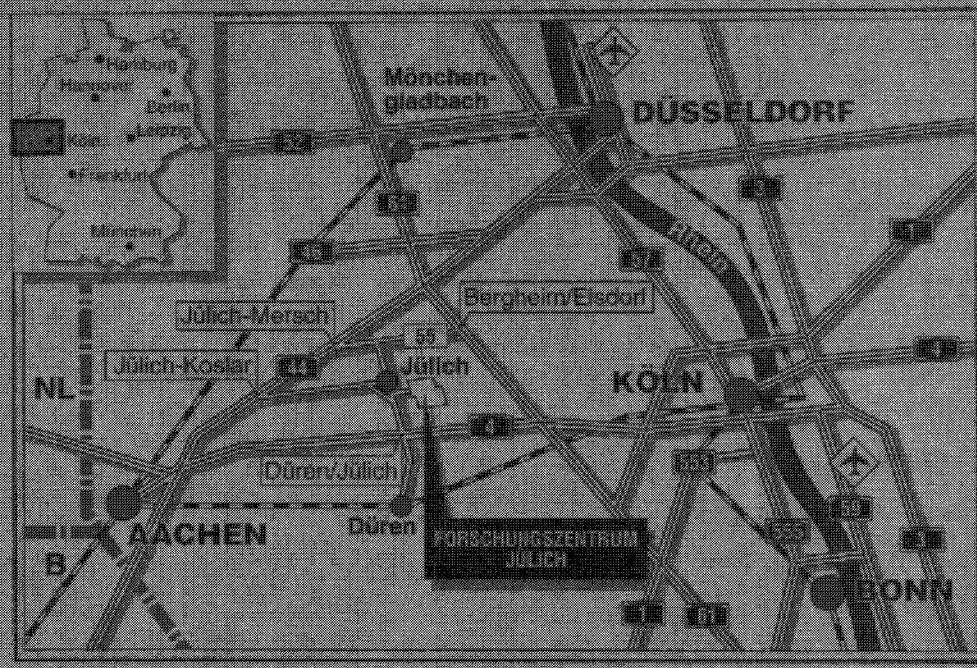
FORSCHUNGSZENTRUM JÜLICH GMBH
KFA

Zentralinstitut für Angewandte Mathematik

Entwicklung und Analyse von dynamischen Loop-Scheduling-Algorithmen für SVM-Fortran

Henning Otto



Berichte des Forschungszentrums Jülich ; 3156
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-3156

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Entwicklung und Analyse von dynamischen Loop-Scheduling-Algorithmen für SVM-Fortran

Henning Otto

Kurzfassung

Shared Virtual Memory (SVM) stellt auf Parallelrechnern mit physikalisch verteiltem Speicher einen logisch zusammenhängenden Adreßraum zur Verfügung. Die Programmiersprache SVM–Fortran erleichtert die effiziente Programmierung von Rechnern, die SVM nutzen. Entscheidend für die Effizienz ist dabei ein hoher Anteil lokaler Datenzugriffe, welcher dadurch erreicht wird, daß der Benutzer statisch die Verteilung von Arbeitspaketen auf Prozessoren vorgibt.

Bei Simulationsprogrammen für komplexe nichtlineare Systeme führt aber eine statische Zuordnung zu einer schlechten Lastverteilung, da sich aufgrund der Systemdynamik Bereiche hoher Rechenintensität kontinuierlich verlagern.

Inhalt dieser Arbeit ist deshalb die Realisierung von dynamischen Loop–Scheduling–Algorithmen für SVM–Fortran. Zunächst wird die bei Parallelrechnern mit physikalisch gemeinsamem Speicher bewährte Methode des Self–Service–Scheduling so angepaßt, daß sie auch auf massiv–parallelen Rechnern mit relativ langsamer Interprozessorkommunikation effizient ablaufen kann. Anschließend wird mit dem Aligned Scheduling ein Verfahren umgesetzt, das die für SVM–Systeme charakteristische Verwaltung von Daten in Seiteneinheiten geeignet berücksichtigt. Als Synthese dieser beiden Verfahren ergibt sich schließlich das Aligned–Self–Service–Scheduling. Dieses gestattet es, die Forderung nach Lokalität beim Datenzugriff mit der nach einer ausgeglichenen Lastverteilung in Einklang zu bringen.

Abstract

Shared Virtual Memory (SVM) provides a global address space on parallel computers with physical distributed memory. The programming language SVM–Fortran was designed to ease efficient programming of these SVM–systems. It turns out that a local data access is the most important factor determining efficiency. In SVM-Fortran, this localtity can be achieved by a static schedule of work–units to the available processors.

Nevertheless, a static work distribution in programs simulating complex nonlinear systems would result in a nonbalanced workload, because the dynamic of these systems leads to a steady removal of workload.

Thus it is the aim of this work to develop dynamic loop scheduling algorithms for SVM–Fortran. In a first step, the method of self–service–scheduling, which is well known for parallel computers with physically shared memory, is adopted for massively parallel machines with their relatively slow communication facilities. A second method – aligned-scheduling – takes into account the special requirements resulting from the arrangement of data in pages. The result is a guaranteed locality during data access. Finally, aligned self–service–scheduling – a synthesis of the two mentioned methods – turns out to be a good candidate to ensure both requirements: local data access as well as balanced workload.

Inhalt

1 Einleitung

Betrachtet man die Architekturkonzepte moderner Parallelrechner, so läßt sich die Zuordnung des Speichers zu den Prozessoren als charakteristisches Merkmal identifizieren. Zwei verschiedene Konstruktionsprinzipien [Bel92, GaPe94, SWLE92] sind dabei zu unterscheiden: Mehrprozessorsysteme mit gemeinsamem Speicher und Systeme, bei denen Prozessorelemente mit jeweils lokalem Speicher über ein Kommunikationsnetzwerk miteinander verbunden sind.

Es hat sich in der Praxis herausgestellt, daß Systeme mit gemeinsamem Hauptspeicher im allgemeinen leichter zu programmieren sind und für eine größere Klasse von Anwendungen befriedigende Leistungen erbringen [Bel92]. Der Grund dafür, daß man sich in letzter Zeit dennoch verstärkt netzwerkgekoppelten Rechnern zuwendet, ist in technologischen Problemen zu sehen, die die Anbindung vieler Prozessoren an einen gemeinsamen Hauptspeicher erschweren.

Netzwerkgekoppelte Parallelrechner lassen sich technisch einfacher realisieren und insbesondere auch leichter zu großen Systemen ausbauen. Der Nachteil dieser Systeme ist, daß sich deren Programmierung unter dem bisher gebräuchlichen Message-Passing-Programmiermodell als aufwendig und fehleranfällig erwiesen hat. Der Programmierer muß gemeinsam genutzte Daten explizit auf die Speicher der beteiligten Prozessoren verteilen. Ist dann ein Zugriff auf einen nicht lokal verfügbaren Teil eines gemeinsam genutzten Datenbereiches notwendig, so muß er die dafür erforderliche Kommunikation explizit in seinem Programm vorsehen. Diese Kommunikationsanweisungen sind dabei sowohl auf der Seite des Senders als auch auf der Seite des Empfängers zu berücksichtigen. Hinzu kommt, daß diese Kommunikation im Vergleich zu einem Speicherzugriff bei Systemen mit gemeinsamem Hauptspeicher derzeit deutlich länger dauert.

Die Programmierung von massiv-parallelen Rechenanlagen mit physikalisch verteiltem Speicher kann erleichtert werden, wenn ein globaler Adreßraum zur Verfügung gestellt wird. Systeme mit virtuell gemeinsamem Speicher [Li86, HuLi89] realisieren diesen globalen Adreßraum durch eine zusätzliche Speicherverwaltungsebene. Sie stellt ein angefordertes Datum zur Verfügung, unabhängig davon, ob es lokal vorhanden ist oder erst aus entferntem Speicher geladen werden muß. Auch wenn sich nicht-lokale Speicherzugriffe in der Art des Zugriffs nicht von lokalen Zugriffen unterscheiden, bleibt dennoch ein deutlicher Unterschied in der Zugriffszeit bestehen. Lokalität beim Datenzugriff erweist sich deshalb als entscheidender Performance-Faktor. Um einen hohen Anteil lokaler Datenzugriffe zu erzielen, stellt die Programmiersprache SVM-Fortran [BGNP94] Sprachmittel zur Verfügung, mit denen der Programmierer Arbeitsverteilungen auf Prozessoren angeben kann.

Um das Potential paralleler Rechnersysteme voll zur Beschleunigung eines einzelnen Anwendungsprogramms nutzen zu können, ist es erforderlich, daß alle zur Verfügung

stehenden Prozessoren während der gesamten Laufzeit ausgelastet sind. Oft läßt sich dies dadurch erreichen, daß schon während der Übersetzung eines Programms statisch eine Einteilung in gleich große Arbeitspakete festgelegt werden kann. Ist jedoch zu diesem Zeitpunkt die Information über den Rechenbedarf der einzelnen Programmteile noch nicht gegeben, bleibt nur eine dynamische Lastverteilung zur Laufzeit.

In technisch-wissenschaftlichen Programmen erweisen sich häufig einige wenige DO-Schleifen als die zeitkritischen Bereiche. Oft kann dabei die Bearbeitung einzelner Schleifeniterationen unabhängig voneinander erfolgen. Ausgereifte Analyse- und Transformationswerkzeuge [Con90, IBM91, Nag90] können heute genutzt werden, um diese Bereiche automatisch in parallelen Programm-Code zu überführen. Auch wenn der Rechenbedarf paralleler Arbeitspakete während der Übersetzung eines Programms noch nicht feststeht, möchte man dieses Potential an parallel ausführbaren Berechnungen nutzen können. Die Möglichkeit einer dynamischen Arbeitsverteilung in parallelen DO-Schleifen ist deswegen besonders wichtig.

Bei Parallelrechnern mit physikalisch gemeinsamem Speicher werden derartige Loop-Scheduling-Verfahren [FeRu90] heute schon mit Erfolg eingesetzt. Realisierungen basieren weitgehend auf einem Modell, bei dem ein zentrales Register oder eine ausgewiesene Stelle im gemeinsamen Speicher anstehende Arbeitspakete verwaltet. Kooperierende Prozessoren entnehmen dem zentralen Vorrat ein neues Arbeitspaket immer dann, wenn sie die Bearbeitung des vorangegangenen abgeschlossen haben.

Bei Rechnern mit physikalisch verteiltem Speicher steht ein gemeinsames Register nicht zur Verfügung, der konkurrierende Zugriff auf eine ausgewiesene Stelle im virtuell gemeinsamen Speicher erfolgt zu langsam. Um auch bei Systemen mit virtuell gemeinsamem Speicher nicht auf ein dynamisches Loop-Scheduling verzichten zu müssen, sollen in der vorliegenden Arbeit geeignete Verfahren entwickelt werden.

Diese Aufgabe läßt sich in zwei Teile gliedern. Zunächst ist ein dynamisches Loop-Scheduling zu entwickeln, das auch auf massiv-parallelen Systemen effizient ablaufen kann. Darauf aufbauend sind dann die besonderen Anforderungen, die die Organisation virtuell gemeinsamen Speichers an das Schleifen-Scheduling stellt, geeignet zu berücksichtigen.

In Kapitel 2 werden Idee und Realisationsmöglichkeiten eines logisch gemeinsamen Speichers vorgestellt. Es schließt sich in Kapitel 3 eine kurze Beschreibung der Programmiersprache SVM-Fortran an. Danach wird in Kapitel 4 aufgezeigt, warum beliebige Verteilungen von Schleifeniterationen auf Prozessorelemente erforderlich sind. Die Prinzipien dynamischer Lastausgleichsverfahren werden in Kapitel 5 erläutert. Eine Darstellung der Umsetzung dieser Prinzipien findet sich in Kapitel 6. Das Aligned-Scheduling ist Thema in Kapitel 7. Schließlich folgt in Kapitel 8, als Synthese der in Kapitel 6 und 7 erzielten Ergebnisse, die Beschreibung eines neuen Scheduling-Verfahrens. Dieses vereinigt, bei einem moderaten Overhead, die Vorteile eines lokalen Datenzugriffs mit denen eines dynamischen Lastausgleichs.

2 Logisch gemeinsamer Speicher

Betrachtet man moderne Parallelrechner, so lassen sich zwei Konstruktionsprinzipien unterscheiden: Auf der einen Seite stehen Rechner, bei denen moderat viele, sehr leistungsfähige Prozessoren gemeinsam auf einen Hauptspeicher zugreifen. Auf der anderen Seite stehen Anordnungen von im Prinzip eigenständigen Einzelrechnern, die über privaten Hauptspeicher verfügen und über ein Netzwerk miteinander gekoppelt sind. Diese Ansätze unterscheiden sich in mehrfacher Hinsicht:

- **Zahl und Leistung der Prozessoren:** In Systemen mit gemeinsamem Hauptspeicher ist die Zahl nutzbarer Prozessoren aufgrund der Komplexität der Anbindung von vielen Prozessoren an einen gemeinsamen Speicher und den daraus resultierenden Kosten begrenzt. Um in dieser Situation eine möglichst große Gesamtsystemleistung zu erhalten, wählt man Prozessoren am oberen Ende der Leistungsskala. Massivparallele Rechnersysteme schöpfen ihr Leistungspotential hingegen aus der hohen Zahl kooperierender Prozessorelemente. Diese großen Prozessorzahlen lassen sich realisieren, weil die zugrundeliegende Netzwerkarchitektur so gewählt ist, daß jede Vergrößerung die Struktur der Verbindung erhält. Lediglich der maximale Abstand im Netzwerk wächst an. In solchen Systemen entscheidet man sich für Standardprozessoren, da diese kostengünstig sind, nicht eigens entwickelt werden müssen, und nicht zuletzt auch, weil Höchstleistungsprozessoren die heute realisierbaren Kapazitäten der Verbindungsnetzwerke überfordern würden.

- **Verhältnis von Peak- zu Sustained-Leistung:** Es hat sich in der Praxis herausgestellt, daß Systeme mit gemeinsamem Hauptspeicher Anwendungen aus einem größeren Bereich effizient ausführen können, weil bei gemeinsamem Hauptspeicher die Kommunikation schneller abläuft und aufgrund der höheren Einzelprozessorleistung für die gleiche Arbeit weniger parallel zu verwaltende Prozesse notwendig sind.

- **Programmiermodell:** Das Shared-Memory-Programmiermodell ist für den Anwender bequemer, da auf gemeinsame Daten auch unmittelbar ohne zusätzlichen Programmieraufwand zugegriffen werden kann. Die Portierung vorhandener sequentieller Programme fällt leichter. Beim Message-Passing-Programmiermodell ist es Aufgabe des Programmierers, die Daten explizit auf die beteiligten Prozessoren zu verteilen. Bei einem Zugriff auf gemeinsame Daten muß er dann die erforderlichen Kommunikationsanweisungen sowohl für den Sender als auch für den Empfänger explizit im Programm-Code vorsehen.

Die aktuelle Situation läßt sich wie folgt bewerten: Aus der Sicht des Programmierers hätte man gern immer schnellere und immer größere Parallelrechner vom Shared-Memory-Typ. Da diese aber nicht in der gewünschten Größenordnung realisierbar sind, muß man sich mit netzwerkgekoppelten massiv-parallelen Rechnern beschäftigen. Dabei versucht man

die Vorteile beider Konstruktionsprinzipien, ein komfortables Programmiermodell und eine 'kostengünstige' skalierbare Hardware, miteinander zu verbinden (siehe Abbildung 2.1).

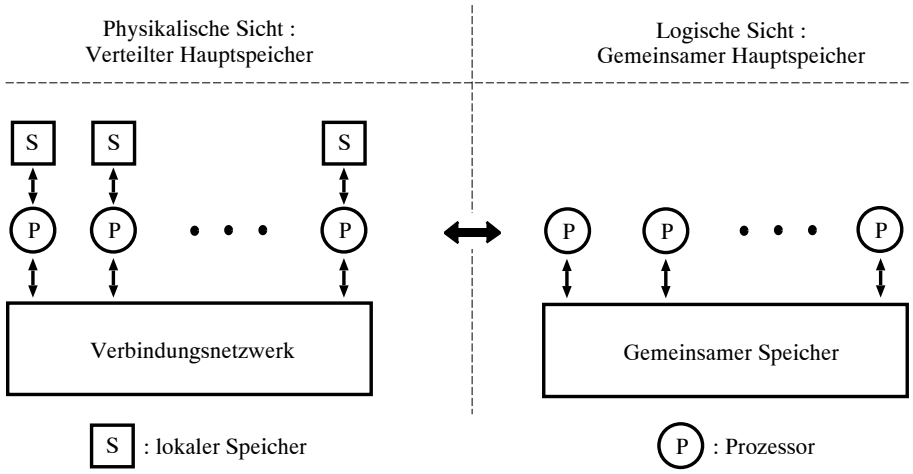


Abbildung 2.1: Das Prinzip des logisch gemeinsamen Speichers

In der Literatur werden verschiedene Möglichkeiten vorgestellt, wie diese Abbildung von physikalisch verteiltem auf logisch gemeinsamen Speicher realisiert werden kann. Drei verschiedene Ansätze werden gegenwärtig diskutiert:

1. **Global Shared Memory:** Ein Zugriff auf entfernten Speicher ist direkt möglich.
2. **Compiler-basierte Realisierungen:** Der Compiler fügt bei nicht-lokalen Speicherreferenzen die notwendigen Kommunikationsanweisungen ein.
3. **Virtual Shared Memory (VSM):** Es können mehrere Kopien eines Datums existieren. VSM garantiert, daß alle denselben Wert enthalten.

Abbildung 2.2 zeigt, welche Konzepte heute existieren, um physikalisch verteilten Speicher zu adressieren.

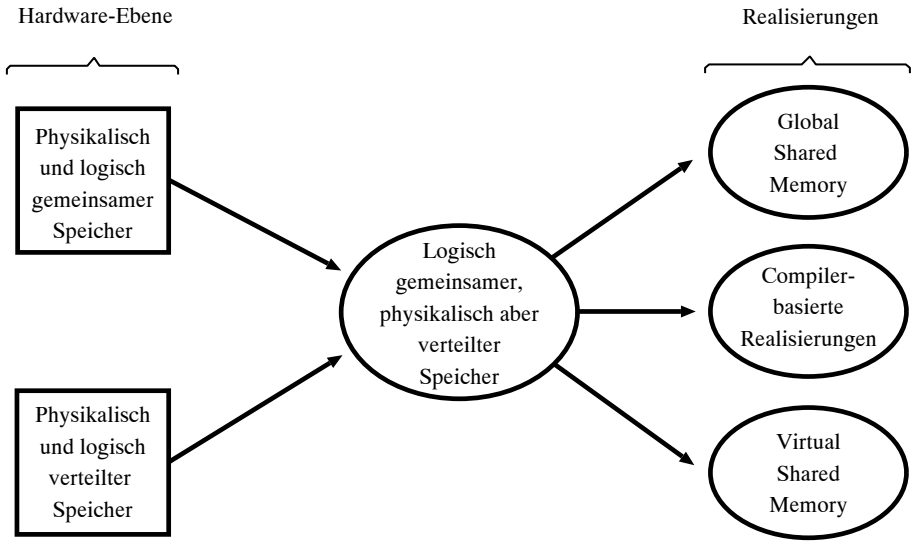


Abbildung 2.2: Logisch gemeinsamer, physikalisch aber verteilter Speicher

2.1 Global Shared Memory

Eine Möglichkeit, die angestrebte Abbildung von physikalisch verteiltem auf logisch gemeinsamen Speicher zu verwirklichen, ist das Prinzip des Global Shared Memory (vgl. [Oed93]). Dabei verfügt jeder Prozessor physikalisch über eigenen lokalen Speicher. Auf diesen kann über ein Hochgeschwindigkeitsnetz jeder andere Prozessor gleichberechtigt, wenn auch mit größerer Verzögerung, direkt zugreifen. Die Gesamtheit aller lokalen Speicher bildet so den für alle Prozessoren gemeinsamen Hauptspeicher. Der Zugriff auf diesen gemeinsamen Hauptspeicher erfolgt aus Sicht des Programmierers über einen globalen Adreßraum. Die Hardware entscheidet bei einem Zugriff auf eine bestimmte Stelle im globalen Adreßraum, ob ein lokaler oder ein entfernter Zugriff vorliegt. Bei einem entfernten Zugriff wird die Zugriffsanweisung ohne Zutun des Programmierers über das Netzwerk an die Speicherverwaltungseinheit des entfernten Knotens weitergeleitet. Diese greift dann auf das angesprochenen Datum zu und schickt den Wert bei einem Lese-Zugriff wieder an den anfragenden Knoten zurück. Ein entfernter Speicherzugriff unterscheidet sich für den Programmierer also lediglich in der Ausführungszeit, nicht aber in der Art des Zugriffs. Der Zugriff auf geteilte Daten kann nötigenfalls mit den Shared-Memory Synchronisationsprimitiven (Critical-Sections und Exclusive-Regions) gesichert werden. Kopien von Daten können vom Programmierer auf den lokalen Speichern angelegt werden, um Lesezugriffe zu beschleunigen. Sieht man von den lokalen Kopien ab, so existiert jedes Datum nur ein einziges Mal im Gesamtsystem, die Datenkonsistenz ist also a priori gegeben. Der Programmierer trägt dafür Sorge, daß auch beim Schreiben auf Daten, von denen lokale Lesekopien existieren, der deterministische Programmablauf gewährleistet bleibt.

Soll die Arbeit für ein breites Spektrum von Anwendungsprogrammen unter Global Shared Memory effizient bleiben, so ist ein Netzwerk mit einer sehr kurzen Startup-Zeit notwendig, da unter Umständen relativ häufig auf kleine Dateneinheiten in nicht-lokalem Speicher zugegriffen wird.

2.2 Compiler-basierte Realisierungen

Steht eine derart leistungsfähige Kommunikations- und Speicherverwaltungs-Hardware nicht zur Verfügung, so läßt sich ein logisch gemeinsamer Speicher auch auf der Ebene des Compilers realisieren. Der Benutzer gibt über Sprach-Direktiven eine Verteilung seiner Daten auf die Speicher der beteiligten Prozessoren vor. Schon während der Übersetzungszeit eines Programms wird dann entschieden, ob ein Datenzugriff lokal oder entfernt erfolgt. Bei einem entfernten Zugriff generiert der Compiler entsprechende Message-Passing-Anweisungen, die den erforderlichen Datentransfer leisten. Als Beispiele seien High Performance Fortran [HPF93, HPF94], Vienna-Fortran [ZBCM92] und Fortran D [HKKK92] genannt.

2.3 Virtual Shared Memory

Möchte man einen logisch gemeinsamen Speicher auf Rechnersystemen realisieren, die physikalisch über verteilten Speicher verfügen, so muß man in Hard- oder in Software einen Mechanismus zur Verfügung stellen, der diese Abbildung leistet. Man kann erwarten, daß diese Realisierung für sich selbst wenig Rechenzeit und Speicher in Anspruch nimmt, daß

sie Lokalität beim Datenzugriff gewährleistet, und schließlich, daß sie für den Benutzer transparent bleibt.

Virtual Shared Memory[1] (VSM) ist das dritte Konzept, mit dem diese Forderungen erfüllt werden sollen. VSM wurde erstmalig von Kai Li 1986 [Li86, HuLi89] zusammen mit einer ersten Realisierung auf einem Workstation-Cluster vorgestellt. Inzwischen hat VSM eine gewisse Verbreitung gefunden.

Da die vorliegende Arbeit auf VSM aufbaut, sollen im folgenden die Grundprinzipien und die sich daraus ergebenden Forderungen an eine auf VSM basierende Programmiersprache etwas detaillierter dargestellt werden.

Der zentrale Begriff bei der Realisierung von VSM lautet **Caching**. Caching bedeutet, daß für jeden Prozessor ein schneller lokaler Zwischenspeicher existiert, der eine Kopie gemeinsamer Daten enthalten kann. Der Vorteil besteht darin, daß ein wiederholter Zugriff auf 'lokale' Daten schnell durchgeführt werden kann. Da im System mehrere Kopien eines Datums existieren können, muß das zugrundeliegende Protokoll gewährleisten, daß alle diese Kopien immer dieselbe Sicht auf gemeinsame Daten haben. Die einzige Operation, bei der diese gemeinsame Sicht aufgehoben werden kann, ist die Schreiboperation. Das Protokoll muß also bei einer Schreiboperation die gemeinsame Sicht auf das geschriebene Datum sicherstellen. Caching ist dann effektiv, wenn der Vorteil des schnellen lokalen Datenzugriffs den Nachteil zusätzlicher Synchronisation zur Aufrechterhaltung der einheitlichen Sicht auf gemeinsame Daten überwiegt.

Besondere Bedeutung kommt den möglichen Prinzipien zur Aufrechterhaltung der Datenkohärenz zu.

2.3.1 Kohärenzmodelle

Bevor einzelne Methoden der Kohärenzsicherung vorgestellt werden, soll der Begriff der Speicherkohärenz selbst näher erläutert werden. Man unterscheidet zwischen starker und schwacher Speicherkohärenz. Die folgenden Definitionen sind [BCZ90] entnommen.

Definition 1 *Gemeinsamer Speicher wird als* **stark kohärent** *bezeichnet, wenn eine Lese-Operation immer genau denjenigen Wert liefert, den die letzte Schreiboperation an der betreffenden Adresse hinterlassen hat.*

Mit dieser Definition wird also die Situation, wie sie natürlicherweise in einem System mit physikalisch gemeinsamem Hauptspeicher besteht, nachgebildet [DSB86, DSB88]. Das Ziel ist, Erfahrungen, Methoden und Algorithmen, die für diese Rechner existieren, unverändert zu übernehmen. Die folgenden Ausführungen werden zeigen, daß dies bei der Realisierung zu einem nicht unerheblichen Aufwand führt, weshalb es sinnvoll erscheint, auch ein Modell schwächerer Datenkohärenz [BCZ90] zu betrachten:

Definition 2 *Gemeinsamer Speicher wird dann als* **schwach kohärent** *bezeichnet, wenn ein Protokoll existiert, das zu bestimmten Synchronisationszeitpunkten gewährleistet, daß eine Leseoperation den Wert liefert, den die letzte Schreiboperation an der betreffenden Adresse hinterlassen hat.*

[1] Möchte man nicht den Aspekt des logisch gemeinsamen Speichers betonen, sondern legt man Wert auf die Realisierung mit Hilfe des virtuellen Speicherkonzeptes, so kann man VSM auch als Shared Virtual Memory (SVM) bezeichnen.

