zurückgeliefert. Ist dies nicht der Fall, so wird -1 zurückgegeben. Diese Funktion ist nur für Testzwecke gedacht.

5.3.5 Interne Funktion ndGetNeighbour: Auslesen eines Nachbarknotens

Aufruf: CALL ndGetNeighbour($tHandle, ndIdx, i, neighbour$)

Eingaben: INTEGER $tHandle$
INTEGER $ndIdx$
INTEGER i

Rückgabe: INTEGER $neighbour$

ndGetNeighbour() gibt in $neighbour$ den $i-$ten Nachbarn des $ndIdx$ten Knoten des Template $tHandle$ zurück. Existiert dieser nicht, so wird -1 zurückgegeben. Diese Funktion ist nur für Testzwecke gedacht.

5.4 Verbindungsfunktionen

5.4.1 edInsertLinked: Verbindung zwischen verknüpften Templates einfügen

Aufruf: CALL edInsertLinked($tHandle, tHandleLinked, ndIdx, ndIdxLinked, result$)

Eingaben: INTEGER $tHandle$
INTEGER $tHandleLinked$
INTEGER $ndIdx$
INTEGER $ndIdxLinked$

Rückgabe: INTEGER $result$

Die Verbindung zum Knoten $ndIdxLinked$ des verknüpften Template $tHandleLinked$ wird im Knoten $ndIdx$ vermerkt.

Der Wert von $result$ ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

Für den Fall, daß eine Verbindung zwischen zwei Knoten $ndIdx1$ und $ndIdx2$ des selben Template definiert werden soll, kann die Funktion edInsert() aufgerufen werden. Es wird dann edInsertLinked($tHandle$, $tHandle$, $ndIdx1$, $ndIdx2$, $result$) ausgeführt.

5.4.2 edInsert: Verbindung zwischen zwei Knoten einfügen

Aufruf: CALL edInsert($tHandle, ndIdx1, ndIdx2, result$)

Eingaben: INTEGER $tHandle$
INTEGER $ndIdx1$
INTEGER $ndIdx2$

Rückgabe: INTEGER $result$

Die Verbindung wird in dem Knoten mit dem kleineren Index, vermerkt. edInsert() arbeitet prinzipiell wie edInsertLinked() (ruft dies deshalb auch auf), ist nur für die Verbindungen zwischen Knoten *eines* Template gedacht. Deshalb wird edInsertLinked($tHandle, tHandle, ndIdx1, ndIdx2, result$) aufgerufen.

Der Wert von *result* ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.4.3 edDeleteLinked: Verbindung zwischen verknüpften Templates löschen

Aufruf: CALL eddeleteLinked($tHandle, tHandleLinked, ndIdx,$
$ndIdxLinked, result$)

Eingaben: INTEGER $tHandle$
INTEGER $tHandleLinked$
INTEGER $ndIdx$
INTEGER $ndIdxLinked$

Rückgabe: INTEGER $result$

Im Template $tHandle$ wird die Verbindung zum verknüpften Template $tHandleLinked$ mit allen ihren Informationen gelöscht.

Der Wert von *result* ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

Für den Fall, daß eine Verbindung zwischen zwei Knoten $ndIdx1$ und $ndIdx2$ des gleichen Template gelöscht werden soll, kann die Funktion edDelete() aufgerufen werden. Es wird dann edDeleteLinked($tHandle, tHandle, ndIdx1, ndIdx2, result$) ausgeführt.

5.4.4 edDelete: Verbindung zwischen zwei Knoten löschen

Aufruf: CALL edDelete($tHandle, ndIdx1, ndIdx2, result$)

Eingaben: INTEGER $tHandle$
INTEGER $ndIdx1$
INTEGER $ndIdx2$

Rückgabe: INTEGER $result$

Im Template $tHandle$ wird die Verbindung zwischen den beiden Knoten $tHandleLinked$ und $ndIdxLinked$ mit allen ihren Informationen gelöscht.

edDelete() arbeitet prinzipiell wie edDeleteLinked() (ruft dies deshalb auch auf), ist nur für die Verbindungen zwischen Knoten *eines* Template gedacht. Deshalb wird edDeleteLinked(*tHandle, tHandle, ndIdx*1, *ndIdx*2, *result*) aufgerufen.

Der Wert von *result* ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.4.5 edAttrSetLinked: Setzen eines Attributs einer Verbindung zwischen verknüpften Templates

Aufruf: CALL edAttrSetLinked($tHandle, tHandleLinked, ndIdx,$
$ndIdx, Linked, attrHandle, attrVal, result$)

Eingaben: INTEGER *tHandle*
INTEGER *tHandleLinked*
INTEGER *ndIdx*
INTEGER *ndIdxLinked*
INTEGER *attrHandle*
DOUBLE PRECISION *attrVal*

Rückgabe: INTEGER *result*

Der Attributwert *attrVal* wird in das Template *tHandle* in der Verbindung zwischen den Knoten *ndIdx* und *ndIdxLinked* (des verküpften Templates *tHandleLinked*) eingefügt. Das zu setzende Attribut wird durch seinen Handle *attrHandle* referenziert.

Der Wert von *result* ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

Für den Fall, daß ein Attribut einer Verbindung zwischen zwei Knoten *ndIdx*1 und *ndIdx*2 des selben Template gesetzt werden soll, kann die Funktion edAttrSet() aufgerufen werden. Es wird dann edAttrSetLinked(*tHandle, tHandle, ndIdx*1, *ndIdx*2, *attrHandle, attrVal, result*) ausgeführt.

5.4.6 edAttrSet: Setzen eines Attribute einer Verbindung zwischen zwei Knoten

Aufruf: CALL edAttrSet($tHandle, ndIdx1, ndIdx2, attrHandle, attrVal,$
$result$)

Eingaben: INTEGER *tHandle*

INTEGER $ndIdx1$
INTEGER $ndIdx2$
INTEGER $attrHandle$
DOUBLE PRECISION $attrVal$

Rückgabe: INTEGER $result$

Der Attributwert $attrVal$ wird in das Template $tHandle$ in der Verbindung zwischen den Knoten $ndIdx1$ und $ndIdx2$ eingefügt. Das zu setzende Attribut wird durch seinen Handle $attrHandle$ referenziert.

edAttrSet() arbeitet prinzipiell wie edAttrSetLinked() (ruft dies deshalb auch auf), ist nur für die Verbindungen zwischen Knoten *eines* Template gedacht. Deshalb wird edAttrSetLinked($tHandle$, $tHandle$, $ndIdx1$, $ndIdx2$, $attrHandle$, $attrVal$, $result$) aufgerufen.

Der Wert von $result$ ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.4.7 edAttrGetLinked: Auslesen eines Attributs einer Verbindung zwischen verknüpften Templates

Aufruf: CALL edAttrGetLinked($tHandle, tHandleLinked, ndIdx,$
$ndIdxLinked, attrHandle, attrVal, result$)

Eingaben: INTEGER $tHandle$
INTEGER $tHandleLinked$
INTEGER $ndIdx$
INTEGER $ndIdxLinked$
INTEGER $attrHandle$

Rückgabe: DOUBLE PRECISION $attrVal$
INTEGER $result$

Der Attributwert $attrVal$ wird aus der Verbindung zwischen den Knoten $ndIdx$ und $ndIdxLinked$ (des verküpften Templates $tHandleLinked$) des Template $tHandle$ ausgelesen. Das zu lesende Attribut wird durch seinen Handle $attrHandle$ referenziert.