An dieser Definition fällt auf, daß die Begriffe *Protokoll* und *bestimmter Zeitpunkt* unspezifiziert bleiben. Diese Unbestimmtheit eröffnet die Möglichkeit, den Aufwand, den die Kohärenzsicherung mit sich bringt, nur dort zu investieren, wo es zum deterministischen Ablauf des Programms auch tatsächlich erforderlich ist [GGH91]. Darin liegt aber auch gerade das Problem, denn die Information über notwendige Synchronisationszeitpunkte und geeignete Protokolle kann letztlich wieder nur der Programmierer 'von Hand' geben, so daß die oben genannte Forderung nach einer Transparenz der Abläufe beim Datenzugriff dabei nicht erfüllt werden kann.

Wünschenswert erscheint also eine Strategie, bei der als Vorgabe zunächst das starke Kohärenzmodell Gültigkeit hat, um eine korrekte Ausführung sicherzustellen. Um jedoch die Effizienz der Ausführung verbessern zu können, sollte der Programmierer auch die Möglichkeit haben, zu dem schwachen Kohärenzmodell wechseln zu können. Dies ist sinnvoll insbesondere für spezifizierte Regionen, in denen der Programmierer die Datenzugriffsstruktur kennt.

2.3.2 Kohärenzprotokolle

Mehrere Methoden zur Aufrechterhaltung der Datenkonsistenz wurden zuerst von Kai Li [Li86] vorgeschlagen und untersucht. Detailliertere Beschreibungen finden sich auch in [Hel90] und [Lin94]. Hier ist das Problem zu lösen, wie gewährleistet werden kann, daß auch dann, wenn ein kohärenzverletzender Schreibzugriff erfolgt, alle eventuell existierenden lokalen Kopien denselben Wert erhalten.

2.3.2.1 Das Ownership-and-Invalidation-Protokoll

Prinzipiell gibt es zwei Möglichkeiten, die einheitliche Sicht auf Daten zu garantieren [Hel90], auch wenn ein Datum, von dem mindestens eine Kopie im System vorhanden ist, geändert wird. Die erste Möglichkeit verlangt, daß bei einer Änderung eines Datums alle Kopien verworfen werden, während sich bei der zweiten Alternative diese Änderung auf alle Kopien auswirkt.

Realisiert wird die Variante, bei der Kopien verworfen werden, von Kai Li [Li86] dadurch, daß zu jedem Datum ein Besitzer und eventuell mehrere Teilhaber existieren. Der Besitzer übernimmt die Aufgabe der Kohärenzsicherung für dieses Datum. Das Protokoll sieht vor, daß er der einzige ist, der eine kohärenzverletzende Operation (also eine Schreiboperation) ausführen darf. Jeder Prozessor verwaltet eine Tabelle, in der sein eigenes Zugriffsrecht auf jedes existierende Datum eingetragen ist. Es gibt die drei Rechte $\{nil, read, write\}$. Datenkohärenz kann dann nicht verletzt werden, wenn ständig ein Schema aufrechterhalten bleibt, bei dem *ein einziger* Prozessor das Schreibrecht auf ein Datum hat, wobei eventuell mehrere Prozessoren zusätzlich lesen dürfen. Man spricht von einem **Read-Page-Fault**, wenn ein Prozessor auf ein Datum, auf das er kein Zugriffsrecht ($= nil$) hat, lesend zugreifen muß. Ein **Write-Page-Fault** liegt dann vor, wenn ein Prozessor ein Zugriffsrecht auf ein Datum $\in \{nil, read\}$ hat, es aber dennoch schreiben möchte.

Bei einem Write-Page-Fault wird die Kohärenz dadurch gewahrt, daß der Single-Writer-Zustand wiederhergestellt wird. Dazu wird zuvor an sämtliche Prozessoren, die eine (Read-Only-) Kopie dieses Datums halten, eine Ungültigkeitsnachricht versendet, deren Empfang die Teilhaber dazu veranlaßt, ihr Zugriffsrecht auf das betreffende Datum auf *nil* zu ändern.

Die Prozessoren, die auf dieses Datum anschließend weiter zugreifen wollen, müssen es sich dann erneut (inzwischen aktualisiert) vom Besitzer schicken lassen. Der Gedanke dabei ist, daß in der Regel nicht alle Prozessoren, die über eine Kopie verfügen, diese auch tatsächlich noch weiter benötigen und deshalb keine aktualisierte Kopie anfordern werden. Dies reduziert insgesamt das Nachrichtenaufkommen.

Muß ein anderer Prozessor als der Besitzer das Datum neu schreiben, so sendet dieser eine entsprechende Nachricht an den Besitzer. Der Besitzer veranlaßt daraufhin, daß sämtliche Kopien, einschließlich der eigenen, verworfen werden und sendet das Datum an den anfragenden Prozessor. Dieser wird, nachdem er das Datum empfangen hat, zu dessen neuem Besitzer.

2.3.2.2 Das Ownership-and-Updating-Protokoll

Kai Li hat ebenfalls die Variante zur Kohärenzsicherung realisiert, bei der sich eine Schreiboperation immer auf alle vorhandenen Kopien auswirkt. Wie beim Ownership-and-Invalidation-Protokoll existiert auch hier zu jedem Datum ein eindeutiger Besitzer. Dieser ist wieder für die Wahrung der Datenkohärenz verantwortlich. Der Unterschied zum Invalidierungs-Ansatz besteht darin, daß nach einer Schreiboperation der neue Wert allen Teilhabern mitgeteilt wird. Diese Strategie ist dann günstig, wenn fast alle Teilhaber das Datum auch weiterhin nutzen, da hierbei gegenüber dem Invalidierungs-Modell die Anfragenachrichten an den Besitzer entfallen.

2.3.2.3 Das Bus-Snooping-Protokoll

In den beiden oben beschriebenen Protokollen muß im System die Information vorhanden sein, wer der Besitzer eines jeden Datums ist. Ein anderer Prozeß, der eine Read-Only-Kopie oder das Schreibrecht verlangt, kann deshalb den Besitzer direkt adressieren, ohne für jeden Hauptspeicherzugriff einen Broadcast auszulösen, was sicherlich nicht mehr effizient wäre. Darüber hinaus muß zumindest der Besitzer die Information aller Teilhaber (Copy-Set) verwalten, damit er nötigenfalls gezielt Invalidierungs- bzw. Update-Nachrichten versenden kann.

Auf die Speicherung dieser Informationen kann aber verzichtet werden, wenn man von einer etwas anderen Hardware-Konfiguration ausgeht, bei der alle Prozessoren an einen gemeinsamen Bus angekoppelt sind. Ein Prozessor, der eine Kopie oder das Schreibrecht für ein Datum anfordert, gibt eine entsprechende Suchnachricht auf den Bus. Da an diesen Bus auch alle Speicher-Verwaltungs-Einheiten (SVE) aller anderen Prozessoren angeschlossen sind, und diese den Bus ständig abhören (snooping), wird sich schließlich auch der Prozessor, der das Datum besitzt, angesprochen fühlen, so daß er entsprechend antworten kann. Dies entspricht logisch also einem Broadcast, der hier allerdings durch den Bus hardwareunterstützt ist. Dieser Vereinfachung des Protokolls steht der Nachteil gegenüber, daß eine derartige Anordnung nicht skalierbar ist, denn die Buskapazität ist begrenzt.

Dieses Problem läßt sich zwar nicht vollständig beheben, aber zumindest mildern, wenn man statt eines einzelnen Busses eine Hierarchie von Bussen aufbaut. Daraus ergeben sich dann drei Vorteile:

1. Die maximale Entfernung zwischen zwei Prozessoren wächst nur logarithmisch mit der Zahl der Prozessoren.

2. Die Suche kann gleichzeitig in mehreren Teilbäumen erfolgen und ist folglich im Durchschnitt schneller erfolgreich.

3. Die Buskapazität ist nicht mehr so stark einschränkend.

Realisiert findet man dieses Konzept bei der KSR 1 [KSR] der Firma Kendall Square.

2.3.2.4 Das Protokoll zur Verhinderung von Inkohärenzen

Die drei oben beschriebenen Protokolle lassen kohärenzverletzende Schreiboperationen zunächst zu und stellen danach die einheitliche Sicht im Gesamtsystem auf das geschriebene Datum wieder her. Ein anderer Ansatz ist es, kohärenzverletzende Operationen erst gar nicht zuzulassen.

Dies kann realisiert werden, indem Operationen, die zu einer Kohärenzverletzung führen werden, solange blockiert werden, bis der Besitzer eines Datums seine Rechte explizit aufgibt. Eine solche Situation liegt dann vor, wenn:

1. Ein Prozessor schreiben möchte, obwohl eine oder mehrere Kopien des Datums existieren.

2. Ein Prozessor eine Kopie anfordert, obwohl ein anderer gerade das betreffende Datum schreibt.

Dieses Prinzip ist komplementär zum Ownership-and-Updating Protokoll. An die Stelle der Invalidierungsaufforderungen tritt hier die explizite Freigabe durch den Besitzer bzw. die Teilhaber.

Dabei ist dann, neben der zu garantierenden Deadlock-Freiheit, das Problem zu lösen, wann der beste Zeitpunkt für eine Freigabe ist, damit blockierte Prozesse nicht unnötig lange warten müssen. Zusätzlich sollen von dem Prozessor, der gerade ein Datum nutzt, sinnvolle Arbeitseinheiten geleistet werden können, damit er das Datum nach dessen Freigabe nicht unmittelbar wieder neu anfordern muß.

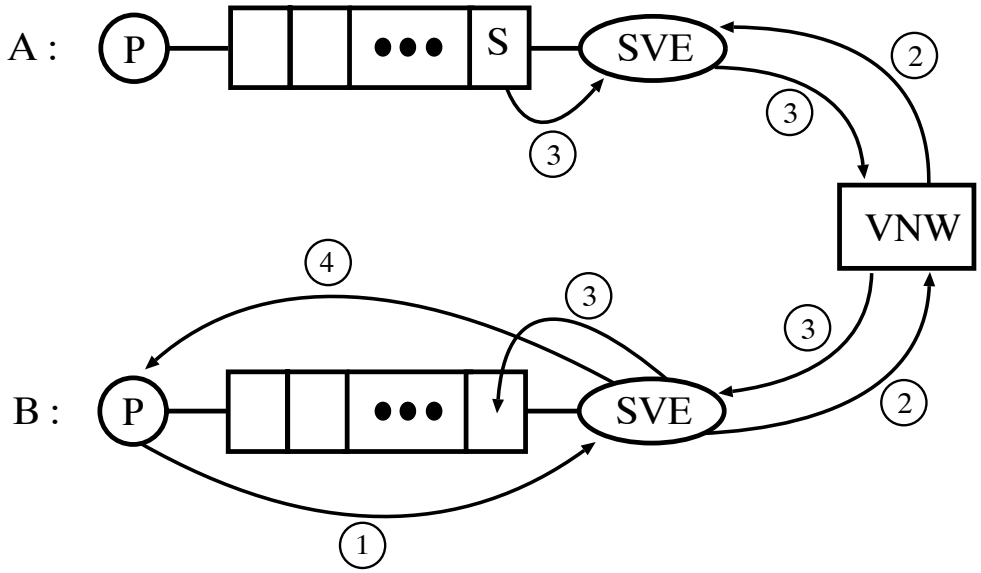
2.3.3 Die Größe von Dateneinheiten

Bei der Implementierung von VSM hat sich herausgestellt, daß eine Kohärenzsicherung für jedes einzelne Datum zuviel Verwaltungsinformation benötigen würde. Darüber hinaus müßten dabei sehr viele kurze Nachrichten versendet werden. Deshalb fassen alle Realisierungen von VSM Daten in Seiten-Einheiten zusammen. Die Maßnahmen zur Kohärenzsicherung wirken dann auf alle Daten in einer Seite gemeinsam. In der Praxis spielt es eine wichtige Rolle, in welcher Größe diese Dateneinheiten vorliegen. Sind sie zu klein, dann ergibt sich ein schlechtes Verhältnis zwischen Offset-Latency und reiner Sendezeit bei der Datenübertragung. Sind sie dagegen zu groß, so muß eventuell mehr übertragen werden, als zur Kohärenzsicherung tatsächlich nötig ist. Zu große Dateneinheiten führen darüber hinaus zum Effekt des **False Sharing**. Mehrere logisch unabhängige Dateneinheiten, auf die eigentlich von verschiedenen Prozessoren auch unabhängig zugegriffen werden könnte, befinden sich gemeinsam auf einer Seite, so daß ein konfliktfreier Zugriff parallel nicht mehr möglich ist.

Häufig werden mehrere Daten zu einer Seite fester Größe zusammengefaßt. Der optimale Wert dieser Größe hängt von Hard- und Software-Gegebenheiten ab, d.h. er kann sich zur Laufzeit ändern. Da die Größe einer Seite aber im allgemeinen fest ist, kann man nicht immer das Optimum realisieren. Interessant sind Ansätze, deren Dateneinheiten logisch zusammenhängende Objekte mit variabler Größe sind [Hel90]. Dies erschwert aber die Realisierung, weshalb bisher nur ein Prototyp zu Forschungszwecken verwirklicht worden ist.

2.3.4 Pagetrashing

Unabhängig davon, welches Protokoll zur Kohärenzsicherung verwendet wird, sieht man in der Praxis häufig einen unangenehmen Effekt: In einer Situation, in der mehrere Prozessoren gleichzeitig um eine gemeinsame Seite konkurrieren, stellt sich ein Ablauf ein, der zu erheblichen Performance-Einbußen führt. Dieser soll am Beispiel zweier Prozessoren im Ownership-and-Invalidation-Protokoll dargestellt werden soll (siehe Abbildung 2.3):



P : Prozessor
VNW : VerbindungsNetzWerk
S : Speicherseite
SVE : SpeicherVerwaltungsEinheit

(1) : Seitenfehler auf Seite S
(2) : B fordert A auf, die Seite S zu senden
(3) : Seite S wird übertragen
(4) : B kann seine Berechnung fortführen

Abbildung 2.3: Beim Pagetrashing wiederholt sich der dargestellte Ablauf mit vertauschten Rollen von A und B für jeden Zugriff auf ein Datum in der Speicherseite S.

Die Prozessoren A und B seien mittels ihrer Speicher-Verwaltungs-Einheiten (SVE) über ein Verbindungsnetzwerk (VNW) miteinander verbunden. Zu Beginn sei A der Besitzer

der Seite S. Dann möchte B in S schreiben und löst einen Write-Page-Fault aus, da er kein Schreibrecht hat. Dies hat zur Folge, daß:

1. die SVE von B den Besitzer von S herausfinden werden muß (also A),
2. eine Nachricht mit der Aufforderung, die Seite zu übertragen von B nach A geschickt wird,
3. A seine Berechnung unterbricht, seinen Copy-Set untersucht, eventuell Invalidierungsaufforderungen an Teilhaber der Seite verschickt, die gesamte Seite (obwohl nur ein einzelnes Datum benötigt wird) an B überträgt,
4. und schließlich B das Schreibrecht für die Seite annimmt und seine Berechnung wieder aufnimmt.

Diese neu angefangene Berechnung kann B allerdings nicht lange fortführen, denn der genannte Ablauf wiederholt sich sofort mit vertauschten Rollen von A und B, da auch A weiterhin in die Seite S schreibt.

Insgesamt ergibt sich ein Kommunikations-Overhead, der die eigentliche Rechenzeit leicht um ein Hundertfaches überschreiten kann.

Erkennt der Programmierer dieses Verhalten, so kann er in bestimmten Situationen dadurch Abhilfe schaffen, daß er für jeden beteiligten Prozeß ein eigenes lokales Datum zur Verfügung stellt und erst am Schluß die Zwischenergebnisse zusammenfaßt. Eine andere Möglichkeit ist die Vorgabe der **schwachen Kohärenz** begrenzt auf diesen Programmbereich, falls der Benutzer sicher weiß, daß die Prozessoren A und B zwar konkurrierend in dieselbe Seite, aber auf unterschiedliche Daten schreiben.

2.3.5 Forderungen an SVM-Programmiersprachen

Alle beschriebenen Mechanismen leisten die Abbildung von physikalisch verteiltem auf logisch gemeinsamen Speicher. Der Anwendungsprogrammierer braucht sich zunächst keine Gedanken über die internen Abläufe zu machen. Programme, die für Shared-Memory-Rechner entwickelt worden sind, können nach kleinen Änderungen sofort unter SVM zur Ausführung gebracht werden.

Noch nicht berücksichtigt ist dabei die Frage nach der Performance. Untersuchungen zur Ablaufplanung unter SVM und dessen Einfluß auf die Effizienz der Ausführung sind in [Lin94] durchgeführt worden. Es ergaben sich dabei die folgenden Forderungen:

- **Ausrichtung von Datenfeldern auf Seitengrenzen:** Dies ist wichtig, um *False-Sharing* und damit verbundenes *Pagetrashing* zu vermeiden. Man investiert dabei einen gewissen Speicherverschnitt, da nur eine logische Dateneinheit pro Seite gespeichert werden kann.
- **Angepaßter Zugriff auf Datenfelder:** Um unnötige Seitenfehler zu vermeiden, ist es notwendig, auf Datenfelder, die die Größe einer Seite überschreiten, in derselben Ordnung zuzugreifen, mit der sie im Speicher abgelegt sind (in Fortran also spaltenweise).

- **Prefetching:** Besonders in Situationen, in denen viele Prozessoren auf dieselben Daten innerhalb einer bestimmten Seite lesend zugreifen, kann eine vom Programmierer im voraus initiierte Leseanforderung eine gewisse Entlastung bringen. Busy-Waiting kann so reduziert werden, da sich Kommunikation und Berechnung überlappen.
- **Wiederverwendung von Zugriffsmustern zur Nutzung von Lokalität:** In Systemen mit einer hierarchischen Speicherstruktur ist es besonders wichtig, daß der Anteil lokaler Datenzugriffe relativ hoch ist. Methoden, mit denen dieses Ziel erreicht werden kann, sind in der oben genannten Arbeit [Lin94] sowie in den Abschnitten 3 und 4 beschrieben.
- **Beachtung von Seitengrößen bei dynamischem Load-Balancing:** Bei der Simulation des Verhaltens von SVM-Systemen hat sich unter anderem herausgestellt, daß dynamische Lastverteilungsverfahren in der Form, wie sie bisher implementiert sind, zu Pagetrashing führen. Es ist daher das Ziel dieser Arbeit, Algorithmen zu finden, die SVM-spezifische Anforderungen besser berücksichtigen.

Die am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich entwickelte Programmiersprache SVM-Fortran [BGNP94] berücksichtigt diese Forderungen. Eine genauere Beschreibung des SVM-Fortran Projektes, in das diese Arbeit eingebettet ist, findet sich im anschließenden Kapitel.

3 Die Sprache SVM-Fortran

SVM-Systeme stellen dem Benutzer auf Rechnern mit physikalisch verteiltem Speicher einen zusammenhängenden globalen Adreßraum zur Verfügung. Existierende Programme für Shared-Memory-Rechner sind im allgemeinen nach kleinen Modifikationen direkt unter SVM ablauffähig.

Es stellt sich also die Frage, warum man sich der Entwicklung einer neuen Sprache für Shared Virtual Memory widmen muß. An erster Stelle kann das Fehlen von Sprachprimitiven zur Spezifikation von Parallelität in Standard–Fortran angeführt werden. Darüber hinaus ist aber festzuhalten, daß die Ablauffähigkeit von Shared-Memory-Programmen allein nicht ausreicht. Voraussetzung für eine breite Akzeptanz des Shared-Memory-Programmiermodells auf SVM-Rechnern ist auch eine Performance, die nicht wesentlich hinter der äquivalenter Message-Passing-Systeme zurücksteht.

Es hat sich herausgestellt, daß in SVM-Systemen eine große Zahl von Seitenfehlern zu einer schlechten Performance führt. Seitenfehler sind deshalb besonders aufwendig, weil sie zusätzlich zur reinen Datenübertragung einen Verwaltungsaufwand mit sich bringen, da der Prozessor, bei dem sich ein angefordertes Datum befindet, zunächst ausfindig gemacht werden muß. Die Reduktion der Seitenfehler auf ein Minimum ist daher entscheidend. Erreicht werden kann dies, indem man Lokalität beim Datenzugriff gewährleistet. Lokalität beim Datenzugriff ist dann gegeben, wenn sich im verteilten System Daten gerade dort befinden, wo sie zum Ablauf der Berechnung benötigt werden. Da zur Verfügung stehenden Prozessoren sowohl Daten als auch Arbeitspakete zugeordnet sind, hängt Lokalität gleichzeitig von zwei Verteilungen ab. Erst wenn diese beiden Verteilungen aufeinander abgestimmt sind, kann sich die geforderte Lokalität beim Datenzugriff einstellen.

Das Problem einer effizienten Programmierung in hierarchischen Speicherstrukturen existiert nicht nur bei SVM-Systemen. Bei dem Message-Passing-Programmiermodell ist der Programmierer für beide Zuordnungen explizit verantwortlich. Der entscheidende Nachteil dabei ist, daß er sie ohne jede Unterstützung selbst realisieren muß. Dies führt häufig zu Programmen, die nicht flexibel genug auf Änderungen von Eingabedaten, Systemeigenschaften oder laufzeitabhängigen Parametern reagieren können.

In den vergangenen Jahren wurden deshalb vielfältige Anstrengungen unternommen, Fortran 77 bzw. dessen Nachfolger Fortran 90 um Sprachelemente zu ergänzen, die den Programmierer gerade von dieser Aufgabe entlasten sollen. Einige Hardware-unabhängige Konzepte haben besondere Bedeutung erlangt; genannt seien hier High-Performance-Fortran (HPF) [HPF93, HPF94], Vienna Fortran [ZBCM92] und Fortran D [HKKK92]. Alle drei Konzepte unterstützen datenparallele Anwendungen auf Systemen mit verteiltem Speicher.

Das Gemeinsame dieser Dialekte ist das Verfahren, mit dem die aufeinander abgestimmte Verteilung von Daten und Arbeitslast erreicht werden soll: Der Benutzer gibt über geeig-

nete Sprachprimitive eine Strategie zur Datenverteilung vor. Die dazu 'passende' Arbeitsverteilung - im folgenden auch als Scheduling bezeichnet - wird dann nach der **owner-compute-rule** von Compiler und Laufzeitsystem bestimmt. Diese owner-compute-rule besagt, daß Berechnungen nach Möglichkeit dort ausgeführt werden, wo die benötigten Daten lokal zur Verfügung stehen.

Die folgenden Ausführungen beschreiben die Realisierung der Datenverteilung in HPF genauer, da HPF als herstellerübergreifender Standard eine gewisse Verbreitung gefunden hat und darüber hinaus an einigen Stellen Vorbild bei der Definition der Syntax von SVM-Fortran war.

3.1 Die Datenverteilung und das TEMPLATE in HPF

In HPF spezifiziert der Programmierer die Verteilung seiner Daten auf die zur Verfügung stehenden Prozessoren. Diese Verteilung erfolgt in drei Stufen (vergleiche Abbildung 3.1), in deren Mittelpunkt das **TEMPLATE** steht.

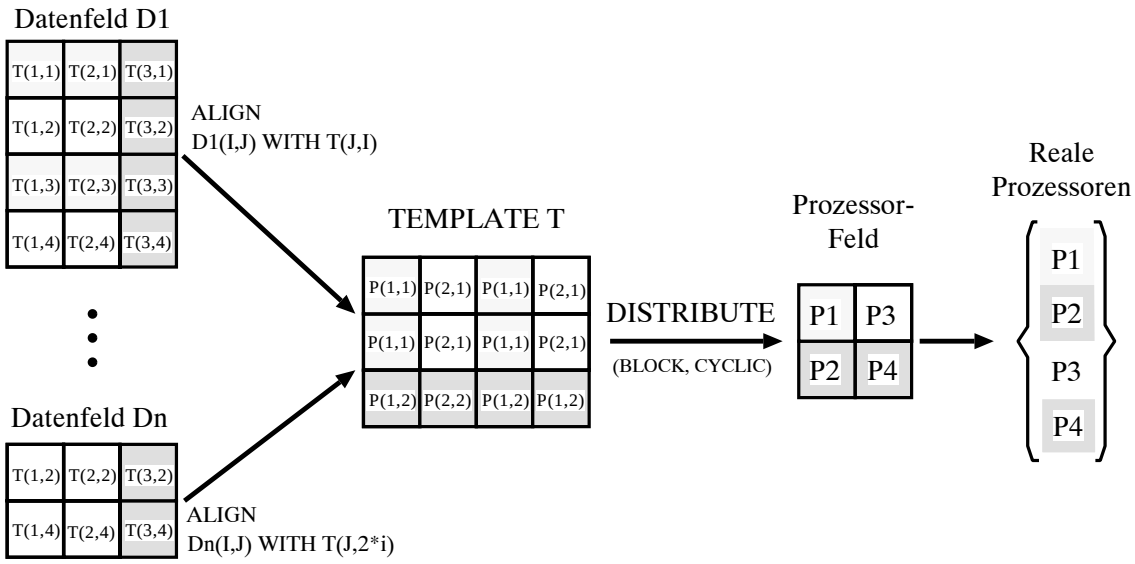


Abbildung 3.1: Die dreistufige Verteilung von Datenfeldern auf physikalische Prozessoren in HPF

Die Zuordnung von Datenfeldelementen auf Prozessoren ist in Abbildung 3.1 durch ein gemeinsames Muster kenntlich gemacht. In der ersten Stufe wird auf der Menge der physikalischen Prozessoren eine Struktur in Form eines ein- oder mehrdimensionalen Prozessorfeldes definiert. Dies erleichtert bei der Programmierung die Anpassung an die Struktur des zu lösenden Problems und abstrahiert von der Verbindungs-Topologie der realen Maschine. In der zweiten Stufe wird dann ein abstrakter Indexraum, der wieder ein- oder mehrdimensional sein kann, auf das Prozessorfeld verteilt. Diesen abstrakten Indexraum bezeichnet man dabei als **TEMPLATE**. Die Verteilung erfolgt für jede Dimension einzeln nach einer benutzerspezifizierten Strategie (z.B. blockweise oder zyklisch). Auf dieses

TEMPLATE können schließlich die Datenfelder ausgerichtet (aligned) werden. Dabei gibt eine Zugriffsfunktion wie z.B.