Der Wert von $result$ ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

Für den Fall, daß ein Attribut einer Verbindung zwischen zwei Knoten $ndIdx1$ und $ndIdx2$ des gleichen Template gelesen werden soll, kann die Funktion edAttrGet() aufgerufen werden. Es wird dann edAttrGetLinked($tHandle$, $tHandle$, $ndIdx1$, $ndIdx2$, $attrHandle$, $attrVal$, $result$) ausgeführt.

5.4.8 edAttrGet: Auslesen eines Attributs einer Verbindung zwischen zwei Knoten

Aufruf: CALL edAttr($tHandle, ndIdx1, ndIdx2, attrHandle, attrVal,$)
$result$
Eingaben: INTEGER $tHandle$
INTEGER $ndIdx1$
INTEGER $ndIdx2$
INTEGER $attrHandle$

Rückgabe: DOUBLE PRECISION $attrVal$
INTEGER $result$

Der Attributwert $attrVal$ wird aus der Verbindung zwischen den Knoten $ndIdx1$ und $ndIdx2$ des Template $tHandle$ ausgelesen. Das zu lesende Attribut wird durch seinen Handle $attrHandle$ referenziert.

edAttrGet() arbeitet prinzipiell wie edAttrGetLinked() (ruft dies deshalb auch auf), ist nur für die Verbindungen zwischen Knoten *eines* Template gedacht. Deshalb wird edAttrGetLinked($tHandle$, $tHandle$, $ndIdx1$, $ndIdx2$, $attrHandle$, $attrVal$, $result$) aufgerufen.

Der Wert von $result$ ist 0, wenn kein Fehler aufgetreten ist. Im Falle eines Fehlers wird -1 zurückgegeben.

5.4.9 Interne Funktion edExistEdgeLinked: Test auf Existenz einer Verbindung zwischen zwei Templates

Aufruf: CALL edExistEdgeLinked($tHandle, tHandleLinked, ndIdx,$
$ndIdxLinked, result$)

Eingaben: INTEGER $tHandle$
INTEGER $tHandleLinked$
INTEGER $ndIdx$

INTEGER *ndIdxLinked*

Rückgabe: INTEGER *result*

edExistEdgeLinked() testet, ob die Verbindung zwischen den Knoten *ndIdx* des Template *tHandle* und *ndIdxLinked* des Template *tHandleLinked* bereits existiert.

Der Wert von *result* ist 1, wenn die Verbindung existiert; *result* ist 0, wenn die Verbindung nicht existiert. Im Falle eines Fehlers wird -1 zurückgegeben.

Für den Fall, daß eine Verbindung zwischen zwei Knoten $ndIdx1$ und $ndIdx2$ des gleichen Template überprüft werden soll, kann die Funktion edExistEdge() aufgerufen werden. Es wird dann edExistEdgeLinked(*tHandle*, *tHandle*, $ndIdx1$, $ndIdx2$, *result*) ausgeführt. Diese Funktion ist nur für Testzwecke gedacht.

5.4.10 Interne Funktion edExistEdge: Test auf Existenz einer Verbindung

Aufruf: CALL edExistEdge(tHandle, ndIdx1, ndIdx2, result)

Eingaben: INTEGER *tHandle*
INTEGER $ndIdx1$
INTEGER $ndIdx2$

Rückgabe: INTEGER *result*

edExistEdge() testet, ob die Verbindung zwischen den Knoten $ndIdx1$ und $ndIdx2$ des Template *tHandle* bereits existiert.

edExistEdge() arbeitet prinzipiell wie edExistEdgeLinked() (ruft dies deshalb auch auf), ist nur für die Verbindungen zwischen Knoten *eines* Template gedacht. Deshalb wird edExistEdgeLinked(*tHandle*, *tHandle*, $ndIdx1$, $ndIdx2$, *result*) aufgerufen.

Der Wert von *result* ist 1, wenn die Verbindung existiert; *result* ist 0, wenn die Verbindung nicht existiert. Im Falle eines Fehlers wird -1 zurückgegeben. Diese Funktion ist nur für Testzwecke gedacht.

5.4.11 Interne Funktion edGetEdges: Ausgeben aller Nachbarn eines Knotens

Aufruf: CALL edGetEdges(tHandle, ndIdx, edArray, edArraySize)

Eingaben: INTEGER *tHandle*
INTEGER *ndIdx*

Rückgabe: INTEGER *edArray*()
INTEGER *edArraySize*

edGetEdges() liefert in einem Vektor *edArray*() alle Nachbarn des Knotens *ndIdx* des Templates *tHandle* zurück. In *edArraySize* wird die Anzahl der gefundenen Nachbarn – und damit die Länge des Vektors *edArray* zurückgeliefert. Eine entsprechende Funktion zur Ausgabe der Nachbarn in verknüpften Templates existiert nicht. Diese Funktion ist nur für Testzwecke gedacht.

5.5 Partitionierungsalgorithmen

5.5.1 Die CHAOS-Bibliothek

CHAOS [46], entwickelt an der Universität von Maryland, bietet Unterstützung bei der dynamischen Datenverteilung, wie sie bei unstrukturierten und adaptiven Problemen auf Rechnern mit verteiltem Speicher auftritt.

Neben Funktionen, die es dem Programmierer ermöglichen, Informationen insbesondere über die Verbindungsstruktur der Gitter aus dem Programmcode zu extrahieren – man spricht hier von einem *Inspector/Executor*-Ansatz –, sind auch *parallele* Partitionierungsalgorithmen verfügbar: RCB, RIB und RSB. Die beiden koordinatenorientierten Algorithmen, RCB und RIB, werden auch in Versionen bereitgestellt, die Knotengewichte berücksichtigen. Verwendet man den RSB-Algorithmus, so werden die Verbindungsdaten in einem *Laufzeit-Datengraphen* zusammengefaßt.

5.5.2 RIB3: Rekursive Inertialbisektion in drei Dimensionen

Aufruf: CALL RIB3($tHandle, maparray, offset, ndata, new\_ind\_list$)

Eingaben: INTEGER *tHandle*

INTEGER *maparray*
INTEGER *offset*
INTEGER *ndata*

Rückgabe: INTEGER *ndata*
INTEGER *new_ind_list*

Durch RIB3() wird das Template *tHandle* mittels Rekursiver Inertialbisektion (siehe Kapitel 4.1.2) verteilt. *maparray* ist ein Arbeitsfeld für den Partitionierer. *ndata* enthält beim Aufruf die Anzahl der Template-Elemente, die sich auf dem jeweiligen Prozessor befinden. Diese Zahl kann sich während der Berechnung der Partition ändern und wird zurückgegeben. *new_ind_list* ist die Liste der Indizes, die nach der Partitionierung zu dem jeweiligen Prozessor gehören.

5.5.3 RCB3: Rekursive Koordinatenbisektion in drei Dimensionen

Aufruf: CALL RCB3(*tHandle*, *maparray*, *offset*, *ndata*, *new_ind_list*)

Eingaben: INTEGER *tHandle*
INTEGER *maparray*
INTEGER *offset*
INTEGER *ndata*

Rückgabe: INTEGER *ndata*
INTEGER *new_ind_list*

Durch RCB3() wird das Template *tHandle* mittels Rekursiver Koordinatenbisektion (siehe Kapitel 4.1.1) verteilt. *maparray* ist ein Arbeitsfeld für den Partitionierer. *ndata* enthält beim Aufruf die Anzahl der Template-Elemente, die sich auf dem jeweiligen Prozessor befinden. Diese Zahl kann sich während der Berechnung der Partition ändern und wird zurückgegeben. *new_ind_list* ist die Liste der Indizes, die nach der Partitionierung zu dem jeweiligen Prozessor gehören.