ALIGN Datenfeld D1(I,J) WITH TEMPLATE T(J,2*I))

die Zuordnung von Datenfeldelementen zu TEMPLATE-Elementen an. Obwohl die zusätzliche Ebene des TEMPLATES nicht prinzipiell notwendig ist [CMZ93], hat man sie in HPF realisiert. So kann beim Zugriff auf Teile eines schon verteilten Datenfeldes oder bei einem Zugriff auf dasselbe Datenfeld mit einem anderen Zugriffsmuster leicht herausgefunden werden, bei welchem Prozessor sich die adressierten Daten befinden. Der Programmierer wird dabei nicht mit komplizierten Adreßumrechnungen belastet. Komfortabel sind TEMPLATES auch in Situationen, in denen mehrere Datenfelder an einem gemeinsamen Feld ausgerichtet werden sollen, ohne daß dabei auf dieses Feld selbst zugegriffen wird.

3.2 Die Arbeitsverteilung und das TEMPLATE in SVM-Fortran

Die Zielrechner für HPF und SVM-Fortran weisen einen physikalisch verteilten Speicher auf. Dies bringt für beide Sprachen das Problem einer aufeinander abgestimmten Wahl von Scheduling und Datenverteilung mit sich.

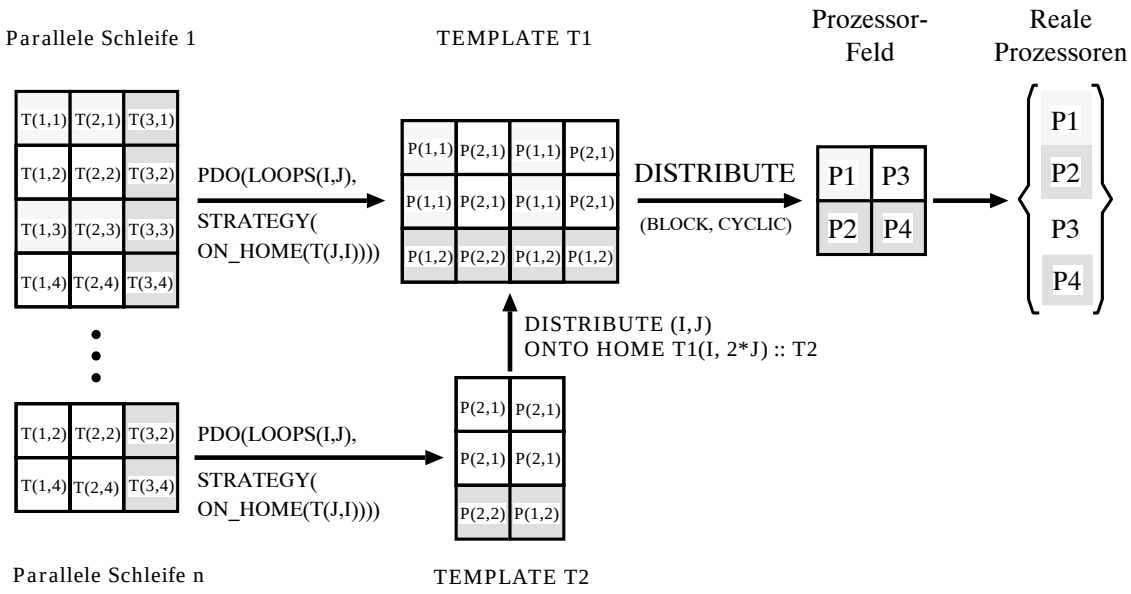


Abbildung 3.2: Die dreistufige Verteilung von Schleifeniterationen auf physikalische Prozessoren in SVM-Fortran

Allerdings unterscheidet sich SVM-Fortran von HPF darin, daß der Benutzer aufgrund der Transparenz der Speicherverwaltung zunächst keine Kenntnis über die gerade vorliegende Verteilung seiner Daten hat. Die erstrebte Koinzidenz von Datenverteilung mit dem Schleifen-Scheduling erscheint also zunächst als nicht-realisierbar.

Wie in Abschnitt 2.3 dargestellt erfolgt die Speicherung von Datenseiten aber nicht regellos, denn eine Speicherseite befindet sich immer bei dem Prozessor, der zuletzt auf sie

zugegriffen hat. Bei einem wiederholten Zugriff eines Prozessors auf dieselbe Seite kann man annehmen, daß sie in der Zwischenzeit bei diesem Prozessor verblieben ist. Dies ist dann der Fall, wenn kein anderer Prozessor schreibend auf diese Seite zugegriffen hat und wenn diese Seite nicht wegen Speichermangels ausgelagert werden mußte.

Erfolgversprechend ist ein Ansatz, bei dem der Benutzer eine Scheduling-Strategie vorgibt, deren wiederholte Anwendung schließlich wegen der Migrationseigenschaft der Daten zu der erwünschten Lokalität beim Zugriff führt. SVM-Fortran benutzt deshalb ebenfalls TEMPLATES. Sie werden allerdings hier nicht mehr zur Speicherung von Datenverteilungen, sondern zur Speicherung von Arbeitsverteilungen verwendet, damit ein bestimmtes Zugriffsmuster mehrfach realisiert werden kann.

3.3 Weitere Eigenschaften von SVM-Fortran

SVM-Fortran [BGNP94] - eine Erweiterung der im wissenschaftlichen Bereich etablierten Sprache Fortran 77 - wurde mit dem Ziel entwickelt, SVM-spezifische Sprachelemente zur Verfügung zu stellen und diese auf ihre Eignung zu untersuchen.

Neben dem TEMPLATE-Konzept hatten folgende Aspekte bei der Festlegung der Sprache zentrale Bedeutung:

- Bereitstellung von Sprachdirektiven zur Spezifikation von Parallelität
- Unterstützung von Daten- und Funktionsparallelität
- Flexible Strategien zur Arbeitsverteilung, insbesondere auch irreguläre Verteilungen
- Benutzerdefinierte Prozessorfelder zur Abstraktion von der physikalischen Verbindungstopologie
- Unterscheidung von gemeinsamen und privaten Variablen
- Spezielle SVM-Unterstützung wie: Prefetching, Page-Locking, Wechsel zwischen strikter und schwacher Datenkohärenz
- Wahl zwischen exklusiver oder redundanter Ausführung in benutzerspezifizierten Regionen
- Erleichterung einer schrittweisen Parallelisierung von sequentiellen Fortran-Programmen
- Maschinenunabhängigkeit

SVM-spezifische Direktiven werden durch das Präfix CSVM$ gekennzeichnet. Dadurch wird erreicht, daß diese Sprachelemente auf anderen Systemen als Kommentar interpretiert werden, so daß die so geschriebenen Programme unverändert zu Testzwecken auch dort zur Ausführung gebracht werden können. Um ein Maximum an Flexibilität zu garantieren, ist eine dreistufige Transformation eines vorhandenen Fortran 77 Programms in einen auf der jeweiligen Zielmaschine lauffähigen Code realisiert (siehe Abbildung 3.3). In der ersten Stufe gibt der Programmierer in seinem Standard–Fortran 77 Programm mit Hilfe der CSVM$-Direktiven die zur Parallelverarbeitung notwendigen Anweisungen an.

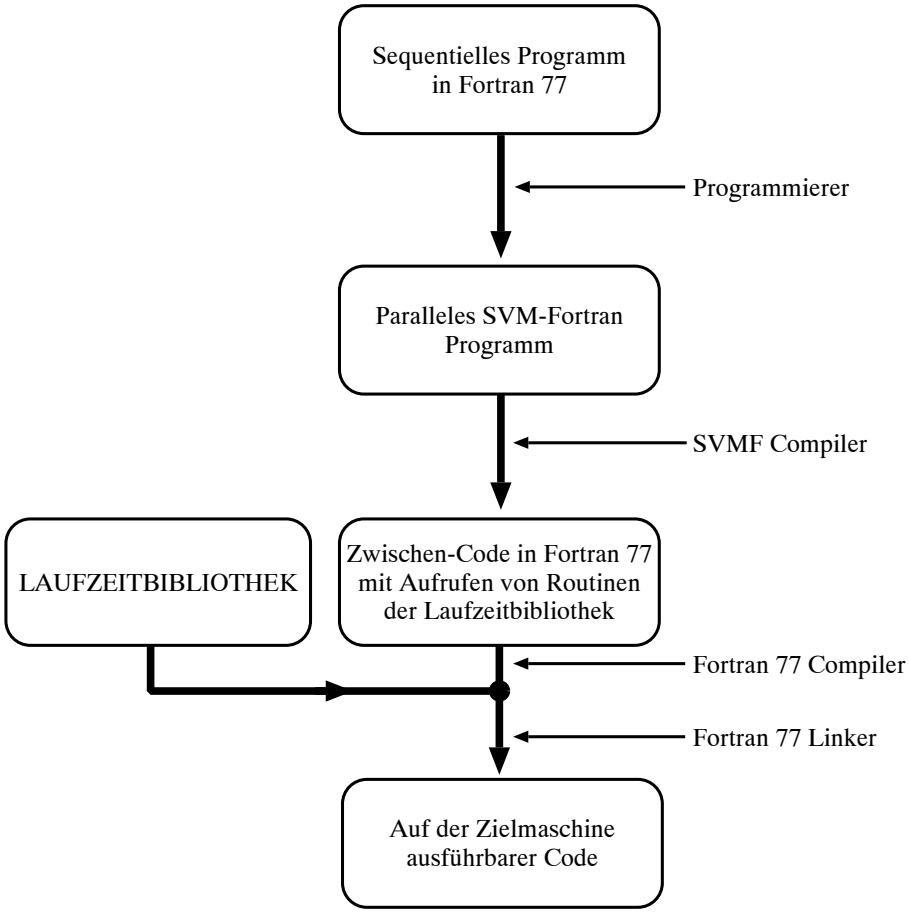


Abbildung 3.3: Die dreistufige Code-Generierung

Diese werden von dem SVM-Fortran-Compiler zu Anweisungen in Standard–Fortran 77 und zu Aufrufen von geeigneten Laufzeitroutinen expandiert. Schließlich wird mittels des für die jeweilige Zielmaschine vorhandenen Fortran 77 Compilers unter Einbindung der Laufzeitbibliothek der ausführbare Code generiert.

4 Nicht-regelmäßige Verteilungen von Schleifeniterationen

In SVM-Systemen dauert ein nicht-lokaler Speicherzugriff im Vergleich zu einem lokalen Zugriff deutlich länger. Neben der reinen Datenübertragung erfordert das Auffinden des betreffenden Datums eine nicht zu vernachlässigende Zeit (vergleiche Abschnitt 2.3). Handelt es sich bei dem Zugriff um eine Schreiboperation, so ist zusätzlich der Ablauf des Kohärenzsicherungsprotokolls abzuwarten, bevor die Berechnung fortgeführt werden kann. Unter dem Betriebssystem ASVM [Zei93] liegen gemessene Page-Fault-Zeiten in einer Größenordnung von 2,5 ms. Ein nicht-lokaler Speicherzugriff unterscheidet sich damit etwa um den Faktor 10000 von einem lokalen Zugriff. Für eine effiziente Ausführung ist deshalb ein möglichst großer Anteil von lokalen Datenzugriffen entscheidend.

Die parallele Schleife ist das Sprachelement, das feingranulare Parallelität ermöglicht. Lokalität beim Datenzugriff ist deshalb für parallele Schleifen besonders wichtig. Diese Lokalität beim Datenzugriff liegt dann vor, wenn der einzelne Prozessor genau denjenigen Teil der gesamten Schleifeniterationen ausführt, dessen Daten in seinem lokalen Speicher vorhanden sind. In Schleifen wird im allgemeinen auf mehrere Felder lesend und schreibend zugegriffen, weshalb dies nicht immer für jeden Feldzugriff möglich ist. Konkurrierende Schreibzugriffe in SVM–Systemen sind dabei besonders aufwendig; deshalb ist man bemüht, die Lokalität zumindest für schreibende Zugriffe zu gewährleisten.

Da benutzergesteuerte Datenverteilungen ausscheiden, wählt man in SVM–Systemen geeignete Schleifen–Scheduling–Strategien, um Datenlokalität zu erreichen. In [Lin94] sind Verfahren zur Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamem Speicher untersucht worden. Dabei haben sich zwei Schleifen–Scheduling–Strategien als geeignet herausgestellt, um Lokalität beim Datenzugriff zu erreichen:

- **Aligned–Scheduling:** Diese Strategie stellt sicher, daß dort gerechnet wird, wo die benötigten Daten vorliegen. Es bedarf der Unterstützung durch das SVM–Laufzeitsystem, denn jeder Prozessor muß zur Laufzeit aus der Page–Owner–Tabelle die Information auslesen können, welche der zu bearbeitenden Feldelemente gerade lokal vorhanden sind. Nur diese werden dann von ihm selbst bearbeitet. Das primäre Zielsystem ASVM [Zei93] bietet bisher als einziges Basissystem diese Funktionalität. Alle anderen SVM–Realisierungen stellen diese Information bisher nicht zur Verfügung, zum einen, weil die Idee des Aligned–Scheduling noch relativ neu ist, zum anderen aber auch, weil die Speicherverwaltung in SVM–Systemen für den Benutzer ursprünglich vollständig transparent sein sollte. Eine detaillierte Darstellung des Aligned–Scheduling findet sich in Kapitel 7.

- **Affinity–Scheduling:** Eine Eigenschaft von SVM–Systemen ist es, daß Daten zu dem Prozessor migrieren, der zuletzt schreibend auf sie zugegriffen hat. Dies führt dazu, daß sich bei einem mehrfachen Zugriff eines bestimmten Prozessors auf denselben Teil eines Datenfeldes die referenzierten Daten beim ersten Zugriff eventuell aus nicht-lokalem Speicher geladen werden mußten. In günstigen Fällen stehen sie dann aber für alle weiteren Iterationen lokal zur Verfügung. Diese günstigen Fälle sind dann gegeben, wenn keine Speicherseiten wegen Speichermangels ausgelagert werden müssen. Darüber hinaus darf in der Zwischenzeit kein anderer Prozessor auf diese Daten schreibend zugegriffen haben.

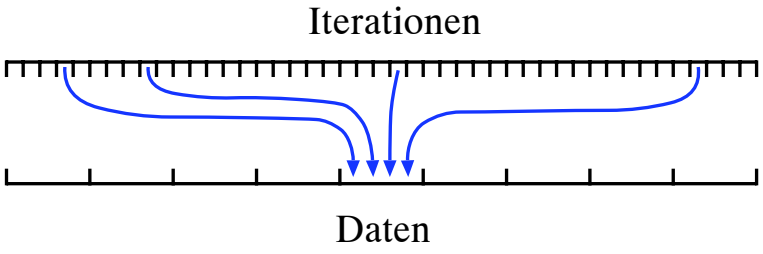
  Liegt in einem Programm nun eine Situation vor, bei der ein parallel bearbeiteter Kern mehrmals iteriert wird, so sorgt man dafür, daß immer dieselben Prozessoren auch dieselben Iterationen bearbeiten, damit ab der zweiten Iteration dort gerechnet wird, wo auch die benötigten Daten lokal zur Verfügung stehen.

In SVM–Fortran ist bisher die Strategie des **Affinity–Scheduling** durch das TEMPLATE–Konzept (siehe Abschnitt 3) realisiert. Der Benutzer gibt dabei eine bestimmte Verteilung von Schleifeniterationen auf Prozessoren vor. Mehrere Schleifen, die auf dasselbe Datenfeld zugreifen, können so nach derselben Strategie verteilt werden, damit sich der oben dargestellte Ablauf einstellen kann. Bisher kann der Benutzer die regulären Verteilungen BLOCK, GENERAL BLOCK, CYCLIC und BLOCK CYCLIC vorgeben.
Die Vorteile dieser regulären Verteilungsstrategien sind:

- Eine einfache Realisierbarkeit
- Ein geringer Speicherbedarf, da zur Spezifikation wenige Parameter ausreichen (siehe [Plu94])
- Eine effiziente Berechnung zur Laufzeit

Dennoch sind diese regulären Strategien in einigen Fällen nicht flexibel genug:

1. Bei **indirektem Feldzugriff** greift man in einer parallelen Schleife auf ein Datenfeld nicht direkt über den Laufindex, sondern indirekt über eine allgemeine Indexfunktion oder über ein Mapping–Array zu.



   Dabei ergeben sich für Iterationen, die im Speicher aufeinanderfolgende Daten adressieren, im allgemeinen keine regulären Verteilungsstrukturen.

2. Beim **Aligned–Scheduling** hängen die lokal zu bearbeitenden Iterationen über eine Align–Funktion von der gerade vorliegenden Datenverteilung ab. Diese Datenverteilung ergibt sich aber, wie in Abschnitt 2.3 dargestellt, aus dem dynamischen Ablauf

im System. Sie ist im allgemeinen für den Anwender transparent, nicht deterministisch und insbesondere nicht von regulärer Struktur. Möchte man beim Aligned–Scheduling das sich ergebende Zugriffsmuster speichern, so reichen die regulären Verteilungsmuster dazu nicht mehr aus.

3. **Dynamische Lastbalancierung** ist dann notwendig, wenn die Arbeitslast in einer einzelnen Schleifeniteration nicht konstant ist, sondern von Laufzeitparametern abhängt. Es wird angestrebt, daß auch in diesem Fall alle Prozessoren gleichmäßig ausgelastet sind, so daß zur Verfügung stehende Ressourcen möglichst gut ausgenutzt werden können. Da eine dynamische Lastverteilung von Informationen zur Laufzeit abhängt, ergibt sich dabei im allgemeinen kein deterministischer Ablauf. Man muß deswegen davon ausgehen, daß sich die dabei ergebende Verteilung von Arbeitspaketen auf Prozessoren keine reguläre Struktur mehr aufweisen wird. Berücksichtigt man, daß dynamische Lastverteilungsverfahren auf massiv–parallelen Rechnern einen relativ großen Overhead mit sich bringen (vergleiche Abschnitt 5), so wird der Wunsch deutlich, ein sich so ergebendes Verteilungsmuster speichern zu können. Liegt eine Verteilung dann einmal vor, so kann man sie wiederverwenden, ohne erneut den Overhead, den die dynamische Verteilung mit sich bringt, investieren zu müssen.

Die angeführten Punkte zeigen, daß selbst Basissprachmittel wie z.B. der indirekte Feldzugriff innerhalb einer parallel auszuführenden Schleife auf Iterationsverteilungen von nicht–regulärer Struktur führen. Ein Mechanismus, der diese Art der Verteilung realisiert, ist also unverzichtbar. Dabei sind insbesondere die Spezifikation und die Speicherung dieser Verteilungen zu leisten.

4.1 Die gewählte Datenstruktur

Möchte man also auch nicht–reguläre Iterationsverteilungen speichern können, so ist dazu eine geeignete Datenstruktur erforderlich. Der Begriff 'nicht–reguläre Verteilung' deutet schon an, daß man eine derartige Verteilung nicht mehr durch die Angabe weniger Parameter spezifizieren kann. Es bleibt deswegen keine andere Möglichkeit, als eine Tabelle anzulegen, die die Zuordnung von TEMPLATE–Elementen auf Prozessorfeldelemente speichert.

Mit der Frage der Art der Darstellung ist zusätzlich auch die Frage verbunden, ob die Verteilung von TEMPLATES zentral oder besser dezentral zu speichern ist.

Der erste Ansatz, der der logischen Sicht dieser Verteilung als Abbildung von TEMPLATE–Elementen auf Prozessorfeldelemente am weitesten entspricht, ist die zentrale Speicherung einer Tabelle, die für jeden möglichen Schleifenindex einzeln einen Eintrag mit dem zugehörigen Prozessorelement enthält. Dabei ergeben sich allerdings gleich mehrere Probleme:

- Wenn nur ein zentraler Server die gesamte Information verwaltet, müssen alle Prozessoren, die an parallelen Schleifen partizipieren, vor der eigentlichen Ausführung erst die lokal auszuführenden Arbeitspakete ermitteln. Dies bedeutet zusätzliche Kommunikation und einen Engpaß bei dem zentralen Server.

- Jeder der an einer parallelen Schleife mitarbeitenden Prozessoren benötigt nur die Information über diejenigen Iterationen, die er selbst zu bearbeiten hat. Die genannte Darstellung enthält die Information aber gerade umgekehrt: Statt Iterationen pro Prozessor wären Prozessoren pro Iteration gespeichert. Eine explizite Invertierung des TEMPLATES vor jeder Ausführung einer parallelen Schleife wäre also notwendig.

- Da bei der Ausführung von parallelen Schleifen jeder beteiligte Prozessor die Information in der invertierten Darstellung benötigt, die Information aber zusätzlich zentral in der nicht invertierten Form vorliegen würde, würde insgesamt der doppelte Speicherplatz belegt werden.

- Es ist zugelassen, daß Schleifeniterationen auf mehrere Prozessorfeldelemente repliziert werden. In diesem Fall speichert ein TEMPLATE nicht mehr eine Abbildung von Iterationen auf Prozessorfeldelemente, sondern eine Abbildung auf eine Menge von Prozessorfeldelementen. Die Speicherung wäre deutlich aufwendiger.

Diese Probleme lassen sich vermeiden, wenn man zu einer Darstellung übergeht, bei der jeder Prozessor in einer Liste nur diejenigen Iterationen speichert, die er selbst zu bearbeiten hat.

Bei der Wahl der Datenstruktur für ein derartiges **lokales invertiertes TEMPLATE** sind nun zwei Alternativen denkbar:

1. Es wird eine Liste angelegt, die jede zu bearbeitende Iteration einzeln enthält.

2. Es wird eine Liste angelegt, in der zusammengehörige Blöcke von Iterationen abgespeichert werden.

Bewertet man diese Alternativen hinsichtlich Speicherbedarf, Berechnungsaufwand und Eingliederung in die vorgegebene Struktur des Laufzeitsystems, so ist eindeutig die zweite Variante vorzuziehen:

1. Der Speicherbedarf für die blockweise Darstellung ist deutlich günstiger, falls die Iterationen eines Prozessors auch tatsächlich Blöcke bilden. Im Extremfall sind sogar nur zwei Zahlen für Anfang und Ende eines einzigen Blocks zu speichern. Im umgekehrten Extremfall, falls sich also nur Blöcke der Länge eins ergeben, ist der Speicherbedarf doppelt so groß wie bei der ersten Alternative, da dann als Anfang und Ende eines jeden Blocks dieselbe Zahl zweimal abgespeichert wird. Denkbar wäre in diesem Fall der Übergang auf die Einzelspeicherung von lokal durchzuführenden Iterationen. Dieser Übergang wäre ohne großen programmiertechnischen sowie zeitlichen Aufwand realisierbar, da in der Liste lediglich jedes zweite Datum gelöscht werden müßte.

2. Damit parallele Schleifen effizient auf den beteiligten Prozessoren ausgeführt werden [Plu94], ist es unbedingt erforderlich, daß aufeinanderfolgende Iterationen auch tatsächlich als ein zusammenhängender Block von einer einzigen DO–Schleife abgearbeitet werden. Hat man also das **lokale invertierte TEMPLATE** schon in blockweiser Darstellung vorliegen, so ergeben sich untere und obere Schleifengrenzen

einfach und schnell ohne jede Berechnung durch Auslesen von zwei Zahlen aus dem lokalen Speicher. Die bei der Speicherung von Einzeliterationen notwendige Konvertierung in die blockweise Darstellung ist bei der Ausführung einer parallelen Schleife zu aufwendig. Deshalb muß auch dann, wenn der Speicherbedarf der blockweisen Darstellung ungünstig ist, auf den oben erwähnten Übergang zur Speicherung von Einzeliterationen verzichtet werden.

3. Beim Anlegen eines **lokalen invertierten TEMPLATES** während der Ausführung der DISTRIBUTE–Direktive sind die auftretenden Blöcke zu berechnen. Der dabei verwendete Algorithmus ist sehr einfach und effizient (siehe Abschnitt 4.2).

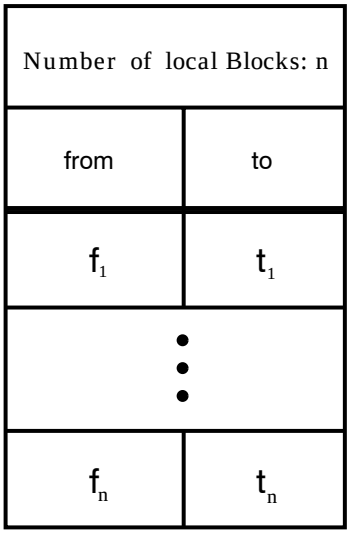


Abbildung 4.1: Die Datenstruktur "lokales invertiertes TEMPLATE"

Selbst mit dieser verteilten Art der Speicherung bleibt prinzipiell das Problem eines unter Umständen recht großen Speicherbedarfs bestehen. Falls sich nur Blöcke der Länge eins ausbilden, müssen für jede lokal auszuführende Iteration zwei Integer–Zahlen gespeichert werden. Die Zahl der Blöcke, die ein bestimmter Prozessor zu speichern hat, ergibt sich erst während der Laufzeit. Da der exakte Speicherbedarf deswegen nicht vorher bestimmt werden kann, soll hier der schlechteste Fall abgeschätzt werden. Dieser liegt dann vor, wenn ein einzelner Prozessor alle geraden (oder alle ungeraden) Iterationen ausführt. Es ergeben sich $\lceil \frac{N}{2} \rceil$ Blöcke, wobei N die Zahl aller Iterationen in der betreffenden Dimension bezeichnet. Jeder dieser Blöcke enthält zur Darstellung von unterer und oberer Grenze zwei Integer–Werte. Setzt man für einen Integer–Wert maschinenabhängig 4 Byte an, so ergibt sich ein maximaler Speicherbedarf von $4 * (N + 1)$ Byte. Bei mehrdimensionalen TEMPLATES multipliziert sich dieser Platzbedarf entsprechend des Zahl und Ausdehnung der Dimensionen.

In SVM–Fortran sind nun zwei Möglichkeiten vorgesehen, wie man statisch eine nicht–reguläre Verteilung von Schleifeniterationen auf Prozessorfelder spezifizieren kann:

1. **Die indirekte DISTRIBUTE–Direktive**
2. **Die gekoppelte DISTRIBUTE–Direktive**

Diese Direktiven sollen im folgenden näher erläutert werden.

4.2 Die indirekte DISTRIBUTE–Direktive

Die größte Flexibilität bei der Spezifikation von TEMPLATES hat der Anwender mit dieser Direktive. Er kann dabei in jeder Dimension direkt angeben, welche Iteration auf welchem Prozessor eines Prozessorfeldes ausgeführt werden soll.

```
CSVM$ PROCESSORS :: P(N,N)
CSVM$ TEMPLATE :: T(N,N)
CSVM$ REDISTRIBUTE (I,J) ONTO P(I,MAP(J)) :: T
```

In diesem Beispiel hat der Programmierer die Verteilung eines zweidimensionalen Iterationsraumes T auf ein zweidimensionales Prozessorfeld P angegeben. In der ersten Dimension soll jede Iteration I auf dem Prozessor, dessen Position im Prozessorfeld P in der ersten Dimension gerade I ist, zugeordnet werden. Die zweite Dimension des Iterationsraumes wird auf die zweite Dimension des Prozessorfeldes P verteilt. Die Verteilung wird durch das Mapping–Array MAP bestimmt. Eine Iteration J soll auf dem Prozessor ausgeführt werden, dessen Position im Prozessorfeld P in der zweiten Dimension gerade MAP(J) ist.

Die Sprachdefinition von SVM–Fortran erlaubt bei der Spezifikation der Indizes des Zielprozessorfeldes (im Beispiel also I bzw. MAP(J)) beliebige Integer–Ausdrücke, so daß insbesondere Konstanten, Funktionen und Speicherreferenzen zulässig sind.

4.2.1 Zwischen–Code für die indirekte DISTRIBUTE–Direktive

Eine derartige DISTRIBUTE–Direktive ist syntaktisch für den Compiler zunächst nichts anderes als ein Kommentar. Sie muß deshalb in ausführbaren Programm–Code übersetzt werden. Die Übersetzung eines SVM–Fortran Anwendungsprogramms erfolgt, wie in Abschnitt 3.2 beschrieben, in drei Schritten. Zuerst werden von dem Präprozessor **svmf** die Direktiven in Fortran 77–Code sowie in Aufrufe von neuen Laufzeitfunktionen übersetzt. Dieser Code wird dann konventionell übersetzt. Schließlich werden neue Laufzeitfunktionen, die den parallelen Programmfluß steuern, hinzugelinkt.

Aus Gründen der Effizienz und der Übersichtlichkeit des erzeugten Zwischen–Codes wird dabei eine möglichst geringe Veränderung des ursprünglichen Fortran–Programmtextes angestrebt. Zusätzliche Verwaltungsarbeit wird im wesentlichen von den Laufzeitfunktionen geleistet, so daß der Zwischen–Code nur um entsprechende Aufrufe erweitert werden muß.

Bei der Expansion der DISTRIBUTE–Direktive im nicht–regulären Fall scheidet diese Methode aus, da die Integer–Ausdrücke, die die Verteilung steuern, in Fortran–Syntax gegeben sind. Darüber hinaus greifen sie auf Daten zu, die ausschließlich im Fortran–Kontext zur Verfügung stehen. Dies erfordert, daß zusätzlicher Programmtext überwiegend als Fortran–Code zu generieren ist.

4.2.1.1 Zwischen–Code für den Standardfall

Effiziente Lösungen von Programmieraufgaben beruhen auf einer geeigneten Wahl von aufeinander abgestimmten Algorithmen und Datenstrukturen. Ein auf die oben diskutierte

Datenstruktur **'lokales invertiertes TEMPLATE'** abgestimmter Algorithmus ist dadurch gegeben, daß dimensionsweise alle Prozessoren, auf die Iterationen verteilt werden sollen, den Iterationsbereich vollständig durchlaufen. Jeder Integer–Ausdruck, der von der Laufvariablen abhängt, wird von jedem Prozessor ausgewertet. Falls die Nummer, die der lokale Prozessor in dieser Dimension innerhalb des Prozessorfeldes hat, mit dem Wert des Integer–Ausdrucks übereinstimmt, so soll die gerade untersuchte Iteration auf diesem Prozessor ausgeführt werden. Die Laufvariable wird folglich in das **lokale invertierte TEMPLATE** eingefügt. In Abbildung 4.2 ist der vom **svmf**-Compiler generierte Zwischen–Code zur Umsetzung der indirekten Verteilung anhand eines Beispiels dargestellt.

Der Vorgang dieses Einfügens ist recht einfach, da sichergestellt ist, daß Iterationen in streng aufsteigender Ordnung hinzugefügt werden: Es wird also lediglich überprüft, ob eine neu hinzukommende Iteration den letzten Block um eins vergrößert oder ob ein neuer Block angefangen werden muß. Da nicht a priori feststehen kann, wieviele Blöcke ein bestimmter Prozessor speichern muß, wurde eine dynamische Vergrößerung der lokalen Liste vorgesehen. Um zu vermeiden, daß dies jedesmal geschieht, wenn ein neuer Block angefangen werden muß, wird der Vergrößerungsschritt in Abhängigkeit von der Größe des TEMPLATES in dieser Dimension und von der Zahl der partizipierenden Prozessoren gewählt.

4.2.1.2 Optimierung für Sonderfälle

Bei der Expansion der DISTRIBUTE–Direktive für nicht–reguläre Verteilungen lassen sich drei Sonderfälle effizienter realisieren:

1. Der Integer–Ausdruck zur Spezifikation des Zielprozessorfeldes ist eine Konstante.
   Bsp.: DISTRIBUTE (I) ONTO P(2)

2. Die Verteilung soll auf alle Prozessoren in einer Dimension repliziert werden, d.h. alle Prozessoren in der betreffenden Dimension sollen den Programm-Code redundant ausführen.
   Bsp.: DISTRIBUTE (I) ONTO P(*)

3. Die Verteilung soll auf einen Bereich einer Dimension eines Prozessorfeldes repliziert werden.
   Bsp.: DISTRIBUTE (I) ONTO P(2:6)

In diesen Fällen tritt die Laufvariable I bei der Spezifikation des Zielprozessorfeldes nicht auf. Die Semantik der DISTRIBUTE–Direktive legt dann fest, daß das TEMPLATE nicht verteilt, sondern sequentialisiert wird. Es werden alle TEMPLATE–Elemente als ein zusammenhängender Block allen Prozessoren zugeordnet, die bei der Spezifikation des Zielprozessorfeldes angegeben sind. Es braucht also in diesen Fällen nicht mehr das gesamte TEMPLATE in der bearbeiteten Dimension durchlaufen zu werden; stattdessen wird geprüft, ob der lokale Prozessor einen solchen vollständigen Iterationsblock erhält. Ist dies der Fall, so kann dies schon bei der Initialisierung des TEMPLATES vermerkt werden.

```
 1:CSVM$ PROCESSORS :: P(2)
 2:CSVM$ TEMPLATE   :: T(5000)
 3:CSVM$ DISTRIBUTE (I) ONTO P(IFF(I)) :: T

 4:      CALL SVM_BARRIER(2,PSET_GET_APS(),1)
 5:      I_AM_MEMBER = I_PROC_TO_INDEX (1, MYPROC(), PROC_INDEX_FIELD)
 6:      CALL SVM_READ_PROC_DIMENSIONS(1,PROC_DIMENSION,RANK)
 7:      SVM_ITER_APS_FIELD(1,1) = PROC_INDEX_FIELD(1)
 8:      SVM_ITER_APS_FIELD(1,2) = PROC_INDEX_FIELD(1)
 9:      SVM_DISTR_DEST_FIELD(1) = 1
10:      CALL SVM_INIT_LOCAL_INV_TEMPL_INDIR
11:     +      (1, 1, 1, I_AM_MEMBER, SVM_DISTR_DEST_FIELD, PPROC_
12:     +       DIMENSION, SVM_ITER_APS_FIELD, PROC_INDEX_FIELD, RANK)
13:      IF(I_AM_MEMBER .EQ. 1) THEN
14:         CALL SVM_MAKE_MYPROC_TO_APS()
15:         CALL SVM_READ_TEMPL_BOUNDS(1,DEST_TEMPL_DIMENSION,RANK)
16:         DO 100, I_0000 = DEST_TEMPL_DIMENSION(1,1),
17:     +                    DEST_TEMPL_DIMENSION(1,2)
18:            INT_EXPR_VAL = IFF(I_0000)
19:            IF ( (INT_EXPR_VAL .LT. PROC_DIMENSION(1,1)) .OR.
20:     +           INT_EXPR_VAL .GT. PROC_DIMENSION(1,2))) THEN
21:                 IF (MYPROCSET() .EQ. 0) THEN
22:                    CALL SVM_ERROR_REPORT ('...')
23:                 ENDIF
24:            ELSEIF (INT_EXPR_VAL .EQ. PROC_INDEX_FIELD(1)) THEN
25:              CALL SVM_ADD_TO_LOCAL_INV_TEMPL (I_0000, 1, 1)
26:            ENDIF
27: 100     CONTINUE
28:         CALL SVM_PSET_RESET_APS()
29:      ENDIF
30:      CALL SVM_BARRIER(3,PSET_GET_APS(),1)
```

Abbildung 4.2: Expansion der Direktive "INDIRECT DISTRIBUTION"

4.3 Die gekoppelte DISTRIBUTE–Direktive

Hat man schon eine Verteilung von Schleifeniterationen auf ein Prozessorfeld in einem TEMPLATE gespeichert und möchte auf dieselben Daten in einer anderen Ordnung zugreifen, so läßt sich die Lokalität beim Datenzugriff dadurch erhalten, daß man das Scheduling der neuen Iterationen abhängig von dem alten TEMPLATE und von dem Zugriffsmuster organisiert. Ein Beispiel hierfür ist ein Mehrgitterverfahren, bei dem man nicht nur auf das gesamte Feld von Gitterpunkten, sondern bei der Vergröberung des Gitters auch auf einen Teil der Gitterpunkte mit einem gewissen Abstand zugreifen muß. Bei dem in Abbildung 4.3 dargestellten Beispiel ist der vollständige Iterationsraum blockweise auf drei Prozessoren verteilt. Jeder Prozessor erhält also genau 5 Iterationen. Die zweite DISTRIBUTE–Anweisung führt in dieser Situation dazu, daß die Iteration I des Teiliterationsraumes T2 genau dort ausgeführt wird, wo vorher schon die Iteration $3 * I$ des gesamten Iterationsraumes T1 ausgeführt wurde. Dies führt dazu, daß jeder Prozessor nur auf Daten zugreift, auf die er zuvor schon einmal zugegriffen hat.

Die einzelnen Zeilen haben dabei die folgende Bedeutung:

1 : Definition eines eindimensionalen Prozessorfeldes der Ausdehnung 2.
2 : Definition eines eindimensionalen TEMPLATES der Ausdehnung 5000.
3 : Indirekte Verteilung, deren Expansion in Zeile 4 bis 30 erfolgt.

4 : Da eventuell zuvor gemeinsame Variablen initialisiert werden, muß auf das Eintreffen aller Prozessoren gewartet werden.
5 : Bestimmung der eigenen Position innerhalb des Prozessorfeldes P.
6 : Bestimmung der Prozessorfeldgrenzen.
7 - 8 : Information, welche Prozessoren eine Schleifeniteration gemeinsam bearbeiten.
9 : Die 1. TEMPLATE–Dimension ist auf die 1. Prozessorfeld–Dimension verteilt.
10 : Initialisierung des 1. TEMPLATES in Dimension 1.
13 : Nur diejenigen Prozessoren durchlaufen die Zeilen 14 bis 28, welche in dem Teil des Prozessorfeldes P, auf das verteilt wird, enthalten sind.
14 : Da der folgende Aufruf der Funktion IFF nicht synchronisiert werden muß, bildet jeder Prozessor für sich allein ein APS.
15 : Bestimmung der TEMPLATE–Grenzen.
16 : Es wird die gesamte TEMPLATE–Dimension durchlaufen.
18 : Auswertung des Integer–Ausdrucks.
19 - 23 : Falls sich der in Zeile 18 berechnete Wert außerhalb der Prozessorfeldgrenzen befindet, so liegt ein Fehler vor.
24 - 26 : Falls der in Zeile 18 berechnete Wert mit der Nummer des lokalen Prozessors innerhalb des Prozessorfeldes übereinstimmt, so wird die Laufvariable zum TEMPLATE T hinzugefügt.
28 : Wiederherstellung des ursprünglichen APS.
30 : Da die nachfolgende Programmausführung eventuell gemeinsame Variablen verändern kann, die in dem Integer–Ausdruck auftreten, muß hier wieder auf das Eintreffen aller Prozessoren gewartet werden.

```
CSVM$ PROCESSORS :: P(3)
CSVM$ TEMPLATE :: T1(15), T2(5)
CSVM$ DISTRIBUTE (BLOCK) ONTO P :: T1
CSVM$ DISTRIBUTE (I) ONTO HOME (T1(3*I)) :: T2
CSVM$ PDO (I) ONTO HOME (T1(I))
      DO I=1,15
         ... = A(I)
      ENDDO ...
CSVM$ PDO (I) ONTO HOME (T2(I))
      DO I=1,5
         ... = A(3*I)
      ENDDO
```

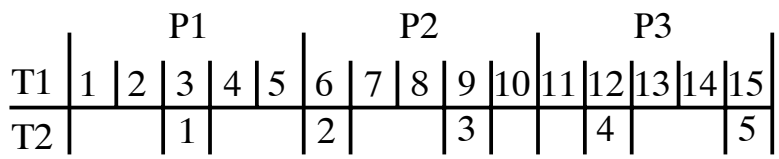


Abbildung 4.3: Iterationsverteilung beim Zugriff auf ein Teilfeld

```
 1:CSVM$ PROCESSORS :: P(3)
 2:CSVM$ TEMPLATE   :: T1(15),T2(5)
 3:CSVM$ DISTRIBUTE (I) ONTO HOME(T1(3*I))::T2

 4:      CALL SVM_BARRIER(3,PSET_GET_APS(),1)
 5:      I_AM_MEMBER = I_SVM_CHECK_IF_TEMPL_IS_DEFINED(1)
 6:      SVM_DISTR_DEST_FIELD(1) = 1
 7:      SVM_TEMPL_ALIGN_SPEC_LIST(1) = 0
 8:      CALL SVM_INIT_LOCAL_INV_TEMPL_LINKED(2, 1, 2, I_AM_MEMBER,
 9:     +         SVM_DISTR_DEST_FIELD, SVM_TEMPL_ALIGN_SPEC_LIST,
10:     +         SVM_DEST_DIM_IS_SEQUENT)
11:      IF(I_AM_MEMBER .EQ. 1) THEN
12:        CALL SVM_MAKE_MYPROC_TO_APS()
13:        CALL SVM_READ_TEMPL_BOUNDS(1, SOURCE_TEMPL_DIMENSION, RANK)
14:        CALL SVM_READ_TEMPL_BOUNDS(2, DEST_TEMPL_DIMENSION, RANK)
15:        IF (SVM_DEST_DIM_IS_SEQUENT(1) .NE. 1) THEN
16:           DO 100, I_0000 = DEST_TEMPL_DIMENSION(1,1),
17:     +                      DEST_TEMPL_DIMENSION(1,2)
18:             INT_EXPR_VAL = 3*I_0000
19:             IF ( (INT_EXPR_VAL .LT. SOURCE_TEMPL_DIMENSION(1,1)) .OR.
20:     +            (INT_EXPR_VAL .GT. SOURCE_TEMPL_DIMENSION(1,2))) THEN
21:                IF (MYPROCSET() .EQ. 0) THEN
22:                   CALL SVM_ERROR_REPORT ('...')
23:                ENDIF
24:             ELSEIF (I_NUM_IS_ON_MYPROC (1,1,INT_EXPR_VAL).NE.0) THEN
25:               CALL SVM_ADD_TO_LOCAL_INV_TEMPL (I_0000, 2, 1)
26:             ENDIF
27: 100       CONTINUE
28:        ENDIF
29:        CALL SVM_PSET_RESET_APS()
30:      ENDIF
31:      CALL SVM_BARRIER(5,PSET_GET_APS(),1)
```

Abbildung 4.4: Expansion der Direktive "LINKED DISTRIBUTION"

4.3.1 Zwischen–Code für gekoppelte Verteilungen

Der Zwischen–Code für die DISTRIBUTE–Direktive, die nach einem zuvor schon verteilten TEMPLATE ausgerichtet wird (**linked DISTRIBUTION**), unterscheidet sich von dem der indirekten Verteilung im wesentlichen nur durch das Kriterium für die Annahme einer Iteration. Dieses Kriterium ist nicht mehr, daß der Wert des Integer–Ausdrucks mit der Adresse des lokalen Prozessors innerhalb des Prozessorfeldes übereinstimmt. Stattdessen wird eine Iteration dann in das neue **lokale invertierte TEMPLATE** übernommen, wenn die von dem Integer–Ausdruck spezifizierte Iteration auf dem TEMPLATE, nach dem verteilt werden soll, schon lokal vorhanden ist. In Abbildung 4.4 ist der vom **svmf**-Compiler generierte Zwischen–Code, der die gekoppelte Verteilung umsetzt, an einem Beispiel dargestellt.

Die einzelnen Zeilen haben dabei die folgende Bedeutung:

1 : Definition eines eindimensionalen Prozessorfeldes der Ausdehnung 2.
2 : Definition eines eindimensionalen TEMPLATES der Ausdehnung 5000.
3 : Gekoppelte Verteilung, deren Expansion in Zeile 4 bis 31 dargestellt ist.

4 : Da eventuell zuvor gemeinsame Variablen initialisiert werden, muß auf das Eintreffen aller Prozessoren gewartet werden.
5 : Nur auf diejenigen Prozessoren, auf die das Quell–TEMPLATE verteilt wurde, wird auch das Ziel–TEMPLATE verteilt.
6 : Speicherung der Quell–TEMPLATE–Dimension, nach der verteilt wird. = 0 bedeutet, daß nicht verteilt, sondern sequentialisiert wird.
7 : Speichert Angaben der template_align_spec_list: 0 $\Leftrightarrow$ Integer–Ausdruck; -1 $\Leftrightarrow$ Stern; J $\Leftrightarrow$ Konstante = J.
8 : Initialisierung des 2. TEMPLATES in Dimension 1.
9 : Nur diejenigen Prozessoren durchlaufen die Zeilen 12 bis 28, welche in dem Teil des Prozessorfeldes P enthalten sind, auf das verteilt wird.
12 : Da der folgende Aufruf der Funktion IFF nicht synchronisiert werden muß, bildet jeder Prozessor für sich allein ein APS.
13 : Bestimmung der Quell–TEMPLATE–Grenzen.
14 : Bestimmung der Ziel–TEMPLATE–Grenzen.
15 : Nur falls die betrachtete Dimension nicht sequentialisiert ist, muß verteilt werden.
16 : Es wird die gesamte Ziel–TEMPLATE–Dimension durchlaufen.
18 : Auswertung des Integer–Ausdrucks.
19 - 23 : Falls sich der in Zeile 18 berechnete Index außerhalb der Grenzen des Quell-TEMPLATES befindet, so liegt ein Fehler vor.
24 - 26 : Falls der in Zeile 18 berechnete Index in dem Quell–TEMPLATE auf dem lokalen Prozessor vorhanden ist, so wird die Laufvariable zum Ziel–TEMPLATE auf dem lokalen Prozessor hinzugefügt.
29 : Wiederherstellung des ursprünglichen APS.
30 : Da die nachfolgende Programmausführung eventuell gemeinsame Variablen verändern kann, die in dem Integer–Ausdruck auftreten, muß hier wieder auf das Eintreffen aller Prozessoren gewartet werden.

Bei dieser Art der Verteilung ist ein weiterer Punkt zu beachten: Lag die gegebene Verteilung, an der die neue auszurichten ist, in einer regulären Darstellung vor und ist die neue Art der Verteilung lediglich eine Kopie der alten (z.B. bei DISTRIBUTE (I) ONTO HOME (T1(I)) :: T2), so wird dies erkannt. Auch die Verteilung des neuen TEMPLATES kann so wieder in der regulären und damit speicherplatzsparenden Art registriert werden. Dies bringt zusätzlich den Vorteil eines effizienteren Ablaufes mit sich, da nicht mehr die gesamte TEMPLATE–Dimension durchlaufen werden muß. Stattdessen genügt es, einmal einige Werte in die lokale Datenstruktur für das Ziel–TEMPLATE einzutragen.

Auch wenn die ursprüngliche Verteilung selbst nicht regulär gewesen ist, führt dieser Kopiermechanismus zu einer Verkürzung der Ausführungszeit.

4.4 Gemessene Ausführungszeiten

Nachdem in den beiden vorangegangenen Abschnitten die Realisierungen der indirekten und der gekoppelten Verteilung dargelegt wurden, wird nun beschrieben, welche Ausführungszeiten sich dabei ergeben.

Den Beschreibungen der internen Abläufe in Abbildung 4.2 sowie in Abbildung 4.4 kann man entnehmen, daß sich die Ausführungszeiten der DISTRIBUTE-Direktiven aus drei Anteilen zusammensetzen. Die Initialisierung weist eine konstante Laufzeit auf. Die Laufzeit der beiden Barriere-Synchronisationen hängt logarithmisch (vergleiche Abbildung 4.5) von der Zahl P der beteiligten Prozessoren ab.

Die dritte Komponente ist der eigentliche Verteilungsvorgang, dessen Laufzeit linear mit der Größe des zu verteilenden Iterationsraumes wächst. Welchen Zeitbedarf dabei die Verteilung einer einzelnen Iteration mit sich bringt, hängt stark von der Form des Integer–Ausdrucks ab, der schließlich die Verteilung steuert. In den Zeilen zwei bis vier in Tabelle 4.1 sind die Werte für drei typische Integer–Ausdrücke angegeben. Die Werte in Zeile drei ergeben sich dabei nur, falls das Mapping–Array MAP auf jedem Prozessor privat vorliegt. Falls dies wegen seiner Größe nicht mehr möglich ist, werden sich zusätzlich bei jedem Prozessor P–1 Read-Page-Faults ergeben. Der Meßwert in Zeile sechs hängt praktisch nicht von der Größe des zu kopierenden Datenbereiches ab.

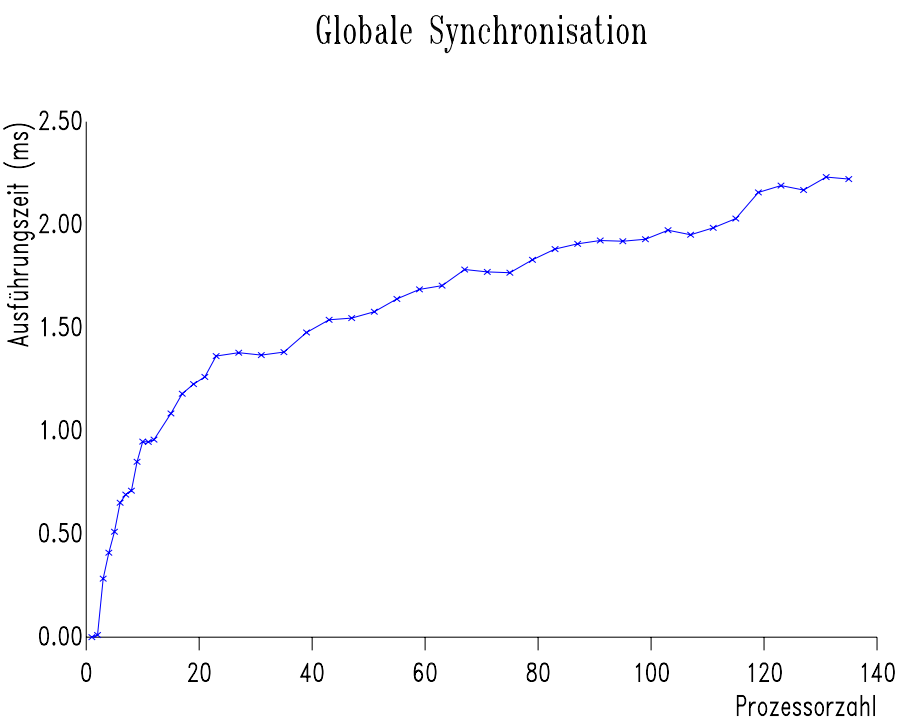


Abbildung 4.5: Gemessene Ausführungszeit einer Barriere-Synchronisation

Stellt man diese Laufzeiten denen im regulären Fall (vergleiche [Plu94]) gegenüber, so ist das Verhalten beider Verteilungsanweisungen bei der Ausführung der Initialisierung

Operation	indirekte Verteilung	gekoppelte Verteilung
Initialisierung	100 μs	117 μs
DISTRIBUTE (I) ONTO ... $\alpha I + \beta$[1]	1,5 μs	6 μs
DISTRIBUTE (I) ONTO ... MAP(I)[1]	1,4 μs	6 μs
DISTRIBUTE (I) ONTO ... FKT(I)[1]	12.5 μs	17 μs
Kopierung einer regelmäßigen Verteilung	/	8,5 μs
Kopierung einer nicht regelmäßigen Verteilung	/	19 μs

Tabelle 4.1: Ausführungszeiten für Bestandteile der DISTRIBUTE-Direktive

und der beiden BARRIER im wesentlichen gleich. Der Unterschied besteht darin, daß im nicht–regulären Fall der Lauf über den Iterationsraum hinzukommt.