5.6 Dynamische Speicherverwaltung in SVM-Fortran

Wie bereits erwähnt, werden globale gemeinsame Variablen über Zeiger realisiert und somit dynamisch angelegt. Die bisherige dynamische Speicherverwaltung war nur auf diesen Aspekt ausgerichtet. Dynamischer Speicher wird einmal zu Beginn des Programms angefordert und erst am Ende des Programms wieder zurückgegeben. Es reicht dann aus, die Allokierung und Deallokierung von *einem* Prozessor durchführen zu lassen. Auch der vom Laufzeitsystem benötigte Speicher läßt sich auf diese Weise verwalten.

Da die Knoten und Verbindungen der unstrukturierten Templates von allen Prozessoren der Anwendung parallel erzeugt werden sollen, mußte die dynamische Speicherverwaltung dementsprechend abgeändert werden, daß jeder Prozessor Speicher anlegen kann und auch bei der Freigabe nachgeführt wird, welcher Prozessor für den jeweiligen Speicherbereich zuständig ist.

5.6.1 Bisherige Speicherverwaltung

Bei der Diskussion der dynamischen Speicherverwaltung existieren drei Ebenen, die betrachtet werden müssen:

- Betriebssystem,
- Dynamische Speicherverwaltung von SVM-Fortran und
- Anwendungsprogramm.

Das Betriebssystem stellt den Speicher jeweils nur in Einheiten von ganzen Seiten bereit. Die dynamische Speicherverwaltung des SVM-Fortran-Compilers teilt eine Seite bei Bedarf in kleinere Abschnitte auf und vergibt diese sukzessive an das (SVM-Fortran-)Anwendungsprogramm. Ist der von der Anwendung angeforderte Speicher mindestens so groß wie eine halbe Seite, so werden von der Speicherverwaltung *Blöcke* von Speicher, d. h. ein Vielfaches der Seitengröße, geliefert. Wird weniger Speicher angefordert, so wird eine Seite fragmentiert, d. h. sie wird in *Fragmente* zerlegt, deren Größe der nächstgrößeren Zweierpotenz entspricht.

Werden z. B. 513 Bytes angefordert, so wird eine freie Seite in acht Fragmente von 1024 Bytes zerlegt. Der zweite Fall, daß ganze Blöcke angefordert werden, ergibt sich bei folgendem Beispiel: Angefordert seien 17000 Bytes. Man erhält dann 3 Seiten, also 24576 Bytes zurück.

Aufgrund der Unterscheidung von Fragmenten und Blöcken werden auch zwei Listen benötigt, in denen die jeweiligen freien Speichereinheiten vermerkt werden. Wird eine Seite in Fragmente aufgeteilt, so wird das erste an

die Anwendung vergeben. Die restlichen werden in der Liste freier Fragmente aufgeführt. Somit kann bei einer Anforderung eines Fragments zunächst überprüft werden, ob ein passendes bereits vorhanden ist oder ob eine neue Seite vom Betriebssystem angefordert werden muß.

Ganze Blöcke werden in einer doppelt verketteten Liste gespeichert. Nur durch Freigabe von Blöcken oder sukzessives Freigeben von benachbarten Fragmenten können Einträge in diese Liste gelangen. Sie wurde bisher im Prinzip nicht benötigt, da dynamischer Speicher erst durch Beendigung des Programms zurückgegeben wird.

5.6.2 Modifizierungen in der Speicherverwaltung

Die Speicherverwaltung muß in zwei Punkten modifiziert werden:

- Freigabe von Speicher während der Laufzeit soll ermöglicht werden.
- Die Verwaltungsstrukturen sollen im privaten Speicher der Prozessoren gehalten werden.

Der erste Punkt wurde notwendig, da auch adaptive Gitter behandelt werden sollen. Dies bedeutet, daß insbesondere Verbindungen (die im gemeinsamen Speicher abgelegt sind) wieder entfernt werden können.

Die Privatisierung der Verwaltungsstrukturen wird angestrebt, da eine globale Verwaltung unnötige Kommunikation zwischen den Prozessoren erfordern würde. Allerdings wird dadurch die Freigabe von dynamischem Speicher schwieriger, da ein Speicherbereich nur von dem Prozessor freigegeben werden kann, der ihn auch angelegt hat. Nur er hat die notwendigen Informationen in seiner lokalen Struktur.

Das Vorgehen wird am vorigen Beispiel vorgeführt. Prozessor 1 fordert 513 Bytes an (Abbildung 5.7), und Prozessor 2 benötigt 17000 Bytes. In der ersten Reihe ist ein Ausschnitt aus dem gemeinsamen Speicher gezeigt. Prozessor 1 besitzt die erste Seite; sie ist fragmentiert. Prozessor 2 besitzt die darauf folgenden 3 Seiten. Die letzten zwei dargestellten Seiten sind noch nicht belegt.

In der zweiten und dritten Zeile sind die zugehörigen Einträge in den lokalen Tabellen der beiden Prozessoren angeführt. Jeder Eintrag entspricht der Struktur `busy` (Abbildung 5.8). Je nach Art der Seite – fragmentiert oder blockweise vergeben – unterscheiden sich die gespeicherten Informationen: Bei fragmentierten Seiten wird als *type* der Zweierlogarithmus der Fragmentgröße gespeichert. Zusätzlich wird in $nfree$ die Anzahl der noch freien Einträge dieser Seite und in $first$ die Adresse des ersten freien Eintrags gespeichert. Im Beispiel wird als *type* 10 angegeben, da ein Fragment die Größe

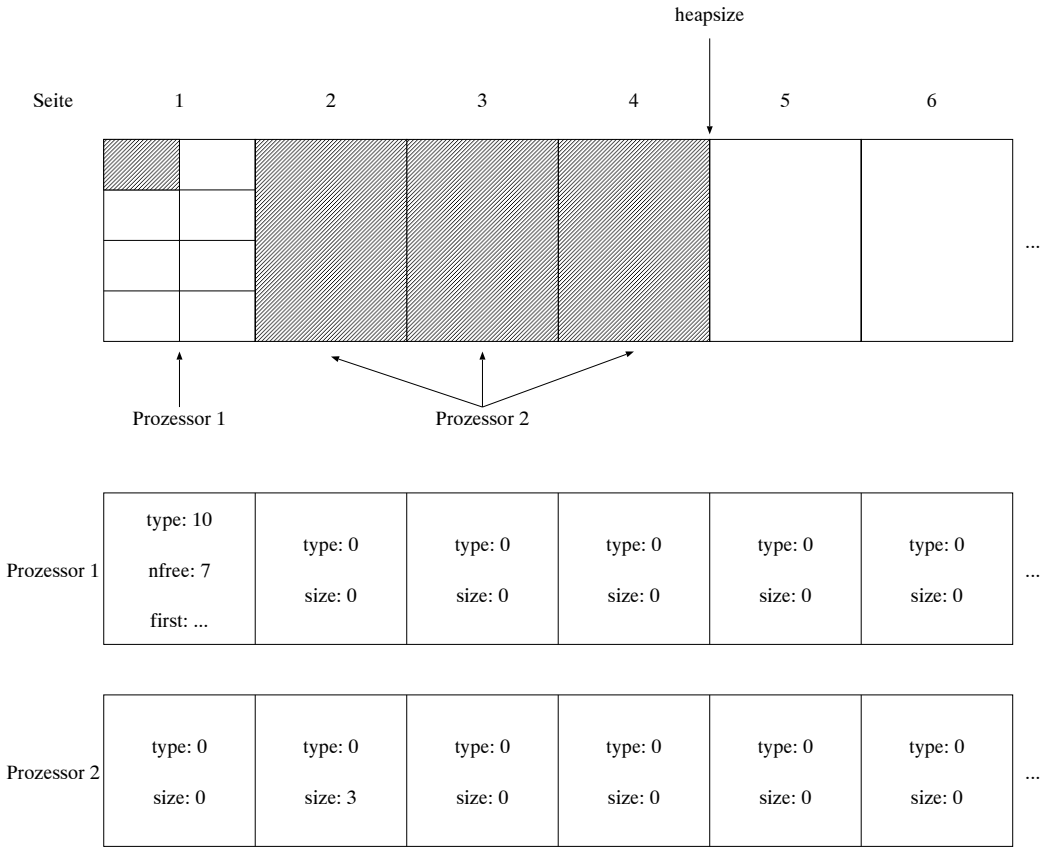


Abbildung 5.7: Seitenverteilung auf zwei Prozessoren

$2^{10} = 1024$ Bytes hat. Ein Fragment wird an die Anwendung vergeben, somit sind weitere sieben Fragmente frei, $nfree$ ist also 7.

Eine Seite, die en bloc vergeben wird, erkennt man daran, daß als *type* 0 angegeben ist. In *size* ist dann die Anzahl der Blöcke gespeichert. Prozessor 2 fordert 17000 Bytes an, erhält also drei Blöcke zugeteilt. In der Tabelle wird für Seite 2 der Typ 0 und in *size* die Größe 3 vermerkt. Die beiden folgenden Einträge, die *in* diesen Block hineinzeigen, dürfen nicht berücksichtigt werden;[2] busy enthält hier die Werte, die bei der Initialisierung eingetragen wurden.

In *heapsize* ist die obere Grenze des vergebenen gemeinsamen Speichers festgehalten. Für Seiten, die hinter dieser Grenze liegen, sind die Werte in *busy* irrelevant. Anders ist dies bei den Seiten 1 und 2. Hier deuten Nullen der Initialisierung in den Einträgen *type* und *size* darauf hin, daß die Seite

[2]Die Speicherverwaltung ist so organisiert, daß auf diese Seiten nicht zugegriffen wird.

```
struct
{
    int type;
    union
    {
        struct
        {
            size_t nfree;
            size_t first;
        } frag;
        size_t size;
    } info;
} busy;
```

Abbildung 5.8: Die Struktur busy

dem jeweils anderen Prozessor gehört. Diese Information wird vor allem bei der Freigabe von Speicher benötigt.

All diese Operationen laufen lokal auf dem jeweiligen Prozessor ab, ausgenommen des Anforderns der Speicherseiten vom Betriebssystem. Dies ist eine globale Operation.

Auch beim Löschen ist diese Zweistufigkeit – nun nicht mehr nur aus Gründen der Leistungssteigerung – notwendig. In Abbildung 5.7 läßt sich dieser Sachverhalt verdeutlichen. Prozessor 1 hat die Informationen für Seite 1, Prozessor 2 die für die Seiten 2, 3 und 4.

Gibt ein Prozessor einen Speicherbereich frei, den er auch angelegt hat, so kann dies direkt geschehen. Speicherbereiche, die von anderen Prozessoren allokiert wurden, erfahren eine Sonderbehandlung. Das Datum wird in einer lokalen Liste zwischengespeichert, bis sie „genügend gefüllt“ ist. Erst dann werden die Daten in eine globale Liste übertragen. Die lokale Liste ist als Stack organisiert, der mit einer initialen Größe angelegt wird. Die Liste gilt als „genügend gefüllt“, wenn sie halbvoll ist. Andere Kriterien sind hier denkbar.

Bevor ein Prozessor Einträge aus seiner lokalen in die globale Liste überträgt, durchsucht er diese nach Einträgen von Speicherbereichen, die von ihm angelegt wurden und gibt sie frei. Ist die globale Liste voll, müssen die Einträge solange in den lokalen Listen zwischengespeichert bleiben, bis ein Prozessor Einträge in der globalen Liste gelöscht hat. Das kann auch dazu führen, daß die initiale Größe der lokalen Stacks überschritten wird und sie somit vergrößert werden müssen. Können zu einem späteren Zeitpunkt die lokalen Informationen wieder in die globale Liste übertragen werden, wird der Stack wieder bis auf seine ursprüngliche Größe verringert.

Die Listen für freie Fragmente und für freie Blöcke sind nun lokal, d. h. löscht ein Prozessor einen Speicherbereich, so kann nur er diesen Speicherplatz erneut verwenden. Liegen etwa Seiten zweier Prozessoren nebeneinander, im Beispiel die Seite 1 des Prozessors 1 und die Seite 2 des Prozessors 2, so konnte man diese in der bisherigen Speicherverwaltung nach einer Freigabe zu einem Block verschmelzen. Dies ist nun nicht mehr der Fall: Prozessor 1 kann nur seine Seite wiederverwenden, und Prozessor 2 kann die Seiten 2, 3 und 4 erneut vergeben. Speicher, der einmal vom Betriebssystem an einen Prozessor vergeben wurde, bleibt also bis zum Programmende im Besitz dieses Prozessors.

Kapitel 6

Einsatz unstrukturierter Templates

Die im vorigen Kapitel eingeführten unstrukturierten Templates werden folgend im AVL-FIRE-Benchmark eingesetzt. Die Gitter *tjunc* und *drall* wurden mit beiden implementierten Partitionierern (Rekursive Koordinatenbisektion und Rekursive Inertialbisektion) zerlegt.

Die Koordinaten der Gitterknoten sind im Programm nicht explizit enthalten. Über die Verbindungen zu den sechs Nachbarknoten wurden aber „Pseudokoordinaten“ vergeben. Geht man von einem bestimmten, regelmäßigen Abstand zwischen den Knoten aus (Abbildung 6.1), so läßt sich ein ungefähres Abbild der Originaltopologie herleiten.

Die Anweisungen zum Erstellen eines unstrukturierten Templates sind in Abbildung 6.2 aufgeführt. Zunächst muß die Verwaltung unstrukturierter Templates mit irrTInit() initialisiert werden. Die Initialisierung muß von jedem Prozessor ausgeführt werden; deshalb die replizierte Region. Das Template iT wird durch tCreate() angelegt. Die Konstante $NINTCF$ gibt die Anzahl der Elemente des jeweiligen Gitters an. Bei dem Datensatz *drall* sind dies 47312 Elemente. In diesem Template werden keine Verbindungen und somit auch keine Verbindungsattribute gespeichert. 3 Knotenattribute, die Raumkoordinaten, sollen abgespeichert werden. Durch die Funktionsaufrufe von tNdAttrDef() werden die drei Handle für die Koordinaten definiert. Die Koordinaten sind DOUBLE-PRECISION-Werte.

Danach werden die benötigten Daten im Template gesammelt. Mit ndInsert() werden die Knoten angelegt. Mit ndAttrSet() werden die Koordinaten eingetragen, die in den Feldern $DXVAL()$, $DYVAL()$ und $DZVAL()$ gespeichert sind.

Im Aufruf des Partitionierers (hier Rekursive Inertialbisektion) ist $IWORK()$ ein lokales Feld, auf dem der Partitionierer arbeiten kann.