4.5 Mehrdimensionale nicht–regulär verteilte TEMPLATES

Bei der Implementierung der Datenstruktur **lokales invertiertes TEMPLATE** hat sich herausgestellt, daß es einen sehr großen Aufwand mit sich bringen würde, wenn man bei der Distribution–Spezifikation für das Zielprozessorfeld (indirect–distribution) bzw. bei der TEMPLATE–Align–Spezifikation für das TEMPLATE–Element, nach dem neu verteilt werden soll (linked–distribution), allgemeine Integer–Ausdrücke zuläßt.
Ein Beispiel für die Dimension 2:

```
Indirect–Distribution :
CSVM$ DISTRIBUTE (I,J) ONTO P(F(I,J), G(I,J)) :: T)
Linked–Distribution :
CSVM$ DISTRIBUTE (I,J) ONTO HOME (T1(F(I,J), G(I,J))) :: T2
```

Falls in diesem Beispiel die Funktionen F und G sowohl von I als auch von J abhängen, so hängen damit auch die Iterationen, die in einer Dimension auszuführen sind, von dem Wert ab, der gerade in der anderen Dimension angenommen wird. Wollte man in dieser Situation analog zu der oben beschriebenen Methode Iterationsblöcke lokal in Integer–Feldern speichern, so müßte man für jeden Wert, den I annehmen kann, ein eigenes Feld von Integer–Blöcken für die Iterationen in J bereitstellen. Selbst wenn man I–Werte, die zu denselben Iterationen in J führen, zusammenfassen würde, wäre der Speicherbedarf weit jenseits realistischer Größenordnungen, insbesondere bei höherdimensionalen TEMPLATES. Das Problem wird zusätzlich dadurch verschärft, daß nicht festgelegt ist, ob die parallelen Schleifen, die später nach diesen TEMPLATES ausgeführt werden sollen, auch dieselbe Schachtelungsreihenfolge einhalten müssen. Da TEMPLATES in SVM-Fortran maximal 7-dimensional spezifiziert werden können, müßte für jede der im Extremfall $7! = 5040$ Permutationen der ohnehin schon unrealistisch große Speicherplatz bereitgestellt werden.

[1] Diese Zeitangaben gelten pro TEMPLATE–Element

Denkbar sind in diesem Fall zwei andere Lösungen: Bei der ersten wird völlig auf eine Speicherung von lokal zu bearbeitenden Schleifen–Iterationen verzichtet. Stattdessen werden diese erst während der Ausführung der Schleife bestimmt. Jeder beteiligte Prozessor durchläuft dazu den geschachtelten Indexraum vollständig und wertet zu jeder so erzeugten Kombination von Indizes die vorgegebenen Verteilungsfunktion aus. Stimmt das Ergebnis dieser Auswertung dann mit der eigenen Adresse innerhalb des Prozessorfeldes überein, so ist das gerade untersuchte Indextupel bei ihm lokal auszuführen. Diese Verschiebung des Ablaufes, der ursprünglich für die DISTRIBUTE–Direktive gedacht war, bedeutet aber eine Sequentialisierung der eigentlich verteilten Schleife. Eine signifikante Reduktion der Ausführungszeit ist so nicht mehr zu erwarten.

Bei dem zweiten Lösungsansatz wird auf der Menge der möglichen Indexvektoren eine Ordnung definiert. Denkbar ist hierbei die Fortran-Abspeicherungsreihenfolge für mehrdimensionale Arrays. Man bildet also jedes auftretende Tupel von Indizes $(I_1, I_2, \ldots, I_n)$ auf einen Skalar I ab. Eine Darstellung, bei der jedes I für einen Punkt im Produktraum steht, gestattet die Speicherung von beliebigen Verteilungen. Wieder speichert man Blöcke von lokal zu bearbeitenden Indizes. Dieses Verfahren hat allerdings zwei entscheidende Nachteile. Erstens ist der Speicherbedarf unter Umständen enorm, da eine eventuell teilweise vorhandene Blockstruktur nur für Iterationen in der innersten Ebene zu einer Reduktion der Datenmenge führen kann. Zweitens muß zur Laufzeit, für jeden linearisierten Indexblock getrennt, die genannte Abbildung invertiert werden, um wieder Iterationsblöcke für jede einzelne Laufvariable zu erhalten.

Da für das Problem, bei der die Verteilung in einer Dimension von der in anderen Dimensionen abhängt, keine befriedigende Lösung gefunden werden konnte, ist man innerhalb des SVM–Fortran–Projektes übereingekommen, diesen Fall auszuschließen.

Damit kann zusammenfassend gesagt werden:

> Die DISTRIBUTE–Direktive verteilt voneinander unabhängige Dimensionen.

Dies hat Konsequenzen für die Semantik paralleler Schleifen, die über ein mehrdimensionales TEMPLATE verteilt werden.

Die Anweisungssequenz :

```
CSVM$ PROCESSORS :: P(3,3)
CSVM$ TEMPLATE :: T(3,3)
CSVM$ DISTRIBUTE (I,J) ONTO P(F(I), F(J)) :: T

      FUNCTION F(I)
      INTEGER F,I
      F = 2
      END

CSVM$ PDO (LOOPS(I), STRATEGY(ON_HOME (T(I))))
      DO I = 1, N
         ....
      ENDDO
```

führt nicht direkt dazu, daß nur der Prozessor P(2,2) alle 9 Iterationen zugewiesen bekommt, sondern dazu, daß alle I–Iterationen allen Prozessoren, die in der 1. Dimension

den Index 2 aufweisen, zugeordnet werden, und daß alle J–Iterationen allen Prozessoren, die in der 2. Dimension den Index 2 aufweisen, zugeordnet werden (siehe Abbildung 4.6). Insgesamt ist also Prozessor P(2,2) der einzige, der sowohl I- als auch J-Iterationen erhält.

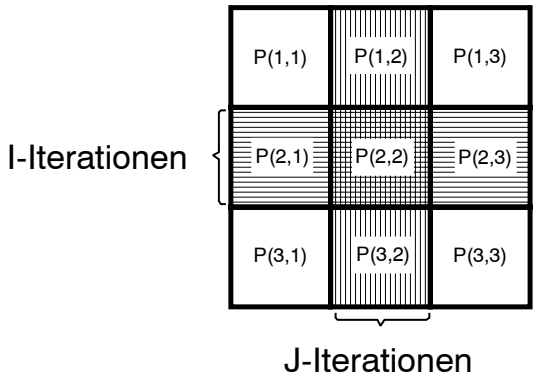


Abbildung 4.6: DISTRIBUTE (I,J) ONTO P(F(I), F(J)), wobei $F \equiv 2$ ist

Im dargestellten Beispiel tritt somit noch kein Unterschied zu einer Verteilung auf, bei der eine Abhängigkeit zwischen den Dimensionen besteht.

Ein unterschiedliches Verhalten stellt aber sich ein, wenn bei der Ausführung einer parallelen Schleife die Verteilung in einer TEMPLATE–Dimension nicht berücksichtigt und stattdessen die Arbeit auf alle Prozessorfeldelemente in dieser Dimension repliziert wird. Als Beispiel:

```
CSVM$ PROCESSERORS :: P(3,3)
CSVM$ TEMPLATE :: T(3,3)
CSVM$ DISTRIBUTE (I,J) ONTO P(F(I), F(J)) :: T
      FUNCTION F(I)
      INTEGER F,I
      F = 2
      END
CSVM$ PDO (LOOPS(I), STRATEGY(ON_HOME (T(I,*))))
      DO I = 1, N
         ....
      ENDDO
```

Da die Verteilungen in den beiden Dimensionen unabhängig voneinander sind, werden in diesem Beispiel die Iterationen (I=1,2,3) auf allen Prozessorfeldelementen P(I=2, J=1,2,3) ausgeführt. Anderenfalls würde hier nur P(I=2, J=2) die Iterationen (I=1,2,3) bearbeiten. Man erkennt an diesem Beispiel darüber hinaus, daß man auf eine Optimierung verzichten muß, bei der nur Verteilungen gespeichert bleiben, die in jeder Dimension Einträge aufweisen.

Damit steht ein Mechanismus zur Speicherung nicht–regulärer Verteilungen von Schleifeniterationen auf Prozessoren zur Verfügung. Die Voraussetzung, um die sich bei einem dynamischen Lastausgleichsverfahren ergebende Verteilung speichern zu können, ist somit gegeben.

5 Dynamische Lastbalancierungsverfahren

Die übliche Vorstellung ist, daß in der Parallelverarbeitung der Einsatz von P Prozessoren die Lösung eines beliebigen Problems unmittelbar um den Faktor P beschleunigt. Die Erfahrung lehrt jedoch, daß dieser optimale Beschleunigungsfaktor P oft nur annähernd und zumeist erst nach erheblicher Umstrukturierung des Programms, häufig aber auch gar nicht erreicht werden kann. Sucht man nach dem Grund dieser unbefriedigenden Situation, so stellt man fest, daß die maximal mögliche Beschleunigung nur dann erreicht werden kann, wenn die Gesamtmenge der zu leistenden Berechnungsarbeit absolut gleichmäßig auf alle partizipierenden Prozessoren verteilt wird. Gelingt dies nicht, so können dafür sowohl prinzipielle als auch realisierungstechnische Probleme verantwortlich sein.

Das prinzipielle Problem besteht darin, daß in jedem Programm immer ein Anteil verbleibt, der nur sequentiell ausführbar ist. Selbst wenn der parallelisierbare Anteil sehr nahe bei 1 liegen sollte, sind aufgrund von Amdahl's Gesetz [Amd67] feste Grenzen für die maximal mögliche Beschleunigung vorgegeben.

Das realisierungstechnische Problem besteht darin, daß eine absolut gleichmäßige Partitionierung der zu leistenden Gesamtarbeit nicht immer leicht möglich ist. Bei dieser Partitionierung unterscheidet man zwei Typen:

1. **Funktionale Partitionierung:** Die Einteilung erfolgt in logisch zusammengehörige, voneinander unabhängige Funktionsblöcke. Sie ist deshalb meist auf der Ebene von Unterprogrammen realisiert. Es ergibt sich dabei eine grobgranulare Parallelität, weil jeweils ganze Unterprogramme einem Prozessor zur Ausführung zugewiesen werden. Diese Zuteilung wird im Betriebssystem vom Task-Scheduler vorgenommen (siehe [Nag93]). Aufgrund der groben Granularität führt dieser Typ der Partitionierung nur in wenigen Spezialfällen zu der erstrebten Gleichverteilung.

2. **Datenbezogene Partitionierung:** Große homogene Datenfelder sind in technisch-wissenschaftlichen Anwendungen typisch. Bei diesem Typ der Partitionierung wird die Bearbeitung dieser Felder aufgeteilt. Die Realisierung erfolgt auf Schleifenebene. Es ergibt sich feingranulare Parallelität, weil einzelne Iterationen einer Schleife, also kleine Arbeitspakete, verteilt werden können. Diese Verteilung wird vom Thread-Scheduler (siehe [Nag93]) geleistet.

Beide Typen sind wichtig. Da jedoch die datenbezogene Arbeitsverteilung in technisch-wissenschaftlichen Anwendungsprogrammen dominiert, und da im allgemeinen nur sie zu einer gleichmäßigen Arbeitsverteilung führen kann, wird in der vorliegenden Arbeit in

erster Linie die Partitionierung von parallelen Schleifen untersucht. Im folgenden sollen also mit zu verteilenden Arbeitspaketen in erster Linie Iterationen einer parallelen Schleife gemeint sein.

Soll eine Schleife parallelisiert werden, so besteht der erste Ansatz darin, jedem Prozessor gleich viele Iterationen zuzuordnen. Dieser Ansatz ist leicht zu realisieren und leistet in vielen Fällen auch das Gewünschte.

Eine gleichmäßige Lastverteilung auf Schleifenebene läßt sich aber nicht immer so einfach realisieren. Der Gleichverteilungsansatz für Iterationen führt nur dann zu einer optimalen Verteilung der Arbeitslast, wenn die Bearbeitung aller Iterationen gleich viel Zeit in Anspruch nimmt und wenn alle Prozessoren, die an der Bearbeitung der Schleife teilnehmen sollen, die ganze Zeit zur Verfügung stehen. Unter den folgenden Bedingungen kann man davon nicht ausgehen:

1. Der Berechnungsaufwand einer einzelnen Iteration ist während der Übersetzungszeit nicht bekannt. Hat er sich aber zur Laufzeit einmal ergeben, so bleibt er während des restlichen Programmablaufs konstant.

2. Der Berechnungsaufwand einer einzelnen Schleifeniteration ist weder zur Übersetzungszeit bekannt noch zur Laufzeit konstant.

3. Es ist nicht garantiert, daß alle Prozessoren, die an der Ausführung einer parallelen Schleife teilnehmen sollen, gleichzeitig zur Verfügung stehen.

4. Prozessoren, die an der Ausführung einer parallelen Schleife teilnehmen, werden vom Betriebssystem in einer Multiprogramming-Umgebung der Schleifenbearbeitung für eine gewisse Zeit entzogen.

Die beiden ersten Punkte beschreiben aus Sicht der Programmiersprache, wann eine dynamische Lastadaption notwendig ist. Anschaulicher läßt sich diese Notwendigkeit aber begründen, wenn man die zu lösenden Probleme betrachtet.

Ein Beispiel ist die Simulation von Strömungsvorgängen. Abbildung 5.1 zeigt die Diskretisierung des Raumes um ein Flügelprofil. Bei der Umströmung eines Flügels besteht besonders an den Kanten die Möglichkeit, daß sich Wirbel ausbilden. Um diese Wirbel numerisch erfassen zu können, ist es erforderlich, daß an den entsprechenden Stellen das Diskretisierungsgitter verfeinert wird. Diese Verfeinerung kann

1. statisch erfolgen: Der Vorteil einer statischen Verfeinerung ist, daß sie im Vergleich zur dynamischen Verfeinerung einfacher zu realisieren ist. Sie reicht in vielen Fällen aus, da aufgrund der Flügelform ungefähr klar ist, wo sich kritische Stellen befinden.

2. dynamisch erfolgen: Der Vorteil einer dynamischen Verfeinerung ist, daß sie nur genau dort erfolgen muß, wo sie auch tatsächlich notwendig ist. Einem geringeren Berechnungsaufwand steht also der Nachteil eines aufwendigeren Verfahrens gegenüber.

Ist eine dynamische Verfeinerung realisiert, so ergibt sich je nach Grad der Verfeinerung ein unterschiedlich hoher Berechnungsaufwand in verschiedenen Regionen. Damit dennoch die gesamte Arbeit gleichmäßig auf alle beteiligten Prozessoren verteilt werden kann, darf die Zuordnung einzelner Regionen zu Prozessoren erst zur Laufzeit erfolgen. Typisch ist an diesem Beispiel die Notwendigkeit zur Anpassung des Programmablaufs an die Dynamik des betrachteten Problems.

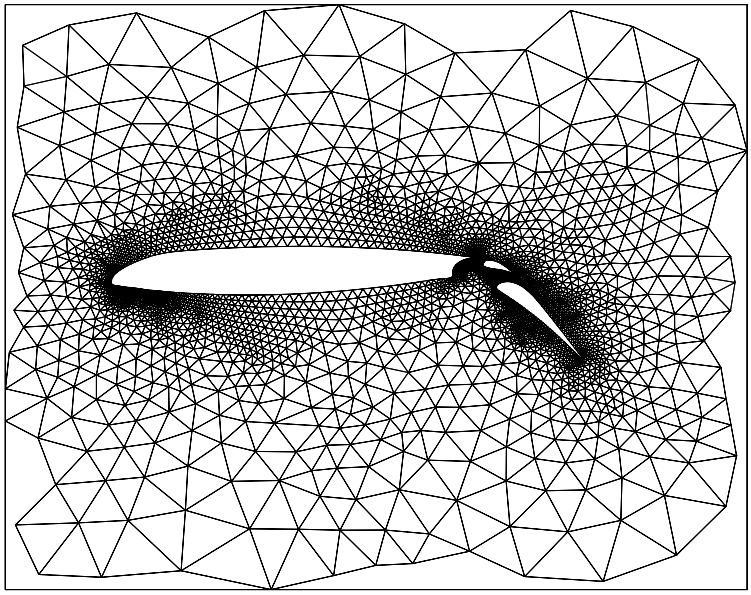


Abbildung 5.1: Gitterverfeinerung bei kritischen Punkten eines Flügelprofils

5.1 Das Prinzip des Self-Service-Scheduling

Möchte man dynamische Lastausgleichsverfahren entwickeln, so ist aus den genannten Gründen die Ebene von parallelen Schleifen besonders wichtig. In Rechnern, die physikalisch über einen gemeinsamen Speicher verfügen, werden solche Verfahren heute schon mit Erfolg eingesetzt (siehe [BePo89, HSF92]). Bei diesen erfolgt die Zuordnung von Schleifeniterationen auf Prozessoren nicht mehr statisch. Sie ergibt sich vielmehr aufgrund der tatsächlich vorgefundenen Arbeitslastverteilung erst dynamisch zur Laufzeit. Alle diese Verfahren beruhen auf dem Prinzip des **Self-Service-Scheduling** [FeRu90]. Es wird dabei eine Menge von verbliebenen Iterationen zentral verwaltet. Jeder Prozessor kann gleichberechtigt auf diesen Vorrat zugreifen, um zur Bearbeitung anstehende Iterationen zu erhalten. Hat ein Prozessor seine Iterationen vollständig bearbeitet, so fordert er weitere an. Alle partizipierenden Prozessoren verfahren in derselben Weise, bis die parallele Schleife vollständig bearbeitet ist. Die Realisierung beruht wesentlich auf gemeinsamen Registern oder ausgewiesenen Stellen im gemeinsamen Hauptspeicher, so daß auf die gemeinsame Menge von verbliebenen Iterationen von allen Prozessoren sehr schnell zugegriffen werden kann. Dieser Zugriff muß exklusiv erfolgen, damit die Aushändigung sowie die Korrektur der Zahl der verbliebenen Iterationen korrekt erfolgt. Im folgenden bezeichne der Begriff Dispatch-Operation eine solche exklusive Aushändigung.

Einmal ausgehändigte Iterationen können nicht mehr zurückgegeben werden. Deshalb ist es für eine gleichmäßige Lastverteilung wichtig, daß nicht zu viele Iterationen auf einmal angefordert werden. Andererseits sollen nicht zu viele Aushändigungsoperationen notwendig sein, weil immer nur eine Aushändigung gleichzeitig erfolgen kann. Es stellt sich somit die Frage nach einer günstigen Größe für auszuhändigende Iterationsblöcke.

5.1.1 Self-Scheduling

Als Self-Scheduling [TaYe86] bezeichnet man das Verfahren, bei dem jeweils eine einzelne Iteration ausgehändigt wird.

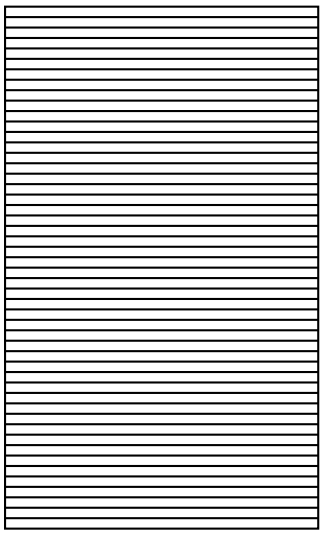


Abbildung 5.2: Beim Self-Scheduling werden einzelne Iterationen ausgehändigt.

Self-Scheduling führt zu optimaler Lastbalance, da gewährleistet ist, daß in dem Moment, in dem die letzte Iteration ausgehändigt wird, alle Prozessoren höchstens noch eine einzige Iteration zu bearbeiten haben. Der Nachteil des Self-Scheduling liegt darin, daß der Overhead im allgemeinen zu groß ist, weil ebensoviele Aushändigungen erfolgen wie Iterationen vorhanden sind.

5.1.2 Chunk-Scheduling

Die Zahl der Dispatch-Operationen wird reduziert, wenn nicht mehr einzelne Iterationen, sondern Iterationsblöcke fester Größe ausgehändigt werden. Diese Blöcke werden allgemein auch als **Chunks** bezeichnet.

Das Problem beim **Chunk-Scheduling** [KrWe85] besteht darin, eine geeignete Blockgröße zu ermitteln, die bei geringem Overhead zu einer guten Lastbalance führt. Der Programmierer muß sie für jede Schleife einzeln durch empirische Untersuchungen bestimmen. Ist die vom Programmierer vorgegebene Chunk-Größe kein Teiler der Iterationszahl N, so sind drei Möglichkeiten gegeben:

1. Ein Chunk (meist der letzte) ist kleiner als die anderen.
2. Ein Chunk (meist der erste) wird um den Rest vergrößert.
3. Der Rest wird möglichst gleichmäßig auf die Chunks verteilt.

5.1.3 Guided-Self-Scheduling

Da beim Chunk-Scheduling die Bestimmung der optimalen Chunk-Größe schwierig, bei einem sich zur Laufzeit änderndem Verhalten sogar unmöglich ist, haben Polychronopoulos

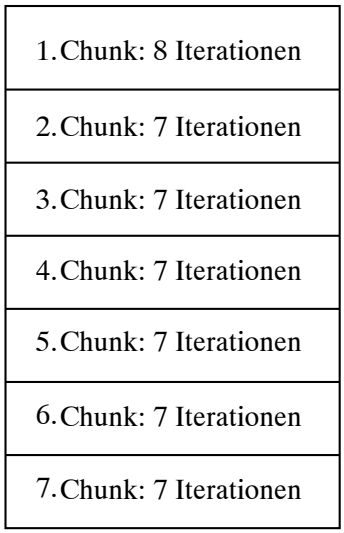


Abbildung 5.3: Chunk-Scheduling: N=50, Chunk-Größe=7

und Kuck 1987 [PoKu87] ein selbstadaptives Verfahren entwickelt. Bei diesem **Guided-Self-Scheduling (GSS)** erfolgt die Bestimmung der Chunk-Größe dynamisch. Im Laufe der Rechnung werden ausgehändigte Chunks immer kleiner. Man erreicht dadurch einen guten Kompromiß zwischen der Zahl der Dispatch-Operationen und einer gleichmäßigen Lastverteilung.

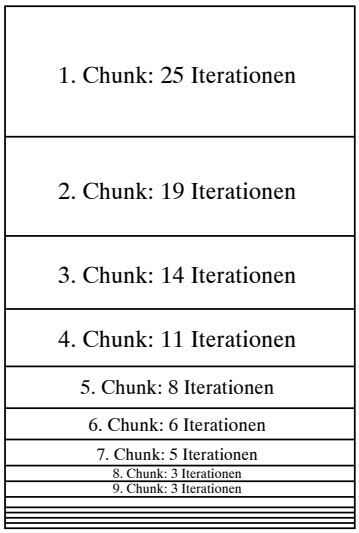


Abbildung 5.4: Guided-Self-Scheduling: N=100, P=4

Mit den Abkürzungen:

N : Zahl aller Iterationen
C_i : Größe des i-ten Chunks
R_i : Zahl der verbliebenen Iterationen nach der i-ten Aushändigung
P : Zahl der zur Verfügung stehenden Prozessoren

lautet die Formel, nach der die Größe des nächsten auszuhändigenden Chunks bestimmt wird:

$$\begin{aligned} C_i &:= \left\lceil \frac{R_i}{P} \right\rceil \\ R_0 &:= N \\ R_{i+1} &:= R_i - C_i = R_i - \left\lceil \frac{R_i}{P} \right\rceil \end{aligned}$$

Diese Wahl der Chunk-Größe führt zu optimaler Lastbalance, auch wenn Prozessoren, die bei der Ausführung einer parallelen Schleife mithelfen sollen, erst später hinzukommen. Vorausgesetzt ist dabei, daß die Bearbeitung aller Iterationen gleich lange dauert und daß kein Prozessor bei der Bearbeitung seines Chunks unterbrochen wird.

Optimal ist das Guided-Self-Scheduling deshalb, weil alle Prozessoren mit maximal einer Iteration Differenz fertig werden.

Die Autoren leiten in ihrer Arbeit als Größenordnung für die Zahl der Chunks σ und damit für die Zahl der Dispatch-Operationen den folgenden Ausdruck her:

$$\sigma \in O\left(P \cdot \ln\left(\frac{N}{P}\right)\right)$$

Das Problem dieser Strategie liegt in der Größe der ersten Chunks. Falls sich die Berechnung dieser großen Chunks verzögert, stehen nicht mehr genügend verbliebene Iterationen zur Verfügung, um diese Verzögerung noch ausgleichen zu können. Als Ursache der verzögerten Berechnung kommen dabei in Frage:

1. der zeitweise Entzug von Prozessoren in einer Multiprogramming-Umgebung, oder
2. ein Bearbeitungsaufwand, der gerade für die Iterationen in großen Chunks größer ist als der Durchschnitt.

5.1.4 Factoring

Das Problem der zu großen ersten Chunks beim Guided-Self-Scheduling löst das **Factoring**. Es wurde 1990 von Humel, Schonberg und Flynn im Rahmen des RP3-Forschungsprojektes bei IBM [HSF91] entwickelt. Die Autoren modellieren den Zeitbedarf, den die Ausführung einer einzelnen Schleifeniteration erfordert, allgemein durch eine Zufallsvariable. In diesem Modell leiten sie eine Formel für den Verlauf der Chunk-Größe her. Sie kommen zu dem Ergebnis, daß

$$\begin{aligned} C_i &:= \left\lceil \frac{R_i}{x_i \cdot P} \right\rceil \\ R_{i+1} &:= R_i - P \cdot C_i = R_i - P \cdot \left\lceil \frac{R_i}{x_i \cdot P} \right\rceil \end{aligned}$$

zu optimalen Ergebnissen führt. Die Größe x_i hängt dabei von P, R_i, σ und μ ab. σ und μ bezeichnen Varianz und Erwartungswert der angenommenen Verteilung. Da für reale Anwendungen die Rechenzeitverteilung der Iterationen selten im Voraus gegeben ist, setzen

die Autoren für ihre weiteren Untersuchungen $x_i \equiv 2$ und gelangen auch mit dieser Vereinfachung zu guten Ergebnissen. Mit dieser Festsetzung werden beim Factoring je P gleich große Chunks der Größe:

$$\begin{aligned} C_i &:= \left\lceil \frac{R_i}{2 \cdot P} \right\rceil \\ R_{i+1} &:= R_i - P \cdot C_i = R_i - P \cdot \left\lceil \frac{R_i}{2 \cdot P} \right\rceil \end{aligned}$$

ausgehändigt.