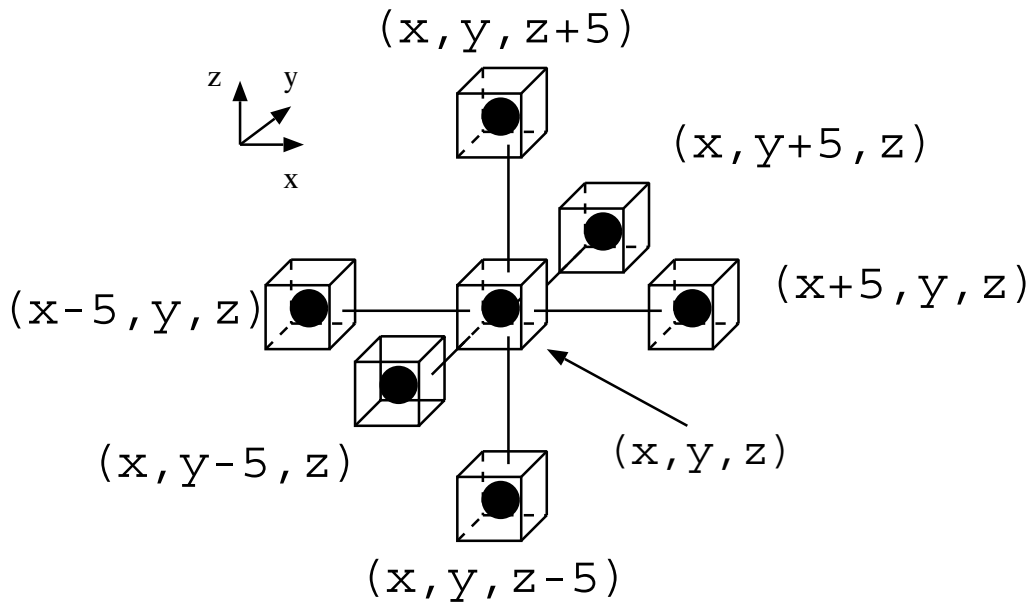


Abbildung 6.1: Abstände der Nachbarknoten

Der Partitionierer bearbeitet die Knoten $ILOWER$ bis $ILOWER + size(MYPROC() + 1)$. Zu Beginn besitzt jeder Prozessor also einen zusammenhängenden Block von Elementen (Abbildung 6.3 (a)). $size()$ ist ein gemeinsamer Vektor, in dem vor dem Unterprogrammaufruf die Anzahl der Elemente pro Prozessor berechnet wurde. Der Partitionierer liefert in $size()$ die Anzahl der Elemente nach dem Partitionierungsschritt zurück. Die privaten Vektoren $localmap()$ enthalten die Indizes der Knoten, die dem jeweiligen Prozessor durch den Partitionierer zugeordnet werden (b). Behalten die Elemente nach der Partitionierung ihren alten Platz in den Feldern des Programms, so sind die Zugriffe jedes Prozessors während der Programmausführung über das gesamte Feld, also über mehrere Seiten, verteilt. Durch eine geeignete Umnumerierung ist dafür zu sorgen, daß beispielsweise für Prozessor 1 ein Zugriff auf die Indizes 1, 3 bzw. 5 so umgelenkt wird, daß er einem Zugriff auf die Indizes 1, 2 bzw. 3 entspricht, die Daten also auf einer Seite gespeichert werden.

Der Programmausschnitt für die Umnumerierung ist in Abbildung 6.4 zu sehen. Die lokalen Indexlisten werden zum gemeinsamen $IMAP()$ zusammengesetzt. Jeder Prozessor bearbeitet in der replizierten Region den durch die Werte $ILOWER$ und $IUPPER$ beschränkten Abschnitt von $IMAP()$. $ILOWER$ und $IUPPER$ bezeichnen den ersten und den letzten Knoten,

```
        INTEGER iT, IX, IY, IZ

C       Initialisierung
CSVM$   REPLICATED_REGION
            CALL irrTInit(IRESLT)
CSVM$   REPLICATED_REGION_END
        CALL tCreate(NINTCF, 0, 3, 0, iT)
        CALL tNdAttrDef(iT, F_DOUBLE, IX)
        CALL tNdAttrDef(iT, F_DOUBLE, IY)
        CALL tNdAttrDef(iT, F_DOUBLE, IZ)

C       Daten im unstrukturierten Template sammeln
CSVM$   PDO(LOOPS(NC), STRATEGY(BLOCK))
        DO NC = NINTCI, NINTCF
            CALL ndInsert(iT, NC, IRESLT)
        END DO

CSVM$   PDO(LOOPS(NC), STRATEGY(BLOCK))
        DO NC = NINTCI, NINTCF
            CALL ndAttrSet(iT, NC, IX, DXVAL(NC), IRESLT)
            CALL ndAttrSet(iT, NC, IY, DYVAL(NC), IRESLT)
            CALL ndAttrSet(iT, NC, IZ, DZVAL(NC), IRESLT)
        END DO
        ...

C       Aufruf des Partitionierers (RIB)
CSVM$   REPLICATED_REGION
            CALL RIB3(iT, IWORK, ILOWER,
     >              size(MYPROC() + 1, localmap)
CSVM$   REPLICATED_REGION_END
```

Abbildung 6.2: Initialisierung des unstrukturierten Template

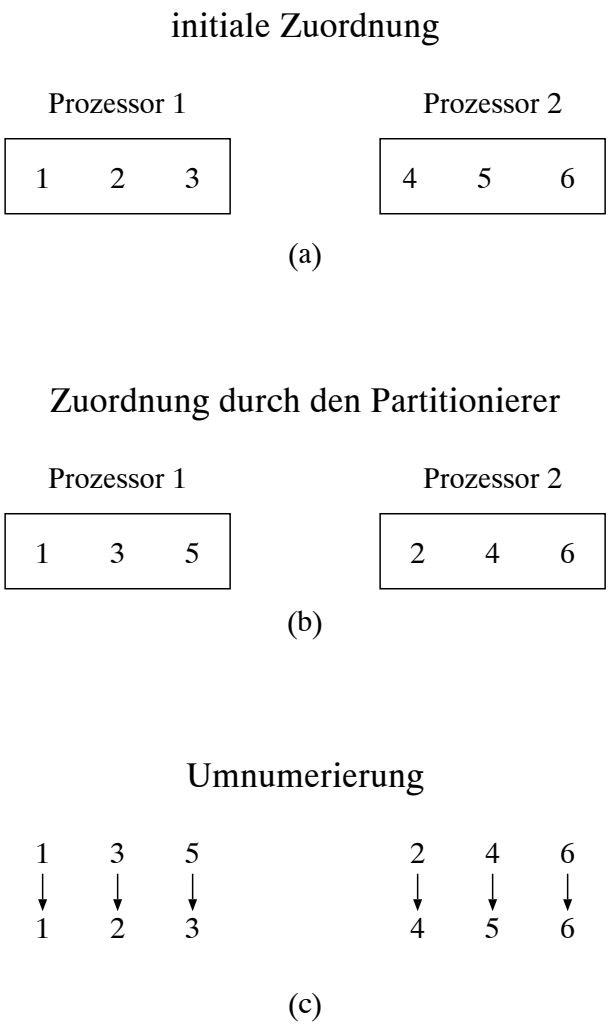


Abbildung 6.3: Verteilung der Elemente auf die Prozessoren

```
          INTEGER iT, IX, IY, IZ

C         IMAP() bilden
CSVM$     REPLICATED_REGION
               DO NC = ILOWER, IUPPER
                    IMAP(NC) = localmap(NC - ILOWER + 1
               END DO
CSVM$     REPLICATED_REGION

C         Umnumerierung
CSVM$     PDO(LOOPS(NC), STRATEGY(BLOCK))
          DO NC = NINTCI, NINTCF
               MAP(IMAP(NC)) = NC
          END DO
```

Abbildung 6.4: Umnumerierung nach dem Partitionierungsschritt

der dem Prozessor zugeteilt wurde. In $MAP()$ werden die neuen Indizes gespeichert.