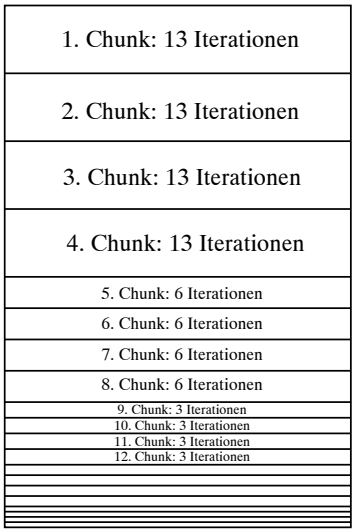


Abbildung 5.5: Factoring: N=100, P=4

In der Originalarbeit wird keine Aussage über die Zahl der Dispatch-Operationen gemacht, weshalb sie an dieser Stelle abgeschätzt werden soll. Um eine obere Grenze der Zahl der Chunks zu erhalten, kann man die obere Gauß-Klammer auflösen und die Zahl verbliebener Iterationen abschätzen zu:

$$\frac{R_i}{2} - P = R_i - \frac{R_i}{2} - P \leq \quad R_{i+1} \quad \leq R_i - \frac{R_i}{2} = \frac{R_i}{2}$$

Induktiv folgt mit $R_0 = N$:

$$N \cdot \left(\frac{1}{2}\right)^{i+1} - P \cdot \sum_{j=0}^{i} \left(\frac{1}{2}\right)^{j} \leq \quad R_{i+1} \quad \leq N \cdot \left(\frac{1}{2}\right)^{i+1}$$

beziehungsweise:

$$N \cdot \left(\frac{1}{2}\right)^{i} - 2 \cdot P \leq \quad R_i \quad \leq N \cdot \left(\frac{1}{2}\right)^{i}$$

Es folgt maximal eine weitere Iteration, wenn zum ersten Mal $R_{i_0} \leq 2 \cdot P$ erreicht wird. Deshalb wird i_0 so bestimmt, daß $R_{i_0} \leq 2 \cdot P$ gilt:

$$R_{i_0} \leq \quad N \cdot \left(\tfrac{1}{2}\right)^{i_0} \quad \leq 2 \cdot P,$$

also

$$i_0 \geq \frac{\ln\left(\frac{N}{2 \cdot P}\right)}{\ln(2)} = \frac{\ln\left(\frac{N}{P}\right)}{\ln(2)} - 1$$

Das Verfahren ist somit spätestens nach

$$i_0' = \left\lceil \frac{\ln\left(\frac{N}{P}\right)}{\ln(2)} \right\rceil$$

Iterationen abgeschlossen

Da in einer Iteration jeweils P gleich große Chunks ausgehändigt werden, folgt mit:

$$\sigma := P \cdot i_0' :$$

$$\sigma \in O\left(P \cdot \ln\left(\frac{N}{P}\right)\right)$$

Die Zahl der Dispatch-Operationen für das Factoring liegt somit in derselben Größenordnung wie die des Guided-Self-Scheduling. Da kleinere Chunks zugeteilt werden, ergeben sich mehr Dispatch-Operationen als beim Guided-Self-Scheduling. Simulationsergebnisse in [HSF91] zeigen jedoch, daß deren Anzahl gegenüber dem Guided-Self-Scheduling nicht signifikant erhöht ist.

Insgesamt ermöglicht das Factoring also eine gute dynamische Lastadaption bei einer moderat großen Zahl von Dispatch-Operationen.

6 Self-Service-Loop-Scheduling für SVM-Fortran

Mit dem Self-Service-Scheduling steht ein bewährtes Verfahren zur Realisierung von dynamischer Lastadaption auf Schleifenebene zur Verfügung. Insbesondere das Factoring ist geeignet, diese Lastadaption flexibel und mit einem vertretbaren Aufwand zu realisieren. Es liegt daher nahe, dieses Verfahren, das ursprünglich für Rechner mit gemeinsamem Hauptspeicher konzipiert worden ist, auch für Rechner mit verteiltem Hauptspeicher zu nutzen. In diesem Kapitel werden die Probleme beim Übergang auf verteilte Systeme und deren Lösungsmöglichkeiten beschrieben. Drüber hinaus wird dargestellt, wie die besonderen Anforderungen, die SVM an das Schleifen-Scheduling stellt, geeignet berücksichtigt werden können.

6.1 Probleme beim Übergang zu großen Prozessorzahlen

Die Realisierung des Self-Service-Scheduling bei Rechnern mit physikalisch gemeinsamem Hauptspeicher beruht wesentlich auf einem **gemeinsamen Register**, auf das jeder Prozessor schnell zugreifen kann. In diesem Register wird die globale Information über verbliebene Iterationen bzw. äquivalent dazu, die Information über die Nummer des gerade zur Bearbeitung anstehenden Iterationsblocks verwaltet. Ein schneller Zugriff auf dieses gemeinsame Register ist hier besonders wichtig, da die Hauptlast an paralleler Arbeit gerade in parallelen Schleifen geleistet wird, so daß deren effiziente Ausführung entscheidend für die Gesamt-Performance ist.

Bei Rechnern, die auf physikalisch verteiltem Speicher beruhen, finden sich solche gemeinsamen Register heute jedoch noch nicht[1]. Da auch SVM-Systeme über einen gemeinsam adressierbaren Hauptspeicher verfügen, könnte man die Idee verfolgen, die Menge von verbliebenen Iterationen über diesen virtuell gemeinsamen Speicher zu verwalten. Wie groß aber der Unterschied beim Zugriff auf gemeinsame Daten zwischen SVM-Systemen und Rechnern mit physikalisch gemeinsamen Registern ist, zeigt Tabelle 6.1.

Diese Gegenüberstellung der Zugriffszeiten auf gemeinsame Daten bei verschiedenen Systemen läßt eine Realisierung des Self-Service-Scheduling bei SVM-Systemen zunächst als

[1] Als Ausnahme wären eventuell die Fetch-And-Increment Register der Cray T3D zu nennen. Dabei handelt es sich zwar nicht um Register im eigentlichen Sinn, sondern um einen ausgezeichneten Bereich im Hauptspeicher eines bestimmten Prozessors, auf den alle Prozessoren gleichberechtigt zugreifen können. Ein Lesezugriff auf diese Speicherstelle bewirkt, daß der gelesene Wert automatisch und atomar in einem Schritt inkrementiert wird. Der gesamte Vorgang dauert nicht länger als jeder andere Zugriff auf nichtlokalen Speicher auch, also maximal $4\mu s$. Er ist damit im Vergleich zu anderen massiv-parallelen Rechnern sehr schnell.

Kommunikationsoperation	Dauer	Maschinentakte
Shared-Register Zugriff[2] bei einer Cray Y-MP	126 ns	21
Message-Passing Startup ohne Message-Koprozessor	90 μs	4.500
Write-Page Fault im ASVM-System	2,5 ms	125.000

Tabelle 6.1: Kommunikationszeiten verschiedener Systeme

aussichtslos erscheinen. Die Wartezeiten beim Zugriff auf den virtuell gemeinsamen Speicher sind viel zu lang, um feingranular auf Schleifenebene dynamische Verteilungsverfahren verwirklichen zu können.

Angesichts dieses Problems besteht die einzige Möglichkeit zur Verwirklichung eines dynamischen Schedulings darin, die schnellste zur Verfügung stehende Kommunikation (bei der aktuellen Zielmaschine also das Message-Passing) zu nutzen. Da diese im Vergleich zur Dauer eines lokalen Speicherzugriffs immer noch relativ lange dauert, erscheint ein Verfahren erfolgversprechend, bei dem sich Kommunikation und Berechnung überlappen. Das Ziel dabei ist, die unvermeidliche Wartezeit nicht mehr ungenutzt verstreichen zu lassen. Stattdessen können in der Zeit bis zum Eintreffen der angeforderten Daten andere Berechnungen durchgeführt werden [Gil94]. Damit dieses Verfahren gelingen kann, sind einige Voraussetzungen zu erfüllen:

1. Der erforderliche Kontextwechsel muß vom Betriebssystem geleistet werden können.
2. Die Zeit, die dieser Wechsel selbst in Anspruch nimmt, muß im Vergleich zu der zu leistenden Berechnungsarbeit gering sein. Dies bedeutet, daß die Granularität der Teilaufgaben nicht beliebig klein werden darf.
3. Bei der Programmierung eines solchen Ablaufs ist dafür Sorge zu tragen, daß die entsprechenden Prefetch-Operationen rechtzeitig abgesetzt werden können, so daß die angeforderten Daten dann, wenn auf sie zugegriffen werden soll, auch tatsächlich zur Verfügung stehen.

Ferner ist zu beachten, daß in der Regel jeder dieser Kontextwechsel dazu führt, daß Daten im Cache-Speicher im jeweils anderen Kontext nicht genutzt werden können. Daten im Cache-Speicher müssen so nach fast jedem Wechsel neu aus dem Hauptspeicher geladen werden.

Ein weiteres Problem beim Übergang zu großen Prozessorzahlen ist, daß der Zugriff auf *eine einzige* Menge von Restiterationen sicher einen Engpaß darstellen würde.

6.2 Ein verteilter Scheduler

Möchte man das Self-Service-Scheduling auf massiv-parallelen Architekturen realisieren, so besteht nur dann eine Chance für einen effizienten Ablauf, falls die genannten Lösungsstrategien berücksichtigt werden können. Im folgenden wird dargelegt, wie die angesprochenen

[2] siehe [LNVD91]

Lösungsstrategien

- Verteilung der Menge der verbliebenen Iterationen,
- Message-Passing als Kommunikationsmechanismus und
- Überlappung von Berechnung und Kommunikation

umgesetzt wurden, um einen verteilten Scheduler zu realisieren.

Möchte man die Menge von verbliebenen Iterationen verteilt verwalten, so ist zu entscheiden, wieviele Prozessoren jeweils konkurrierend auf einen gemeinsamen Teilvorrat zugreifen sollen. Entscheidet man sich für viele Prozessoren, so ergeben sich entsprechend wenige dieser Teilvorräte. Jeder einzelne Teilvorrat kann deshalb relativ viele Iterationen enthalten, so daß gute Voraussetzungen für eine dynamische Lastbalancierung gegeben sind. Das Problem bei diesen großen Teilvorräten ist aber wieder der konkurrierende Zugriff vieler Prozessoren auf wenige gemeinsame Ressourcen.

Nutzen hingegen nur wenige Prozessoren gemeinsam einen Vorrat, so wird dieser Vorrat relativ klein sein müssen, weil sich insgesamt viele dieser Teilvorräte ergeben. Bei ungleicher Lastverteilung werden dann einige Teilvorräte früher erschöpft sein als andere. Das Problem einer gleichmäßigen Lastverteilung stellt sich also zusätzlich auch zwischen diesen Teilmengen. Als Lösung kommt eine erneute Anwendung des Self-Service-Scheduling nun auch bei der Verteilung von Iterationsblöcken auf diese Teilmengen in Frage. Die wiederholte Anwendung führt zu einer rekursiv definierten Struktur, bei der in jeder Stufe die Zahl der Teilvorräte ab-, deren Größe hingegen zunimmt. In dieser Rekursion erkennt man die Definition eines balancierten Baumes, so daß sich ein balancierter Baum als die geeignete Kommunikationsstruktur für die dynamische Verteilung von Iterationen in massiv-parallelen Rechnern herausstellt.

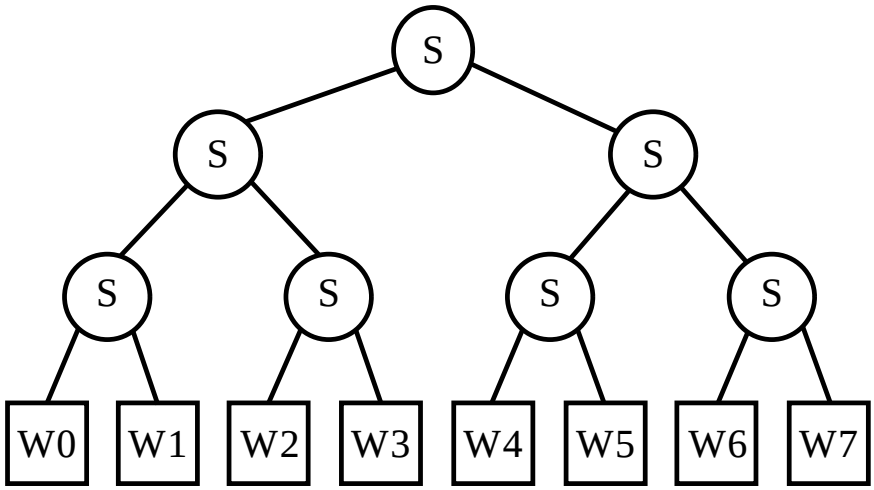


Abbildung 6.1: Worker und hierarchischer Scheduler

Hat man sich einmal für die Kommunikationsstruktur eines balancierten Baumes entschieden, so kann man diesen zusätzlich dazu nutzen, um im verteilten System globale Information zu akkumulieren oder zu verteilen. Die globale Information besteht bei der betrachteten Aufgabe in dem Wissen über die Terminierung einer Schleife. Nach der Aushändigung

der letzten Iteration sind so in einem *binären* Baum $\lceil 2 * ld(N) \rceil$ Kommunikationsstufen erforderlich[3], bis jeder der N beteiligten Knoten die Ausführung der Schleife beenden kann.

Es wurde deshalb eine Anordnung verwirklicht, bei der eine Menge von Worker-Knoten an einen hierarchischen Scheduler in Baumstruktur angebunden ist (vergleiche Abbildung 6.1).

Die Worker-Knoten sind dabei diejenigen, welche die eigentliche Berechnungsarbeit leisten. Jeder Worker, der zur Bearbeitung einer parallel auszuführenden Schleife hinzutritt oder der die ihm zugewiesene Arbeit vollständig erledigt hat, fordert bei dem ihm zugeordneten Scheduler-Knoten neue Iterationen an. Als Antwort erhält er entweder einen zu bearbeitenden Iterationsblock oder aber die Nachricht, daß keine weiteren Iterationen mehr anliegen, er also seine Bearbeitung dieser Schleife abschließen kann.

Innere Knoten im Scheduler-Baum haben lediglich die Aufgabe, die zu bearbeitenden Iterationsblöcke im System zu verteilen. Dazu verfügt jeder innere Scheduler-Knoten über eine eigene Warteschlange, in der er seinen lokalen Vorrat der Arbeitspakete verwaltet. Empfängt er von einem seiner beiden Nachfolger eine Nachricht mit der Aufforderung, weitere Iterationsblöcke auszuhändigen, so entnimmt er der Warteschlange – soweit noch vorhanden – die angeforderte Zahl der Arbeitspakete und händigt diese an den Absender der Anfrage aus. Ist sein eigener Vorrat an Iterationsblöcken erschöpft, so fordert er bei seinem Vorgänger neue Iterationen an. Erst wenn er diese erhalten hat, kann er wieder die Anfragen seiner Nachfolger befriedigen. Stellt sich bei der Anfrage an seinen Vorgänger aber heraus, daß alle Iterationen schon ausgehändigt worden sind, so gibt er diese Information statt der angeforderten neuen Iterationsblöcke an seinen Nachfolger weiter.

Die Zahl der Blöcke, die ein innerer Scheduler-Knoten auf einmal anfordert, entspricht dabei seinem Gewicht, also der Zahl der Worker in dem von ihm aufgespannten Unterbaum. Dies berücksichtigt, daß ein innerer Knoten auch als ein virtueller Worker-Knoten aufgefaßt werden kann, in dem die Rechenkapazität seiner Nachfolger akkumuliert ist. Mit dieser Wahl der Zahl der auf einmal auszuhändigenden Blöcke wird eine Verringerung des Nachrichtenaufkommens erreicht. Diese Zahl ist dabei, abhängig vom Gewicht des anfragenden Knotens, gerade so klein gewählt, daß sie die Forderung nach einer gleichmäßigen Lastverteilung nicht verletzt.

Die Wurzel des Scheduler-Baumes schließlich ist die einzige Stelle, an der die Größe der Iterationsblöcke bestimmt wird. Es sind die in Abschnitt 5 vorgestellten Verfahren

- SELF-SCHEDULING,
- CHUNK-SCHEDULING,
- GUIDED-SELF-SCHEDULING und
- FACTORING

realisiert. Zur Berechnung des jeweils nächsten Iterationsblocks hat die Wurzel die Information über den Gesamtiterationsraum, über die Zahl der verbliebenen Iterationen sowie beim CHUNK-SCHEDULING die Information über die vorgegebene Chunk-Größe. Die Bestimmung der nächsten Blöcke erfolgt immer dann, wenn eine Anfrage von einem

[3] ld bezeichnet den Logarithmus zur Basis 2 (logarithmus dualis). Die Konstante 2 ergibt sich, da von einem Knoten maximal 2 Nachrichten ausgehen bzw. bei ihm eintreffen. Die obere Gaußklammer berücksichtigt, daß N nicht immer eine Zweierpotenz sein muß.

Nachfolger eingetroffen ist. Die so bestimmten Iterationsblöcke werden zu einer Liste zusammengefaßt und in einer einzigen Nachricht an den anfragenden Nachfolger versendet. Ist der Iterationsraum vollständig ausgehändigt, so ist in der Nachricht an die Nachfolger ein entsprechender Eintrag enthalten. Die Wurzel kann das Warten auf eingehende Anfragen erst dann beenden, wenn eine Terminierungsnachricht an beide Nachfolger ergangen ist.

6.2.1 Abbildung auf vorhandene Prozessoren

Möchte man die beschriebene Anordnung aus einem Scheduler in Baumstruktur und einer Menge von Workern auf vorhandene Prozessoren abbilden, so scheidet eine eins-zu-eins Abbildung von logischen Knoten auf reale Prozessoren aus. In einem balancierten binären Baum ist die Summe der Nicht-Blattknoten immer um genau eins kleiner als die Zahl der Blattknoten. Dies bedeutet, daß nur die Hälfte der Prozessoren für die eigentliche Berechnungsaufgabe zur Verfügung stünde. Auch eine stärkere Aufspaltung des balancierten Scheduler-Baumes, die zu einer geringeren Zahl von Scheduler-Prozessoren führen würden, ist keine Lösung, da sich dabei wieder Kommunikationsengpässe bei inneren Knoten ergeben würden.

6.2.1.1 Asynchrone Message-Handler

Da innere Scheduler-Knoten im wesentlichen nur Nachrichten empfangen und neue aussenden, selbst aber nur sehr wenig Rechenleistung erbringen müssen, möchte man keine reinen Scheduler-Prozessoren investieren. Erfolgversprechend ist es deshalb, alle zur Verfügung stehenden Prozessoren zunächst als Worker einzusetzen, einige von ihnen aber zusätzlich auch mit Scheduler-Aufgaben zu betreuen. Dabei ist dann die Aufgabe zu lösen, wie der Wechsel zwischen diesen beiden Aufgaben günstig erfolgen kann. Dieser Wechsel kann aber nicht im Rahmen der Realisierung paralleler Schleifen erfolgen. In Abbildung 6.2 ist der

```
1 : GET_NEXT_BOUNDS (from, to, continue_iter)
2 : WHILE (continue_iter)
3 :   DO I = from, to
4 :     ...
5 :   ENDDO
6 :   GET_NEXT_BOUNDS (from, to, continue_iter)
7 : ENDWHILE
```

Abbildung 6.2: Zwischen-Code für eine parallele Schleife

Pseudo-Code für die Umsetzung paralleler Schleifen angegeben. Die While-Schleife von Zeile 2 bis 7 wird solange durchlaufen, wie weitere Iterationen zur Bearbeitung anstehen. Im Inneren dieser While-Schleife werden zugewiesene Blöcke von Iterationen schließlich ausgeführt. Die blockweise Ausführungsweise ist für die Effizienz der Ausführung wichtig, damit so wenig Aufrufe der Funktion GET_NEXT_BOUNDS wie möglich erfolgen müssen(vergleiche [Plu94]). Um die Effizienz dieser Ausführung zu erhalten, darf der Aufruf des Schedulers nur außerhalb der Bearbeitung eines Iterationsblocks erfolgen, in Abbildung 6.2 also zwischen den Zeilen 2 und 3. Aber selbst, wenn der Scheduler nach der Bearbeitung jeder einzelnen Iteration, also zwischen den Zeilen 3 und 4, aufgerufen würde,

kann nicht ausgeschlossen werden, daß die Wartezeit zwischen zwei Scheduler-Aktivitäten unakzeptabel lang wird. Diese Wartezeit ist deshalb besonders kritisch, weil während dieser Zeit eventuell alle anderen Prozessoren, die im Scheduler-Baum nachfolgen, vergeblich auf weitere Arbeitspakete warten. Es wird deshalb ein Mechanismus benötigt, der diesen Wechsel automatisch immer dann vornimmt, wenn gerade eine Scheduling-Aufgabe anliegt. Die Message-Passing NX-Library der Intel Paragon [Int94] sieht für diese Aufgabe die Message-Empfangsanweisung **hrecvx** vor. Die Semantik dieser Anweisung ist der Empfang der nächsten anliegenden Nachricht des angeforderten Typs. Nach dem Absetzen dieses Befehls wird die Programmausführung asynchron fortgeführt, ohne auf das Eintreffen der angeforderten Nachricht zu warten. Die Ankunft der angeforderten Nachricht löst dann einen Interrupt aus, der eine vom Benutzer definierte Behandlungsroutine aktiviert. Bei der Realisierung des Schedulers werden diese Behandlungsroutinen eingesetzt, um neu eintreffende Iterationen in die Warteschlange einzufügen, um bei Bedarf neue Iterationen anzufordern und schließlich um eintreffende Anfragen der Nachfolger zu beantworten.

Der Vorteil dabei ist, daß die Ausführung der parallelen Schleife nur so lange wie unbedingt nötig unterbrochen wird und daß die Scheduling-Operationen unmittelbar bei Bedarf erfolgen. Darüber hinaus müssen für das Scheduling keine dedizierten Prozessoren abgestellt werden. Da Versendung und Empfang der Nachrichten asynchron von der Berechnung der Schleifeniterationen erfolgen, können Berechnung und Kommunikation überlappen. Voraussetzung dafür ist, daß neue Iterationen angefordert werden, schon bevor der interne Vorrat vollständig erschöpft ist. Anfragen, die eintreffen, bevor neue Iterationen geliefert werden konnten, können deshalb meist sofort befriedigt werden.

6.2.2 Abbildung des binären Baumes auf reale Prozessoren

Der Vorteil einer asynchronen Ausführung von Schleifenbearbeitung und Scheduling kann ebenso für die unterste Stufe der Anordnung von Worker- und Scheduler-Knoten ausgenutzt werden. Das Ziel dabei ist ein Ablauf, bei dem jedesmal, wenn ein Prozessor einen Iterationsblock vollständig bearbeitet hat und deshalb einen weiteren anfordert, dieser neue Block nach Möglichkeit unmittelbar zur Verfügung steht. Es soll also die doppelte Message-Passing-Aufsetzzeit eingespart werden, die auftritt, wenn der Worker bei dem ihm zugeordneten Scheduler-Knoten weitere Iterationen anfordert und diese schließlich gesendet werden. Message-Passing läßt sich an dieser Stelle umgehen, falls jeder Prozessor sein eigener Scheduler ist. Dies bedeutet, daß eine Anfrage nach weiteren Iterationen nicht mehr an einen physikalisch entfernten Knoten, sondern an den Scheduler, der auf demselben Knoten arbeitet, gerichtet werden kann. Die Aushändigung des nächsten Iterationsblocks kann so effizient über einen Zugriff auf den von Scheduler und Worker gemeinsam genutzten Speicher erfolgen.

Die unterste Stufe des Scheduler-Baumes ist damit festgelegt. Die Blattknoten und der Worker werden gemeinsam von einem Prozessor realisiert. Noch nicht definiert ist, welche physikalische Prozessoren die inneren Knoten und die Wurzel des Schedulers bilden. In Abbildung 6.3 ist eine mögliche Abbildung für einen Scheduler-Baum mit acht Blättern angegeben. Charakteristisch ist dabei, daß jeder Prozessor nur einmal als Nicht-Blattknoten im binären Baum erscheint.

Dies ist aber nicht die einzig mögliche Art der Abbildung. So hat sich bei der Realisierung globaler Reduktionsoperationen eine andere Anordnung als günstig erwiesen. In Abbildung 6.4 ist dargestellt, daß der Vorgänger zweier benachbarter Knoten jeweils wieder einer

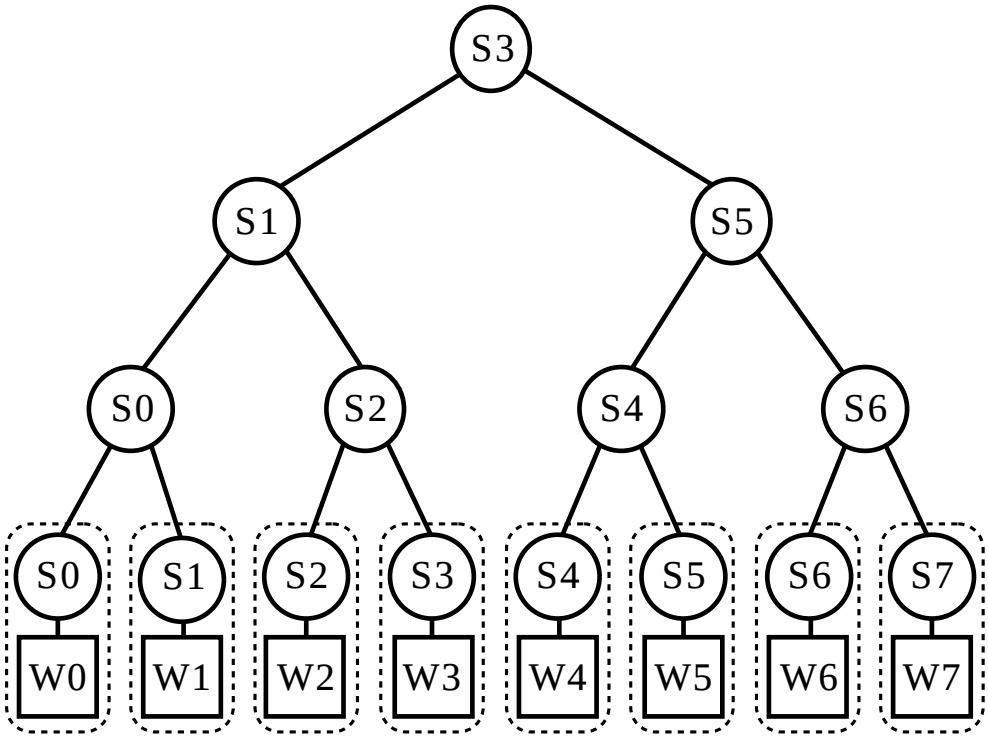


Abbildung 6.3: Jeder Prozessor tritt nur einmal als Nicht-Blattknoten im binären Baum auf.

der beiden Knoten selbst ist. Diese Anordnung ist gerade für globale Reduktionsoperationen vorteilhaft, weil ein einmal berechnetes Zwischenergebnis nur von einem Knoten an den Vorgänger übertragen werden muß. Der andere Nachfolger ist mit dem Vorgänger identisch, weshalb die Versendung einer Nachricht hier entfallen kann.