Die Parallelisierung in dieser Form brachte nicht die erwarteten Ergebnisse (Tabelle 6.1). Bei näherer Betrachtung stellte sich heraus, daß im Berechnungsschritt eine sehr große Anzahl an Lese-Seitenfehlern auftrat. Zugriffe auf die Schnittstellenknoten führen zu vermehrtem Seitenaustausch, da die Knoten über alle Seiten des Teilgitters verteilt sind. Numeriert man die Knoten jedes Prozessors ein weiteres Mal um, so daß in seinem Abschnitt des Vektors $IMAP()$ zuerst die Schnittstellenknoten aufgeführt sind, kann man erwarten, daß weniger Seitenfehler die Folge sein werden.

Beim AVL-FIRE-Benchmark führt eine Blockverteilung schon zu guten Ergebnissen, da das Gitter den Prozessoren quasi scheibenweise zugeteilt wird und die Schnitte quer zur Höhenachse verlaufen. Fällt dieser Schnitt im unteren, großen Block noch relativ großflächig aus, so sind diese Schnitte im oberen Teil des Gitters schon verhältnismäßig klein, und alle Schnittstellenknoten liegen bereits auf einer Seite.

Prozessoranzahl	1	2	4	8	16
Datensatz *tjunc* mit RCB					
Laufzeit [s]	48,78	59,59	79,67	161,23	1636,70
Rechenleistung [MFlops]	2,97	2,43	1,82	0,90	0,09
Datensatz *tjunc* mit RIB					
Laufzeit [s]	48,79	59,66	76,24	203,41	1252,34
Rechenleistung [MFlops]	2,97	2,43	1,90	0,71	0,12
Datensatz *drall* mit RCB					
Laufzeit [s]	164,08	167,09	157,48	150,07	271,35
Rechenleistung [MFlops]	2,99	2,94	3,12	3,27	1,81
Datensatz *drall* mit RIB					
Laufzeit [s]	164,06	166,29	158,58	153,06	236,14
Rechenleistung [MFlops]	2,99	2,95	3,10	3,21	2,08

Tabelle 6.1: Erste Ergebnisse für den AVL-FIRE-Benchmark mit unstrukturierten Gittern

Kapitel 7

Schlußbemerkungen

In der vorliegenden Arbeit wurde gezeigt, daß die Erweiterung des SVM-Fortran-Compilers um unstrukturierte Templates eine einfachere Bearbeitung einer wichtigen Klasse von Anwendungen erlaubt, die auf unstrukturierten Gittern dargestellt werden. Auch für die Klasse der adaptiven Probleme wurde damit die Grundlage zur Behandlung in SVM-Fortran gelegt.

Im Zuge der Implementierung wurde die Speicherverwaltung des SVM-Fortran-Compilers modifiziert: Nun ist es möglich, mit jedem Prozessor Speicher anzufordern. Dadurch wird das Verwalten des Speichers auf die Prozessoren verteilt. Sowohl das Allokieren als auch das Deallokieren von Speicher besteht aus Effizienzgründen aus einer Stufe mit globalem Verhalten und einer zweiten, rein lokalen Stufe.

Außer zur Spezifizierung der Arbeitsverteilung dienen die implementierten unstrukturierten Templates als Sammelstelle für die Geometrie- und Verbindungsinformationen der zu partitionierenden Gitter. Sind diese Informationen einmal vorhanden, muß sich der Anwender um deren Bereitstellung für Partitionierer und für die Visualisierung von Leistungsdaten keine Gedanken mehr machen. Zur Partitionierung werden die parallelen Algorithmen Rekursive Inertialbisektion (RIB) und Rekursive Koordinatenbisektion (RCB) der CHAOS-Bibliothek eingebunden. Die Verfahren liefern Listen lokaler Knoten. Eine Berücksichtigung von Seitengrenzen findet nicht statt.

Die prizipielle Anwendbarkeit der unstrukturierten Templates wurde anhand des AVL-FIRE-Benchmarks demonstriert. Der Benchmark wurde in Kapitel 3 zuerst mit den herkömmlichen regulären Templates parallelisiert und dann der Parallelisierung mittels unstrukturierter Templates (Kapitel 6) gegenübergestellt. Es wurden lediglich die geometriebasierten Partitionierungsverfahren RCB und RIB eingesetzt.

Da in der Parallelisierung mit unstrukturierten Templates die Seitengrenzen und die Trennung der Schnittstellenknoten von den restlichen Knoten ei-

nes Teilgitters vorerst nicht berücksichtigt werden, sind die Laufzeiten nicht vergleichbar mit der Parallelisierung mittels regulärer Templates. Neben der Separierung der Schnittstellenknoten wäre ein weiterer Optimierungsschritt die Ausrichtung der Knoten der Teilgitter auf Seitengrenzen. Scheiniterationen werden in das Template eingefügt. Durch eine komplexe Transformation der Schleife wird dafür gesorgt, daß diese Interationen nicht ausgeführt werden.

Der Einsatz des parallelen RSB-Algorithmus aus der CHAOS-Bibliothek würde zu besseren Ergebnissen führen, da die Verbindungsinformation in ihn einfließt und somit eine Partition mit kleineren Schnittstellen zu erwarten ist. Die Integration dieses Algorithmus wurde bereits begonnen.

Die Direktiven für unstrukturierte Templates können durch die fertiggestellten Funktionen realisiert werden. Die Integration in den Compiler steht noch aus. Momentan muß der Anwender die lokalen Listen der zugeordneten Knoten zu einer globalen Verteilung zusammenfügen. Zukünftig soll dies in der entsprechenden Bibliotheksfunktion erfolgen. Weiterhin können in der Partitionierungsfunktion die Knoten so umnumeriert werden, daß die Schnittstellenknoten aufeinanderfolgende Indizes erhalten und somit nicht über die Seiten des Prozessors verteilt sind, sondern in aufeinanderfolgenden Seiten liegen. Dies hat den Vorteil, daß beim Referenzieren eines Knotens durch einen benachbarten Prozessor direkt die ganze Seite – und damit auch ein großer Teil der von ihm benötigten Knotenmenge – kopiert wird.

Um auch die erwähnten adaptiven Verfahren effizient ausführen zu können, sollten die im Kapitel 4.5 vorgestellten Verfahren genauer betrachtet und implementiert werden. Die PARTY-Bibliothek [45] kann wertvolle Hilfe leisten, da sie ähnlich der CHAOS-Bibliothek Partitionierer und Optimierungsalgorithmen zur Laufzeiteinbindung in Programme bereitstellt.

Zu Visualisierungszwecken wäre es wünschenswert, über eine Instrumentierung des Programmes, ähnlich der Anweisungen zur Leistungsanalyse, die Partitionierungsdaten in einer Datei in einem geeigneten Format abzuspeichern. Ein Visualisierungswerkzeug könnte dann direkt angesteuert werden, ohne daß sich der Anwender Gedanken über das Vorgehen machen müßte. Ein Beispiel ist TOP/DOMDEC [47], ein externer Partitionierer, der auch vielfältige Möglichkeiten zur Visualisierung von Partitionen bietet. Dieses Werkzeug zeichnet sich besonders dadurch aus, daß Partitionen aus Dateien eingelesen werden können. Somit läßt sich TOP/DOMDEC auch als reines Visualisierungswerkzeug nutzen.

Literaturverzeichnis

[1] M. Al-Nasra und D. T. Nguyen. An Algorithm for Domain Decomposition in Finite Element Analysis. In Computers & Structures, 39(3/4):277-289, 1991.