Um entscheiden zu können, welche Anordnung für den gegebenen Anwendungsfall besser geeignet ist, werden die Vor- und Nachteile beider möglichen Anordnungen im folgenden gegenübergestellt:

a) Jeder Knoten tritt nur einfach im binären Baum auf:

- **Vorteile:**
  - Jeder Knoten hat maximal zwei Nachfolger.
  - Die Programmierung ist für alle Knoten gleich.
- **Nachteile:**
  - Kommunikation ist in jeder Knotenverbindung erforderlich.
  - Jeder Pfad von der Wurzel bis zu einem Blatt hat dieselbe Länge $L = \lceil ld(P) \rceil$.

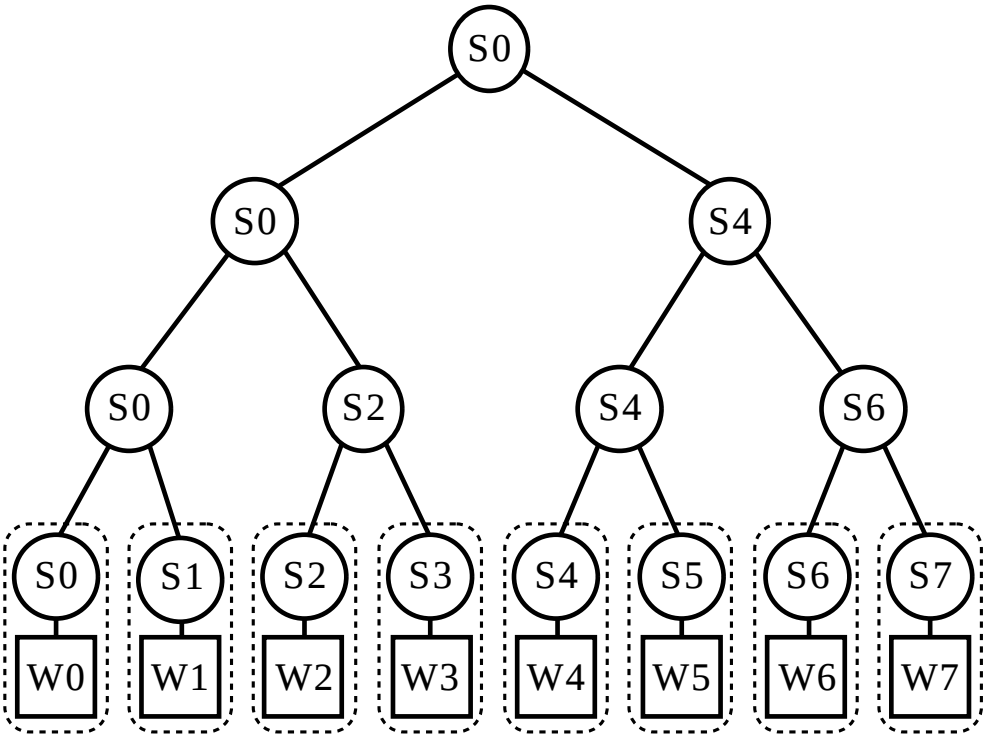


Abbildung 6.4: Der Vorgänger von benachbarten Knoten ist mit einem dieser beiden Knoten identisch. Einige Prozessoren treten mehrfach als Nicht-Blattknoten im binären Baum auf.

b) Der Vorgänger ist mit einem seiner Nachfolger identisch:

- **Vorteile:**
  - Insgesamt ist weniger Kommunikation erforderlich.
  - Der erste Worker kann sofort beginnen.
  - Der längste Pfad ist nicht länger als $L = \lceil ld(P) \rceil$.
- **Nachteile:**
  - S0 hat nicht mehr zwei, sondern $\lceil ld(P) \rceil$ Nachfolger. In Abbildung 6.5 ist die Struktur aus Abbildung 6.4 noch einmal anders dargestellt. Es ist dabei nicht mehr die logische, sondern die reale Verbindungsstruktur von Interesse. Knoten, die logisch an mehreren Positionen im binären Baum auftreten, sind physikalisch nur einmal vorhanden: Sie werden in dieser Darstellung also zusammengefügt. Daraus ergibt sich, daß die Zahl der Nachfolger von unten nach oben wächst, bis schließlich die Wurzel über $\lceil ld(P) \rceil$ Nachfolger verfügt.
  - Die Programmierung ist nicht mehr für alle Knoten gleich.

Besondere Beachtung verdient die Frage nach der Wartezeit des letzten Workers auf seinen ersten Block.

Die Zahl der Nachrichten, die der letzte Worker abwarten muß, bevor er seinen ersten Block erhält, hängt nicht nur von der Tiefe des Baumes ab. Da Nachrichten an die Nachfolgeknoten nur sequentiell bearbeitet werden können, geht die Zahl der Nachfolger eines jeden

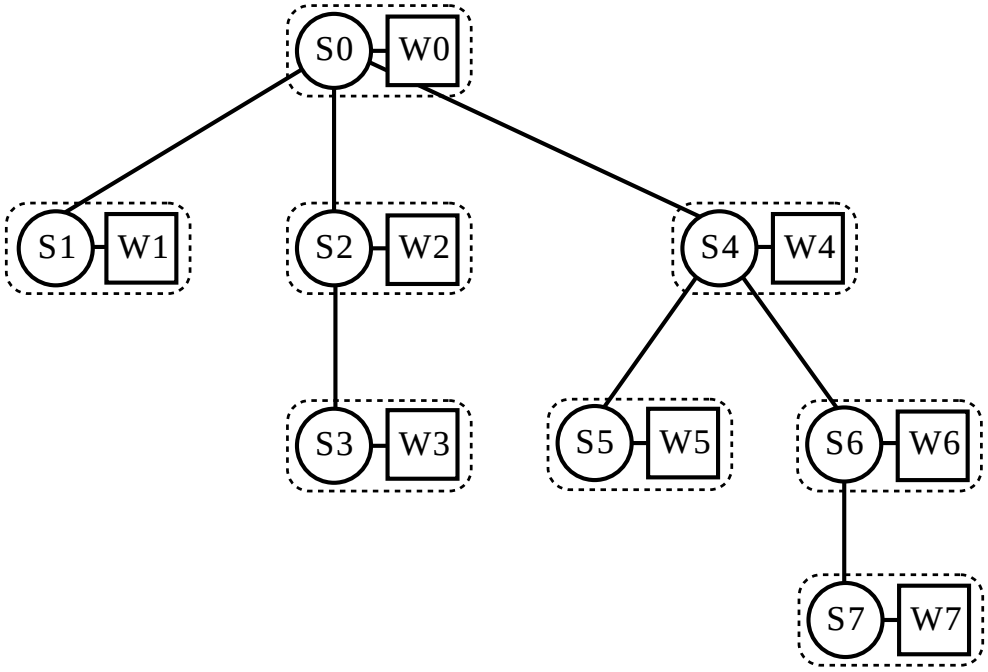


Abbildung 6.5: Die physikalische Verbindungsstruktur, falls einige Prozessoren mehrfach als Nicht-Blattknoten im binären Baum auftreten

Knotens auf dem Pfad von der Wurzel bis zu diesem Worker mit ein. Die größte Zahl der Wartestufen ergibt sich dabei, wenn auf diesem Pfad die erforderlichen Nachrichten jeweils zuletzt bearbeitet werden. In der Struktur von Abbildung 6.3 ist die maximale Tiefe proportional zum Logarithmus der Worker-Zahl. Jeder Scheduler-Knoten hat höchstens zwei Nachfolger. Somit erhält der letzte Worker sein erstes Arbeitspaket, nachdem $2 * \lceil ld(P) \rceil$ Kommunikationsstufen durchlaufen worden sind. Bei der Struktur aus Abbildung 6.5 muß Worker Nr. 7 am längsten warten, falls die Weitergabe an den rechten Nachfolger in jeder Stufe des Schedulers zuletzt erfolgt. Es ergeben sich hier $3 + 2 + 1 = 6$, im allgemeinen Fall also $\sum_{j=1}^{\lceil ld(P) \rceil} j = \frac{\lceil ld(P) \rceil (\lceil ld(P) \rceil + 1)}{2}$ Kommunikationsstufen. Dies ist im Vergleich zur Zahl der Wartestufen bei dem balancierten binären Baum sicher nicht mehr akzeptabel. Ein völlig anders Bild ergibt sich aber, wenn man die Bearbeitungsreihenfolge der ausgehenden Nachrichten nicht dem Zufall überläßt. Führt man nämlich Nachrichten an Nachfolger unter Berücksichtigung der restlichen Pfadlänge aus (in Abbildung 6.5 also von rechts nach links), so erhält jeder Worker seinen ersten Block nach maximal $\lceil ld(P) \rceil$ Stufen.

Insgesamt kann also mit dieser Methode die Zeit, bis zu der Worker mit der Schleifenbearbeitung beginnen können, auf die Hälfte reduziert werden.

6.3 Spezielle Anforderungen von SVM-Systemen

Die im vorigen Abschnitt vorgestellte Realisierung berücksichtigt Anforderungen, die massiv-parallele Message-Passing-Systeme an das dynamische Schleifen-Scheduling stellen. Dabei wurde den besonderen Anforderungen von SVM an das Schleifen-Scheduling

noch keine Beachtung geschenkt. So erfolgt die Berechnung von Iterationsblöcken beim Guided-Self-Scheduling sowie beim Factoring bisher ohne Berücksichtigung der Verwaltung von Daten in Einheiten ganzer Seiten.

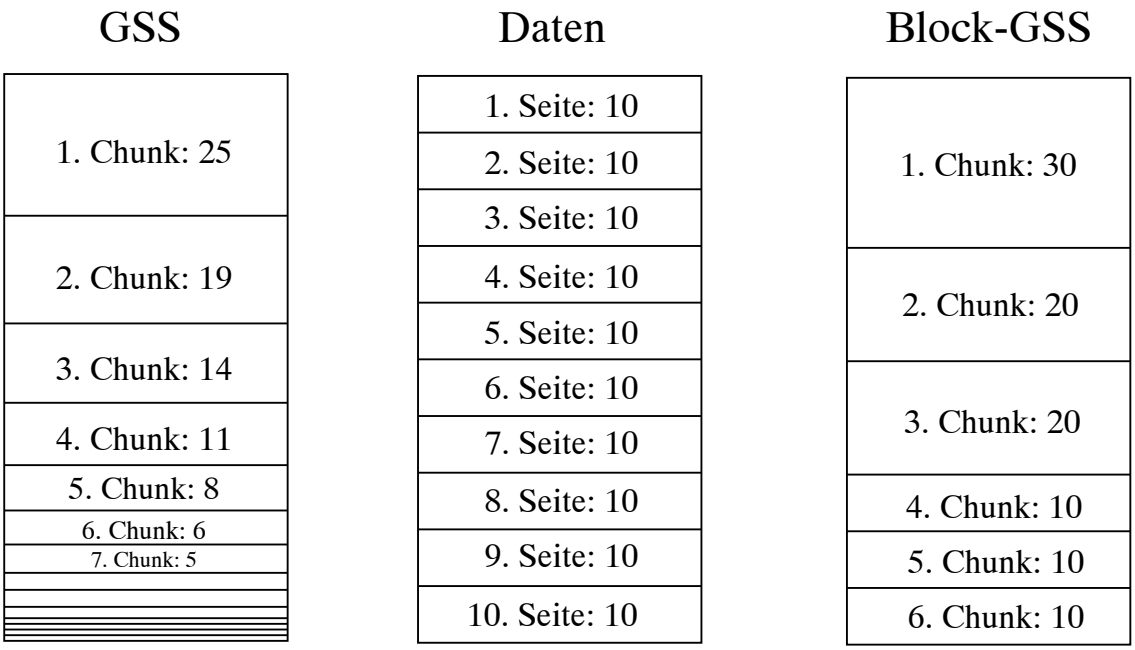


Abbildung 6.6: Loop-Blocking beim Guided-Self-Scheduling (GSS)

In Abbildung 6.6 kann man eine Situation erkennen, bei der aufeinanderfolgende Chunks, die meist verschiedenen Prozessoren zugeteilt werden, zu Zugriffen auf Daten in einer Seite führen. Im gegebenen Beispiel greifen die Prozessoren, die den 1. und den 2. Chunk zugewiesen bekommen haben, gleichzeitig auf Daten der 3. Seite zu. Bei diesem konkurrierenden Zugriff wird sich unvermeidlich Page-Trashing (vergleiche Abschnitt 2.3) einstellen, so daß eine effiziente Ausführung nicht mehr erzielt werden kann.

In [GrWi93] und [Lin94] wird Loop-Blocking als Lösung dieses Problems genannt. Wie ebenfalls in Abbildung 6.6 veranschaulicht, wird dabei die Größe eines neuen Iterations-Chunks so aufgerundet, daß sich ein Vielfaches der Zahl der Daten in einer Speicherseite ergibt. Unter den Voraussetzungen, daß die Datenfelder, auf die in einer solchen Schleife zugegriffen wird, auf einer Seitengrenze im gemeinsamen Hauptspeicher beginnen und daß auf diese Felder nicht indirekt zugegriffen wird, führt Loop-Blocking dazu, daß nur jeweils ein einziger Prozessor auf eine Speicherseite zugreift.

Unter den genannten Voraussetzungen stellt Loop-Blocking eine sehr einfache und wirkungsvolle Methode dar, um Page-Trashing zu reduzieren. Die vom Scheduler realisierten Verfahren wurden deshalb um BLOCK-GUIDED-SELF-SCHEDULING und BLOCK-FACTORING erweitert.

Das Loop-Blocking ist im Prinzip gut geeignet, die seitenweise Verwaltung von Daten unter SVM zu berücksichtigen. Es stellt sich aber angesichts einer Seitengröße von 8192 Byte bei der Intel Paragon die Frage danach, wie groß ein Iterationsraum sein muß, damit sich bei dem Aufrunden auf die folgende Seitengrenze überhaupt noch verschieden große Chunks ergeben. 8192 Byte entsprechen 1024 DOUBLE-PRECISION-Werte oder 2048

32-Bit INTEGER-Werte. Um die gestellte Frage beantworten zu können, hilft zunächst die folgende Feststellung: Geht man bei einer festgehaltenen Zahl von Prozessoren auf einen Iterationsraum über, dessen Ausdehnung ein Vielfaches des ursprünglichen ist, so bleibt das Größenverhältnis aufeinanderfolgender Chunks im wesentlichen wie zuvor. Der i-te neue Chunk hat, abgesehen von Rundungseffekten, die Größe des i-ten alten Chunks, multipliziert mit dem Vergrößerungsfaktor des Iterationsraumes. Die Frage, bei welcher Ausdehnung des Iterationsraumes sich beim Loop-Blocking noch verschieden große Chunks ergeben, läßt sich somit reduzieren auf die Frage, wann sich beim Guided-Self-Scheduling bzw. beim Factoring andere Chunks als solche der Größe 1 ausbilden. Diese Frage aber läßt sich relativ leicht beantworten, wenn man sich die Berechnungsvorschrift für den ersten Chunk ansieht: Sobald jeder Prozessor im Durchschnitt K Iterationen beisteuert, wird der erste Block beim Guided-Self-Scheduling die Größe K bzw. beim Factoring die Große $\lceil \frac{K}{2} \rceil$ annehmen. Folgende Chunks werden dann schrittweise kleiner. Übertragen auf das Loop-Blocking bedeutet dies, daß sich verschieden große Chunks dann einstellen, falls von jedem Prozessor einige Tausend Iterationen zu bearbeiten sind.

7 Aligned-Scheduling

Zum Ende von Kapitel 6 hat sich herausgestellt, daß die Umsetzung des Self-Service-Scheduling-Verfahrens zwar gut geeignet ist, eine dynamische Lastadaption zu leisten, aber in ihrer ursprünglichen Form keine Rücksicht auf die zugrundeliegende Datenverteilung nimmt. Dies führt während des Programmlaufs zu einem hohen Anteil nicht-lokaler Datenzugriffe, worunter die Effizienz erheblich leidet. Loop-Blocking hat sich als ein geeignetes Verfahren zur Berücksichtigung der seitenweisen Datenanordnung in SVM-Systemen erwiesen. Unter der Zusatzbedingung einer Ausrichtung von Datenfeldern auf den Beginn einer Speicherseite in Verbindung mit einem direkten Zugriff auf diese Felder arbeitet beim Loop-Blocking nur jeweils ein einziger Prozessor mit Daten einer Seite. In diesem Kapitel wird beschrieben, wie man in SVM-Systemen eine vorliegende Datenverteilung berücksichtigen kann, auch wenn die genannten Bedingungen nicht erfüllt werden können.

Die Berücksichtigung der aktuell vorliegenden Datenverteilung ist notwendig, damit sich Lokalität beim Datenzugriff einstellen kann. Daten sollen nach Möglichkeit dort bearbeitet werden, wo sie lokal vorhanden sind. In [ZBCM92] und [HKKK92] wird diese Strategie auch als **Owner-Compute-Rule** bezeichnet. Um Lokalität beim Datenzugriff zu erreichen, ist in SVM-Fortran bisher die in Kapitel 3 beschriebene und in [Lin94] als **Affinity-Scheduling** bezeichnete Methode realisiert. Dabei wird eine Verteilung von Schleifeniterationen auf Prozessoren vorgegeben. Die wiederholte Verwendung dieser Verteilung führt unter SVM schließlich dazu, daß sich die Verteilung der Speicherseiten dem Zugriffsmuster anpaßt. Die letzte Aussage kann aber immer nur mit einer gewissen Wahrscheinlichkeit gemacht werden, da Speicherseiten in SVM-Systemen nicht fest an einen bestimmten Prozessor gebunden sind.

Eine direkte Implementierung der Owner-Compute-Rule hingegen ist das in [PDM92] genannte und in [Lin94] untersuchte **Aligned-Scheduling**. Dabei wird das Schleifen-Scheduling an die vorgefundene Datenverteilung eines bestimmten Feldes angepaßt. Realisiert wird Aligned-Scheduling dadurch, daß zunächst jeder Prozessor herausfindet, welcher Teil des zu bearbeitenden Datenfeldes bei ihm lokal vorhanden ist. Anschließend bestimmt er diejenigen Iterationen, die auf diesen lokal vorhandenen Teil zugreifen, und führt sie schließlich aus.

7.1 Bestimmung der Seitenverteilungsinformation

Die Information darüber, welcher Teil eines gemeinsamen Datenfeldes lokal vorhanden ist, kann nur gewonnen werden, wenn das Betriebssystem dem Benutzer den aktuellen Zustand der Page-Owner-Tabelle zur Verfügung stellt. Das von SVM-Fortran genutzte ASVM, entwickelt durch die Firma Intel SSD, ist das einzige System, das bereits heute

diese Funktionalität bietet (siehe [BGM95]). Die Bereitstellung der Seitenzustandsinformation ist dabei nur ein Aspekt einer umfassenden Monitoring-Umgebung für Anwendungsprogramme. Zu beliebigen Zeiten kann das Monitoring für Bereiche im globalen Adreßraum, die der Benutzer vorgeben kann, initialisiert werden. Danach steht in der lokalen Datenstruktur SVM_MON_TASK_PAGE_STATE_INFO ein Feld zur Verfügung, das für jede Seite innerhalb des interessierenden Adreßbereichs den eigenen Besitz- und Zugriffsstatus enthält. Mögliche Zustände sind:

- **SVM_MON_PAGE_READ_ONLY:** Die Seite kann nur gelesen werden. Ein anderer Knoten ist der Besitzer der Seite.
- **SVM_MON_PAGE_OWNED:** Die Seite kann nur gelesen werden. Der Knoten selbst ist der Besitzer der Seite.
- **SVM_MON_PAGE_READ_WRITE:** Die Seite darf geschrieben und gelesen werden.
- **SVM_MON_PAGE_NOT_PRESENT:** Die Seite ist lokal nicht vorhanden.

Für den gegebenen Anwendungsfall muß die Information darüber, welcher Prozessor eine bestimmte Iteration ausführen soll, unbedingt eindeutig sein, damit jede Iteration genau einmal ausgeführt wird. Dies impliziert, daß auch die Information darüber, welcher Prozessor über eine bestimmte Seite verfügt, eindeutig sein muß. Da von einer Speicherseite mehrere Lesekopien existieren können, kommt nur der eindeutige Besitzer einer Seite in Frage, um die entsprechenden Iterationen auszuführen. Dieser Besitzer hat für die betrachtete Seite entweder den Zustand SVM_MON_PAGE_OWNED oder den Zustand SVM_MON_PAGE_READ_WRITE.

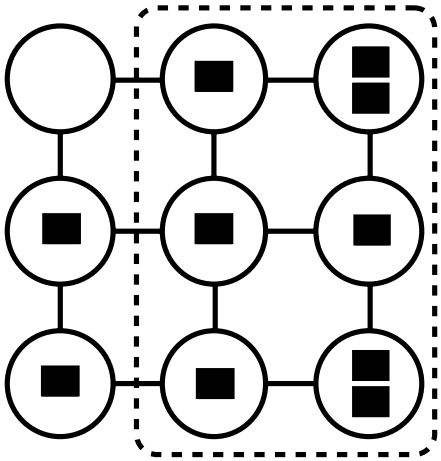


Abbildung 7.1: Eine angenommene Verteilung von Speicherseiten auf ein 3x3-Prozessorfeld

In Abbildung 7.1 ist dargestellt, wie Speicherseiten, die Teile eines gemeinsamen Datenfeldes enthalten, auf ein 3x3-Prozessorfeld verteilt sein könnten. Der eingerahmte Bereich symbolisiert, daß es in SVM-Fortran möglich ist, benutzerspezifizierte Programmbereiche nur auf Teilen eines Prozessorfeldes ausführen zu lassen. Mit dieser Methode wird funktionale Parallelität in SVM-Fortran ausgedrückt. Gerade diese Möglichkeit stellt aber für das

Aligned-Scheduling einen zu berücksichtigenden Sonderfall dar. Läuft nämlich Aligned-Scheduling nicht in dem vollständigen Prozessorfeld ab, so kann (wie in Abbildung 7.1 dargestellt) nicht garantiert werden, daß sich auch tatsächlich alle Speicherseiten des Datenfeldes innerhalb des Teilprozessorfeldes befinden. Würde in dieser Situation lediglich jeder Prozessor diejenigen Iterationen ausführen, die zu seinen lokal vorhandenen Daten gehören, so bliebe die Schleifenbearbeitung unvollständig. Es ist deshalb unvermeidlich, die Information über die globale Seitenverteilung an einer zentralen Stelle zu akkumulieren. Erst dadurch besteht die Möglichkeit, zu entscheiden, ob in der aktuell gültigen Prozessormenge Seiten eines Datenfeldes fehlen. Fehlende Seiten werden gleichmäßig auf die vorhandene Prozessormenge verteilt. Genau genommen werden dabei aber nicht die Seiten selbst verteilt, sondern es erfolgt eine Zuordnung von Seiten auf Prozessoren. Die Seiten selbst werden dann später vom Betriebssystem beim ersten Zugriff auf diese transportiert.

Neben dem Fehlen von Speicherseiten in einem Teilprozessorfeld ist zusätzlich auch ein weiterer Fall zu berücksichtigen. Im verteilten System können aufgrund unterschiedlichen Laufzeitverhaltens zwei oder mehr Prozessoren unabhängig voneinander zu der Entscheidung kommen, daß sich eine bestimmte Seite bei ihnen befindet.

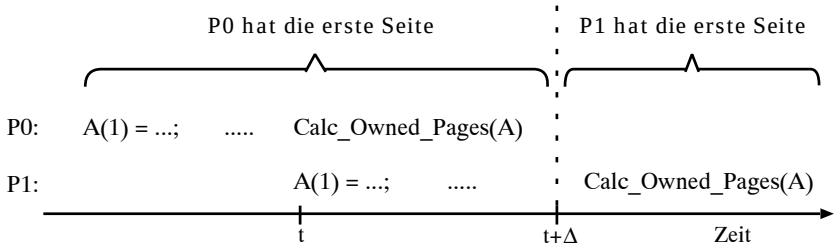


Abbildung 7.2: Sowohl P0 als auch P1 halten sich für den Besitzer der ersten Speicherseite des Feldes A.