[2] Alfred Arnold. PARvis: Eine X-basierte Umgebung zur Visualisierung von parallelen Programmen in Multiprozessorsystemen. Technischer Bericht Jül-2848, Forschungszentrum Jülich, 1993.

[3] Stephen T. Barnard und Horst D. Simon. A Parallel Implementation of Multilevel Recursive Spectral Bisection for Application to Adaptive Unstructured Meshes. In David H. Bailey et al. (Hrsg.), Proceedings of the Seventh Conference on Parallel Processing for Scientific Computing. SIAM, S. 627 - 632, 1995.

[4] Stephen T. Barnard und Horst D. Simon. Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. In Concurrency: Practice and Experience, 6(2):101-117, 1994.

[5] Marsha J. Berger und Shahid H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. In Transactions on Computers. IEEE, C-36(5):570-580, 1987.

[6] Jost Bernert. Neue PARvis-Komponenten zur effizienten Performance-Analyse von SVM-Fortran-Programmen. Technischer Bericht Jül-3158, Forschungszentrum Jülich, 1995.

[7] Rudolf Berrendorf und Michael Gerndt. Compiling Data Parallel Languages for Shared Virtual Memory Systems. Technischer Bericht KFA-ZAM-IB-9517, Forschungszentrum Jülich, 1995.

[8] Rudolf Berrendorf, Michael Gerndt und Martin Mairandres. Programming Shared Virtual Memory on the Intel Paragon Supercomputer. Technischer Bericht KFA-ZAM-IB-9509, Forschungszentrum Jülich, 1995.

[9] Peter Brezany und Viera Sipkova. Coupling Parallel Data and Work Partitioners to VFCS. In Proceedings of HPC Challenges, 1996.

[10] Peter Brezany, Viera Sipkova, Barbara Chapman und Robert Greimel. Automatic Parallelization of the AVL FIRE Benchmark for a Distributed-Memory System. In Proceedings of HPCN, 1995.

[11] W. M. Chan und Alan George. A Linear Time Implementation of the Reverse Cuthill-McKee Algorithm. In BIT, 20(1):8-14, 1980.

[12] F. Darema, D. A. George, V. A. Norton und G. F. Pfister. A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. In Parallel Computing, 7(1):11-24, 1988.

[13] Ralf Diekmann, Derk Meyer und Burkhard Monien. Parallel Decomposition of Unstructured FEM-Meshes. In Proceedings of IRREGULAR, Lecture Notes in Computer Science, Bd. 980, S. 199 - 315, 1995.

[14] Ralf Diekmann, Burkhard Monien und Robert Preis. Using Helpful Sets to Improve Graph Bisections. Technischer Bericht tr-rf-94-008, Universität Paderborn, 1994.

[15] Charbel Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. In Computers & Structures, 28(5):579-602, 1988.

[16] Charbel Farhat, Stephane Lanteri und Horst D. Simon. TOP/DOMDEC – a Software Tool for Mesh Partitioning and Parallel Processing. Technischer Bericht RNR-93-011, NASA, 1993.

[17] Charbel Farhat und Michel Lesoinne. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. In International Journal for Numerical Methods in Engineering, 36(5):745-764, 1993.

[18] J. Flower, S. Otto und M. Salama. Optimal Mapping of Irregular Finite Element Domains to Parallel Processors. In Ahmed K. Noor (Hrsg.), Proceedings of Parallel Computation and Their Impact on Mechanics, S. 239 - 250, 1987.

[19] Geoffrey C. Fox und W. Furmanski. Load Balancing Loosely Synchronous Problems with a Neural Network. In Geoffrey C. Fox (Hrsg.), Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications. Volume I. Architecture, Software, Computer Systems and General Issues, S. 241 - 278, 1988.

[20] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon und David W. Walker. Solving Problems on Concurrent Processors. Volume I. General Techniques and Regular Problems. Prentice Hall, 1988.

[21] Michael R. Garey und David S. Johnson. Computers and Intractability. A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, 1979.

[22] Michael Gerndt. Parallelization of the AVL FIRE Benchmark with SVM-Fortran. Technischer Bericht KFA-ZAM-IB-9520, Forschungszentrum Jülich, 1995.

[23] Michael Gerndt und Andreas Krumme. Automatic Performance Analysis for Shared Virtual Memory Systems. Technischer Bericht KFA-ZAM-IB-9631, Forschungszentrum Jülich, 1996.

[24] Michael Gerndt, Andreas Krumme und Selçuk N. Özmen. Performance Analysis for SVM-Fortran with OPAL. Technischer Bericht KFA-ZAM-IB-9519, Forschungszentrum Jülich, 1995.

[25] David E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.

[26] Dieter Hänel, Cornelius Thill, Ulrich Uphoff und Roland Vilsmeier. Adaptive Grid Methods in Computational Fluid Dynamics. In Wolfgang E. Nagel (Hrsg.), Partielle Differentialgleichungen, Numerik und Anwendungen. Konferenzen des Forschungszentrums Jülich. Bd. 18/1996, Forschungszentrum Jülich, 1996.

[27] John J. Hopfield und David W. Tank. Computing with Neural Circuits: a Model. In Science, 233:625-639, 1986.

[28] Kai Hwang. Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw-Hill, 1993.

[29] George Karypis und Vipin Kumar. Parallel Multilevel Graph Partitioning. Technischer Bericht 95-036, University of Minnesota, 1995.

[30] B. W. Kernighan und S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. In The Bell System Technical Journal, 49(2):291-307, 1970.

[31] S. Kirkpatrick, C. D. Gelatt, Jr. und M. P. Vecchi. Optimization by Simulated Annealing. In Science, 220:671-680, 1983.

[32] Martin Mairandres. Virtuell gemeinsamer Speicher mit integrierter Laufzeitbeobachtung. Technischer Bericht Jül-3199, Forschungszentrum Jülich, 1996.

[33] James G. Malone. Automated Mesh Decomposition and Concurrent Finite Element Analysis for Hypercube Multiprocessor Computers. In Computer Methods in Applied Mechanics and Engineering, 70(1):27-58, 1988.

[34] Nashat Mansour. Physical Optimization Algorithms for Mapping Data to Distributed-Memory Multiprocessors. Doktorarbeit, Syracuse University, 1992.

[35] Nashat Mansour und Geoffrey C. Fox. A Hybrid Genetic Algorithm for Task Allocation. In Richard K. Belew et al. (Hrsg.), Proceedings of the Fourth International Conference on Genetic Algorithms, S. 466 - 473, 1991.

[36] Kevin McManus. A Strategy for Mapping Unstructured Mesh Computational Mechanics Programs onto Distributed Memory Parallel Architectures. Doktorarbeit, University of Greenwich, 1996.

[37] Burkhard Monien, Ralf Diekmann und Robert Preis. Lastverteilungsverfahren für Parallelrechner mit verteiltem Speicher. In Wolfgang E. Nagel (Hrsg.), Partielle Differentialgleichungen, Numerik und Anwendungen. Konferenzen des Forschungszentrums Jülich. Bd. 18/1996, Forschungszentrum Jülich, 1996.

[38] Bahram Nour-Omid, Arthur Raefsky und Gregory A. Lyzenga. Solving Finite Element Equations on Concurrent Computers. In Ahmed K. Noor (Hrsg.), Proceedings of Parallel Computations and Their Impact on Mechanics, S. 209 - 228, 1987.

[39] Henning Otto. Entwicklung und Analyse von dynamischen Loop-Scheduling-Algorithmen für SVM-Fortran. Technischer Bericht Jül-3156, Forschungszentrum Jülich, 1995.

[40] Chao-Wei Ou und Sanjay Ranka. Parallel Incremental Graph Partitioning. Technischer Bericht SCCS-652, Syracuse University, 1994.

[41] Chao-Wei Ou, Sanjay Ranka und Geoffrey C. Fox. Fast and Parallel Mapping Algorithms for Irregular Problems. Technischer Bericht SCCS-729, Syracuse University, 1993.

[42] Selçuk N. Özmen. SAM: Performance-Analyse-Monitor für SVM-Fortran. Technischer Bericht Jül-3116, Forschungszentrum Jülich, 1996.

[43] Ravi Ponnusamy. Runtime and Compilation Methods for Irregular Computations on Distributed Memory Multiprocessors. Doktorarbeit, Syracuse University, 1994.

[44] Alex Pothen, Horst D. Simon und Kang-Pu Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. In Journal on Matrix Analysis and Applications. SIAM, 11(3):430-452, 1990.

[45] Robert Preis und Ralf Diekmann. The PARTY Partitioning – Library, User Guide, Version 1.1. Technischer Bericht, Universität Paderborn, 1996.

[46] Joel Saltz, Ravi Ponnusamy, Shamik D. Sharma, Bongki Moon, Yuan-Shin Hwang, Mustafa Uysal und Raja Das. A Manual for the CHAOS Runtime Library. Technischer Bericht UMIACS CS-TR-3437, University of Maryland, 1995.

[47] Michael Sharp und Charbel Farhat. TOP/DOMDEC. A Totally Object Oriented Program for Visualisation, Domain Decomposition and Parallel Processing. User's Manual. Technischer Bericht, University of Colorado, 1994.

[48] Horst D. Simon. Partitioning of Unstructured Problems for Parallel Processing. Technischer Bericht RNR-91-008, NASA, 1994.

[49] Horst D. Simon und Shang-Hua Teng. How Good is Recursive Bisection. In Journal on Scientific Computing. SIAM, 1995.

[50] Karen A. Tomko und Santosh G. Abraham. Data and Program Restructuring of Irregular Applications for Cache-Coherent Multiprocessors. In Proceedings of ICS, S. 214 - 225, 1994.

[51] Denis Vanderstraeten, Charbel Farhat, P. S. Chen, Roland Keunings, und O. Zone. A Retrofit Based Methodology for the Fast Generation and Optimization of Large-Scale Mesh Partitions: Beyond the Minimum Interface Size Criterion. Technischer Bericht 94.62, Université Cahtolique de Louvain, 1994.

[52] Denis Vanderstraeten und Roland Keunings. Optimized Partitioning of Unstructured Finite Element Meshes. Technischer Bericht 93.32, Université Catholique de Louvain, 1993.

[53] David Vinckier, Budi Saddak und Achim Basermann. Entwicklung einer maschinenunabhängigen Parallelversion eines nichtlinearen FE-Codes mit Kontaktalgorithmus (CONDAT-DYNA3D). Technischer Bericht KFA-ZAM-IB-9524, Forschungszentrum Jülich, 1995.

[54] P. K. W. Vinsome. Orthomin, an Iterative Method for Solving Sparse Sets of Simultaneous Linear Equations. Paper Number SPE 5729, Society of Petroleum Engineers of AIME, S. 149 - 159, 1986.

[55] Chris Walshaw und Martin Berzins. Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes. Technischer Bericht, University of Leeds, 1993.

[56] Chris Walshaw, M. Cross und M. G. Everett. A Parallelisable Algorithm for Optimising Unstructured Mesh Partitions. Technischer Bericht, University of Greenwich, 1995.

[57] Darrel Whitley. A Genetic Algorithm Tutorial. Technischer Bericht 93-103, Colorado State University, 1993.

[58] Roy D. Williams. Unification of Spectral and Inertial Bisection. Technischer Bericht, California Institute of Technology, 1994.

[59] Roy D. Williams. Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations. Technischer Bericht C3P913, California Institute of Technology, 1990.

[60] Yang Zeng und Santosh G. Abraham. Partitioning Regular Grid Applications with Irregular Boundaries for Cache-Coherent Multiprocessors. In Proceedings of the Ninth International Parallel Processing Symposium, 1995.

Danksagung

Die vorliegende Arbeit wurde als Diplomarbeit in Informatik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Fakultät für Elektrotechnik der Rheinisch-Westfälischen Technischen Hochschule Aachen angefertigt.

Dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich (KFA) anfertigen zu können. Die hervorragende Infrastruktur hat nicht unwesentlich zum Gelingen der Arbeit beigetragen. Herrn Prof. Dr. W. Oberschelp vom Lehrstuhl für Angewandte Mathematik insbesondere Informatik der RWTH Aachen bin ich für die Übernahme des Korreferates dankbar.

Mein außerordentlicher Dank gilt Herrn Dr. M. Gerndt für die intensive Betreuung der Arbeit. Bei Fragen und Problemen hat er stets Zeit für ein ausführliches Gespräch gefunden. Man weiß dies erst dann richtig zu schätzen, wenn man sieht, daß dies nicht immer der Fall ist.

Ein großes „Dankeschön“ auch an M. „Garey & Johnson“ Bücker, der mir nicht nur in Sachen NP-Vollständigkeit zu besonnener Formulierung riet, sondern bei Fragen – insbesondere der Theoretischen Informatik – scheinbar immer Zeit zur Diskussion hatte (und das, obwohl zwischendurch sicherlich auch an seiner Promotion zu arbeiten war). Wichtig war auch, daß er als „Mutter der Diplomanden“ für ein gutes Klima unter dem „Nachwuchs“ des Instituts sorgte.

Herr H. Bast von der Firma Intel war stets dazu bereit, Fragen hinsichtlich der Intel Paragon und des ASVM ausführlich und geduldig zu beantworten. Herr R. Berrendorf stillte meinen Wissensdurst in bezug auf die Speicherverwaltung des SVM-Fortran-Compilers. Beiden möchte ich hiermit danken.

Auch Herrn G. „Chefkoch“ Hüpperling sei gedankt, der durch seine Hilfsbereitschaft drucktechnische Probleme erst gar nicht aufkommen lies. Durch seine rheinische, freundliche Art hellte er manch tristen (Sams-)Tag auf.

Vielen Dank auch an T. „Duden '67“ Richert, der mich davon überzeugte, daß ein Einschub nicht generell durch Gedankenstriche kenntlich gemacht

werden muß. Man kann durchaus auch einmal Kommas verwenden. Außerdem weiß ich jetzt, daß ein Semikolon gelegentlich auch durch einen Punkt ersetzt werden kann.

Fachliche Gespräche und entsprechende Unterstützung sind sicherlich wichtig für eine solche Arbeit, doch ohne die Unterstützung im privaten Bereich hätte ich das Studium der Informatik und insbesondere diese Diplomarbeit nicht bewältigen können. Deshalb möchte ich die Gelegenheit nutzen und an dieser Stelle auch meiner Freundin danken, die besonders in der letzten Zeit zurückstecken mußte (leider wird das wohl auch für die folgenden Monate gelten). Auch mein Bruder hat mich stets unterstützt, dies meist dadurch, daß er mir vergegenwärtigte, daß neben dem Lernen das Leben nicht vergessen werden sollte.

Am wichtigsten aber war sicherlich die Unterstützung meiner Eltern, die mich immer in meinen das Studium betreffenden Bestrebungen unterstützt haben, auch in der Phase des eher divergenten Verlaufs meines Grundstudiums. Ich bin dankbar, daß sie mir diese Ausbildung ermöglichten.