In Abbildung 7.2 ist Prozessor P0 zum Zeitpunkt t der Besitzer der ersten Speicherseite des Datenfeldes A, weil er als letzter in sie geschrieben hat. Da er zu diesem Zeitpunkt seine Page-Owner-Tabelle ausliest, gelangt zu der Information, daß er im Moment der Besitzer dieser Seite ist. Zur selben Zeit t schreibt Prozessor P1 in diese Speicherseite. Bis die Besitzrechte von P0 auf P1 übergegangen sind, vergeht die Zeit Δ. Noch bevor diese Zeitspanne Δ abgelaufen ist, hat P0 das Auslesen seiner Page-Owner-Tabelle abgeschlossen. Erst nachdem die Besitzrechte auf P1 übertragen worden sind, kommt auch P1 dazu, seine Page-Owner-Tabelle auszuwerten. Insgesamt halten sich also sowohl P0 wie auch P1 für den Besitzer dieser Speicherseite. Um zu verhindern, daß Iterationen doppelt ausgeführt werden, muß ein Synchronisationspunkt vor dem Auslesen der Page-Owner-Tabelle eingefügt werden. Dieser garantiert, daß alle Schreibzugriffe, die zu einer Änderung der Seitenverteilung führen könnten, abgeschlossen sind, bevor die Page-Owner-Tabellen ausgewertet werden. Der genannte Synchronisationspunkt braucht allerdings nicht zusätzlich in den Programmtext eingefügt zu werden. Da die Akkumulation der globalen Seitenverteilungsinformation ohnehin notwendig ist, kann der Fall doppelt vorhandener Seiten leicht zusätzlich überprüft werden.

Insgesamt erkennt man für Schleifen, bei denen Aligned-Scheduling angewendet wird, die Notwendigkeit einer globalen Operation. Entweder ist explizit ein Synchronisationspunkt einzufügen, oder in Form der Akkumulation der Seitenverteilungsinformation liegt implizit

einen Synchronisationspunkt vor.

Eine Optimierung durch den Verzicht auf den Synchronisationspunkt vor der Ausführung der Schleife, wie sie für andere parallele Schleifen möglich ist, wenn Datenabhängigkeiten dies zulassen, scheidet hier also aus.

7.2 Die Umkehrung der Align-Funktion

Nachdem die Information über lokal vorhandene Speicherseiten vorliegt, ist der folgende Schritt bei der Realisierung des Aligned-Scheduling die Ermittlung der Iterationen, die auf lokale Daten zugreifen (vergleiche Abbildung 7.3).

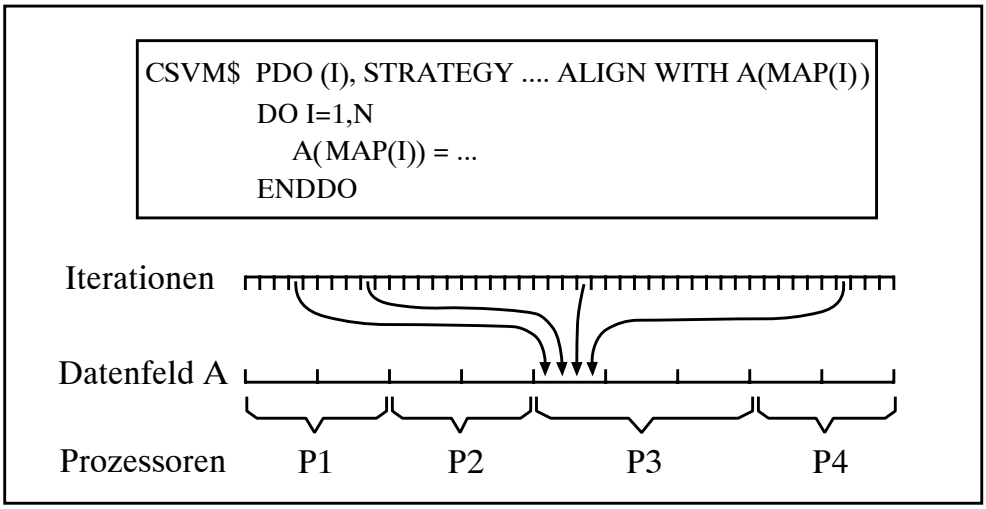


Abbildung 7.3: Beim Aligned-Scheduling wird der Prozessor, der eine bestimmte Schleifeniteration ausführen soll, in zwei Stufen ermittelt.

Abbildung 7.3 verdeutlicht, daß beim Aligned-Scheduling die Zuordnung von Iterationen zu Prozessoren in zwei Stufen erfolgt. In der ersten Stufe werden die Iterationen über eine Align-Funktion auf Datenfeldelemente abgebildet. Jedes derartige Feldelement gehört zu einer Speicherseite, welche sich bei einem bestimmten Prozessor befindet.

Wenn die Zuordnung von Speicherseiten auf Prozessoren bekannt ist, schließt sich die Umkehrung der Align-Funktion an, um die Iterationen zu bestimmen, die gemeinsam auf Daten in einer Seite zugreifen. Dabei sind zwei Fälle zu unterscheiden:

1. Die Align-Funktion ist eine lineare Funktion und deshalb direkt umkehrbar.

2. Die Align-Funktion ist durch eine Tabelle mit beliebig gewählten Einträgen gegeben und deshalb nicht direkt umkehrbar.

Ist die Align-Funktion linear, so kann die Umkehrung direkt für jedes betrachtete Datenfeldelement einzeln erfolgen. Insbesondere ist keine Kommunikation zwischen den Prozessoren erforderlich. Der Nachteil besteht aber darin, daß durch lineare Funktionen nur eine eingeschränkte Klasse von Verteilungen angegeben werden kann.

Möchte man aber auch allgemeinere Align-Funktionen wie z.B. Mapping-Arrays zulassen, so scheidet die direkte Umkehrung aus. Es bleibt nur noch der Weg über die Funktionsauswertung. Wie bei der Realisierung der DISTRIBUTE-Direktive wird dabei für alle erlaubten Schleifenindizes die Align-Funktion ausgewertet. So kann entschieden werden, ob die gerade untersuchte Iteration zu einem Zugriff in eine bestimmte Speicherseite führt. Das Problem bei diesem Verfahren ist, daß alle beteiligten Prozessoren den vollständigen Iterationsraum durchlaufen müssen, was einer Sequentialisierung der eigentlich parallel zu bearbeitenden Schleife gleichkommt. Bei der Umsetzung der DISTRIBUTE-Direktive (vergleiche Abschnitt 4.2 und 4.3) konnte dies akzeptiert werden, da eine DISTRIBUTE-Anweisung nur wenige Male im Lauf eines Programms, nicht aber bei jeder parallelen Schleife, ausgeführt wird. Hier jedoch besteht der Wunsch nach einer Beschleunigung dieses Umkehrungsvorgangs.

7.2.1 Parallele Umkehrung der Align-Funktion

Eine Beschleunigung kann, da die Zahl auszuführender Operationen nicht reduziert werden kann, nur durch eine parallele Auswertung erreicht werden.

In einem ersten Schritt wird der Iterationsraum blockweise partitioniert und auf die Prozessoren verteilt. Dann durchläuft jeder Prozessor den ihm zugeordneten Teiliterationsbereich und wertet für jeden Index die Align-Funktion aus. Das Ergebnis wird interpretiert als Offset relativ zum Anfang des Datenfeldes, an dem das Schleifen-Scheduling auszurichten ist. Wenn die Anfangsadresse des Datenfeldes bekannt ist, kann so entschieden werden, auf welche Speicherseite jede Iteration zugreifen wird. Iterationen, die gemeinsam auf eine bestimmte Speicherseite zugreifen, werden in einer geordneten Liste zusammengefaßt. Die folgende Aufgabe ist die Akkumulation der bisher im System verteilt vorhandenen Information, denn jeder Prozessor hat bisher lediglich die Information, auf welche Seiten ein Teil des Gesamtiterationsbereichs zugreifen wird. Die Akkumulation erfolgt durch sukzessives Zusammenführen von Teillisten, bis schließlich die Wurzel eines binären Baumes über die Gesamtinformation verfügt (siehe Abbildung 7.4). Die Iterationslisten speichern dabei zusammenhängende Blöcke (vergleiche Abschnitt 4.1), so daß sich die Größe der versendeten Nachrichten verringert, falls sich tatsächlich Iterationsblöcke ausbilden.

Bei der gerade erläuterten Methode sammelt sich die Gesamtinformation schließlich bei einem einzigen Prozessor. Dies ist gleich aus zwei Gründen ungünstig: Zum einen kann die Datenmenge sehr groß werden, wenn sich kaum Iterationsblöcke ausbilden, und zum anderen wird die Information darüber, welche Iterationen auf eine bestimme Speicherseite zugreifen, nicht an einer zentralen Stelle, sondern genau dort benötigt, wo die entsprechende Seite auch tatsächlich vorhanden ist. Erforderlich ist deshalb eine Struktur, bei der jeder Prozessor, der über mindestens eine betroffene Speicherseite verfügt, die Wurzel eines solchen Reduktionsbaumes ist. Es hat sich herausgestellt, daß sich diese logisch erforderlichen binären Bäume in einer einzigen Verbindungs- und Kommunikationsstruktur zusammenfassen lassen. Die Verbindungsstruktur, die dies leistet, ist in der Literatur als Banyan-Netzwerk [HwBr84] bekannt.

In Abbildung 7.5 ist die Konstruktion eines Banyan-Netzwerks für vier Knoten dargestellt. Man erkennt in jedem Knoten die Wurzel eines balancierten binären Baumes. Beim Übergang von einer Stufe i auf die nächste Stufe (i+1) ist jeder Knoten mit sich selbst und mit einem anderen Knoten im Abstand 2^i verbunden. Die Wahl dieser Kommunikationsstruktur gestattet es, die verteilte Information so zu akkumulieren, daß sich das Wissen

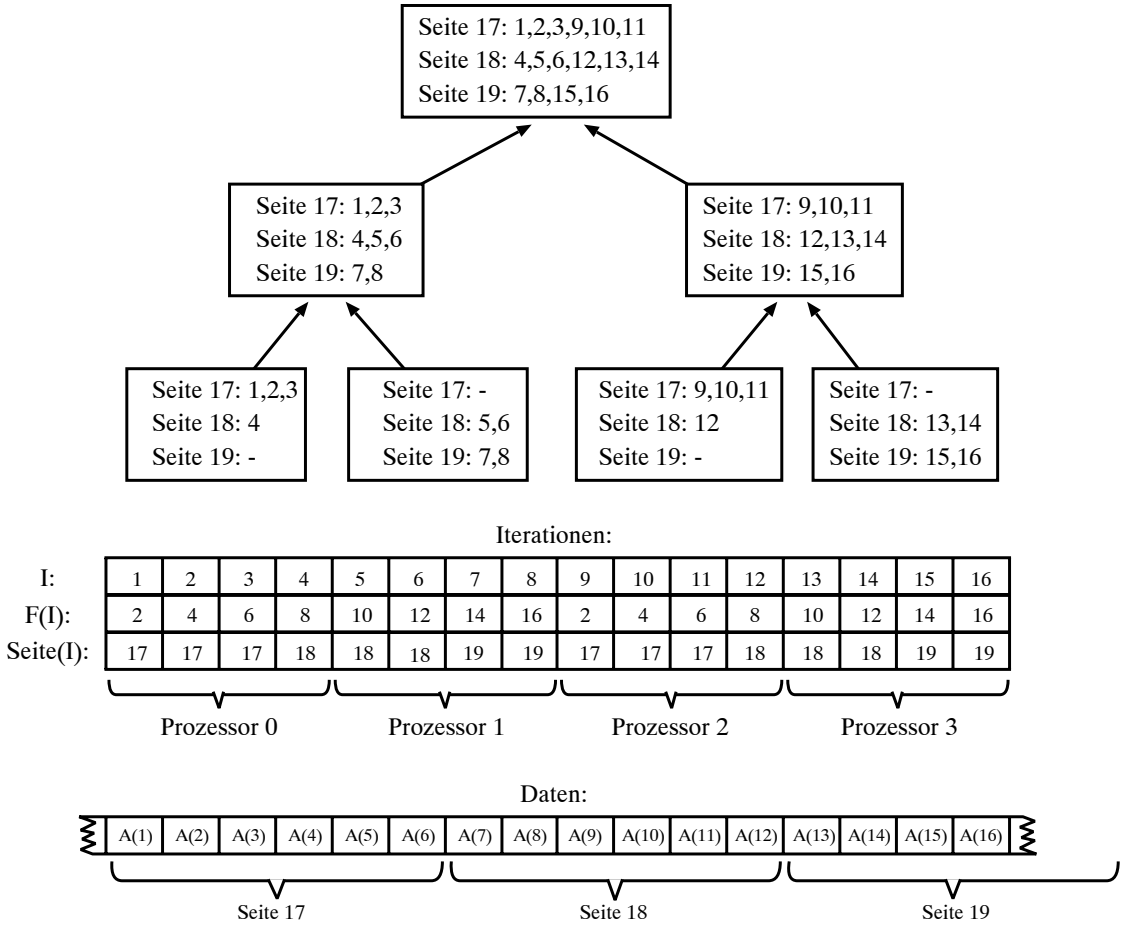


Abbildung 7.4: Ein Beispiel zur parallelen Invertierung der Align-Funktion. Jede Speicherseite nimmt 6 Feldelemente auf. Die Align-Funktion lautet: F(I) = 2*I MOD' 16. MOD' ist eine modifizierte Modulo-Funktion mit Wertebereich von 1 bis 16 statt von 0 bis 15.

wieder in verteilter Form im System befindet. Auch andere Kommunikationsstrukturen wie z.B. das Omega-Netzwerk könnten dies leisten. Das Banyan-Netzwerk ist aber hier besonders geeignet, weil es in seiner Kommunikationsstruktur Fixpunkte aufweist, so daß jeweils nur mit einem einzigen anderen Knoten kommuniziert werden muß.

In Abbildung 7.6 ist dargestellt, wie Teillisten in der Banyan-Struktur schrittweise zusammengeführt werden.

7.2.2 Analyse des Laufzeitverhaltens

Wie immer beim Übergang auf ein paralleles Verfahren stellt sich auch hier die Frage nach der Effizienz. Es soll deswegen im folgenden der Berechnungsaufwand des Verfahrens, bei dem jeder Prozessor den gesamten Iterationsraum durchläuft, mit dem Aufwand des auf der Banyan-Struktur basierenden Verfahrens verglichen werden. Das Verfahren, bei dem jeder Prozessor den gesamten Iterationsraum durchläuft, sei hier als sequentiell bezeichnet,

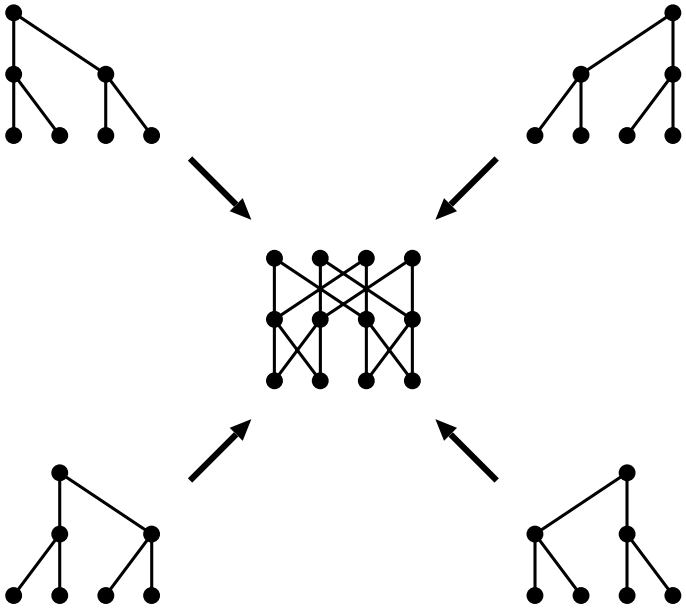


Abbildung 7.5: Die Struktur eines Banyan-Netzwerks

auch wenn alle Prozessoren daran beteiligt sind. In folgenden bezeichne N die Größe des Iterationsraumes und P die Zahl der Prozessoren.

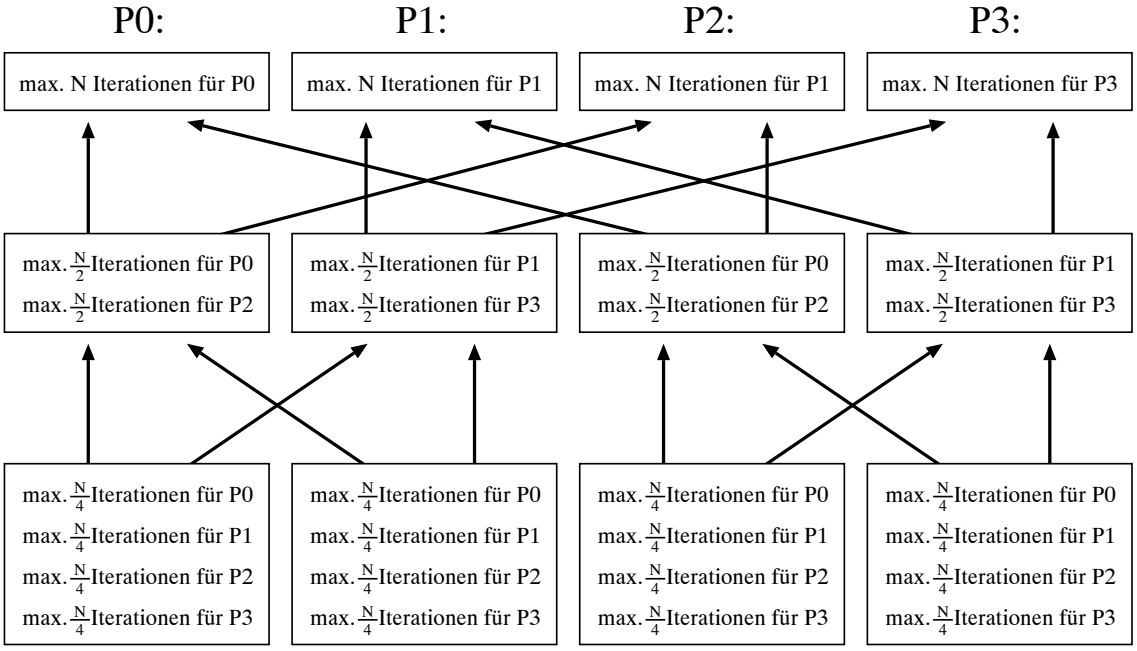


Abbildung 7.6: Der Datenfluß im Banyan-Netzwerk

Bei der sequentiellen Ausführung sind von jedem Prozessor zu leisten:

- Die N-fache Auswertung der Align-Funktion
- Das N-fache Anfügen an eine sortierte Liste

Bei der parallelen Ausführung sind nacheinander zu leisten:

- Die $\frac{N}{P}$-fache Auswertung der Align-Funktion
- Das $\frac{N}{P}$-fache Anfügen an eine sortierte Liste
- Das Zusammenfügen der sortierten Listen

Der Anteil, der sich aus der Auswertung der Align-Funktion und dem Anfügen an die sortierten Listen zusammensetzt, laßt sich somit linear mit der Anzahl der Prozessoren beschleunigen. Weiterer Analyse bedarf die Zeit, die das Zusammenfügen der sortierten Listen erfordert. Dieser Zeitbedarf setzt sich aus zwei Bestandteilen zusammen:

1. der Rechenzeit, die das Zusammenfügen der Teillisten in Anspruch nimmt und
2. der Datenübertragungszeit.

Die Rechenzeit, die das Zusammenfügen der Teillisten in Anspruch nimmt, darf als Konstante angesetzt werden, weil der Iterationsraum in aufsteigender Ordnung auf die Prozessoren aufgeteilt wird. Dies hat zur Folge, daß eine Teilliste von Iterationen, die gemeinsam auf eine Speicherseite zugreifen, lediglich an eine andere Teilliste angefügt werden muß. Ein Mischen ist nicht erforderlich.

Der Aufwand für die Datenübertragung läßt sich nicht einfach angeben, denn es stellt sich heraus, daß er von der Verteilung der Iterationen auf die Prozessoren abhängt. Insgesamt sind in jeder der $\lceil ld(P) \rceil$ Stufen der Banyan-Struktur N Iterationen zusammenzuführen. Diese gesamte Zahl ist jedoch für das Laufzeitverhalten nur von untergeordneter Bedeutung – entscheidend ist die maximale Datenübertragungszeit in jeder Stufe. Diese wiederum hängt davon ab, ob sich Blöcke ausbilden, und wie gleichmäßig die Iterationen auf die Prozessoren verteilt sind. Da eine exakte Angabe nicht möglich ist, sollen der günstigste, der ungünstigste und der durchschnittliche Fall diskutiert werden:

- Der **ungünstigste Fall** liegt dann vor, wenn alle geraden (oder alle ungeraden) Iterationen von einem einzigen Prozessor auszuführen sind. In diesem Fall sind in der letzten Stufe von einem einzigen Prozessor zwei Teillisten der Ausdehnung $\frac{N}{4}$ zusammenzuführen. Jeder Iterationsblock hat dabei die Länge eins. In der zweiten Stufe verbinden zwei Prozessoren vier Teillisten der Ausdehnung $\frac{N}{8}$. Die Länge der Teillisten halbiert sich von Stufe zu Stufe, bis schließlich auf der untersten Stufe von $\frac{N}{2}$ Prozessoren zwei Listen der Länge $\frac{N}{2*P}$ zusammenzufassen sind. Als Summe dieser Längen ergibt sich:

$$S = N * \sum_{J=1}^{\lceil ld(P) \rceil} (\frac{1}{2})^j < N$$

  Es sind also akkumuliert in keinem Fall mehr Iterationen zu übertragen als zu berechnen.

- Der **günstigste Fall** liegt dann vor, wenn bereits im ersten Schritt jeder Prozessor seine eigenen Iterationen und keine anderen gefunden hat. Ein Zusammenführen von Teillisten ist dann nicht mehr erforderlich.

- In einem **durchschnittlichen Fall** erhält jeder Prozessor gleich viele Iterationen. Diese sind gleichmäßig über den Iterationsbereich verteilt. In diesem Fall bleibt die Gesamtzahl der Listeneinträge, die in jeder Stufe zu übertragen sind, konstant $\equiv \frac{N}{P}$. Es ergibt sich ein akkumulierter Übertragungsaufwand von $\lceil ld(P) \rceil * \frac{N}{P}$.

Der Aufwand für die Kommunikation setzt sich aus der Startup-Zeit und der Zeit für die Datenübertragung zusammen. Für die folgenden Abschätzungen wird eine Message-Passing-Startup-Zeit von 90 μs und eine Datenübertragungsrate von 50 $\frac{MByte}{s}$ vorausgesetzt. Diese während der ASVM-Testzeit ermittelten Werte fallen ungünstiger aus als die während des Normalbetriebs erreichbaren (ca. 40 μs Startup-Zeit; 80 $\frac{MByte}{s}$ Datenübertragung). Dies ist damit zu begründen, daß in der aktuell vorliegenden Version des Betriebssystems ASVM nicht mit dem Kommunikationskoprozessor zusammenarbeitet. Eine Verbesserung ist für die folgende Version des Betriebssystems angekündigt.

Die akkumulierte Startup-Zeit in der Banyan-Struktur beträgt damit für P Prozessoren ca. $\lceil ld(P) \rceil * 90 \mu s$. Tabelle 4.1 kann man eine Auswertungszeit für die Align-Funktion von mindestens $1,4 \mu s$ entnehmen. Dies bedeutet hier, daß in der Zeit, die durch das Aufsetzen der Kommunikation insgesamt verbraucht wird, höchstens $\lceil ld(P) \rceil$ * 64,3 Iterationen sequentiell untersucht werden könnten. Diese Zahl fällt kaum ins Gewicht, denn wenn P Prozessoren eingesetzt werden, sollte die Problemgröße so gewählt sein, daß jeder Prozessor mindestens die Daten einer Seite zu bearbeiten hat. Die wenigsten Iterationen ergeben sich, wenn Double-Precision-Werte bearbeitet werden. Eine Speicherseite der Große 8192 Byte nimmt genau 1024 Double-Precision-Werte auf, so daß der Iterationsraum insgesamt dann mindestens P * 1024 Elemente groß ist. Dies bedeutet einen Anteil von höchstens 6,3 * $\frac{\lceil ld(P) \rceil}{P}$ % für die akkumulierte Message-Passing-Startup-Zeit. Es ergibt sich ein maximaler Overhead für P = 2 Prozessoren von 3,15 %. Mit wachsender Prozessorzahl nimmt dieser relative Anteil noch weiter ab.

Ebenso sind die Kosten der reinen Datenübertragung akzeptabel. Bei der aktuell realisierten Übertragungsrate von 50 $\frac{MByte}{s}$ ergeben sich pro Iterationsblock, für den 2 Integer-Werte, also 8 Byte, zu übertragen sind, zusätzlich 155 ns. Die Datenübertragungszeiten dürfen also, selbst für den oben diskutierten ungünstigsten Fall, im Vergleich zum Aufwand, den das Auswerten der Align-Funktion mit sich bringt, vernachlässigt werden.

Bei der Bewertung des Aligned-Scheduling ist der zusätzliche Aufwand, den das Auslesen der Page-Owner-Tabelle und das Invertieren der Align-Funktion mit sich bringt, dem Vorteil eines lokalen Datenzugriffs gegenüberzustellen. Angesichts von Page-Fault-Zeiten um ca. 2,5 ms fällt diese Gegenüberstellung eindeutig zu Gunsten des Aligned-Scheduling aus. Dies gilt insbesondere deswegen, weil das Aligned-Scheduling zumindest beim Zugriff auf das Datenfeld, an dem das Schleifen-Scheduling ausgerichtet wird, ein Page-Trashing garantiert verhindert.

Allerdings ist das Aligned-Scheduling nicht in jeder Situation das optimale Verfahren. Um z.B. eine ausgewogene Lastverteilung zu erreichen, ist es völlig ungeeignet. Befinden sich gerade alle Datenseiten bei einem einzigen Prozessor, so führt das Aligned-Scheduling dazu, daß diesem Prozessor die gesamte Arbeit zugewiesen wird.

8 Aligned-Self-Service-Scheduling

Bisher wurden die Schleifen-Scheduling-Verfahren Self-Service-Scheduling und Aligned-Scheduling vorgestellt. Das **Self-Service-Scheduling** hat sich als geeignetes Verfahren zur Realisierung einer dynamischen Lastverteilung erwiesen. Als Nachteil hat sich herausgestellt, daß die Aushändigung von Schleifeniterationen ohne jede Berücksichtigung der gerade vorliegenden Datenverteilung erfolgt. Gerade diese Berücksichtigung der vorliegenden Datenverteilung ist jedoch die Stärke des **Aligned-Scheduling**. Das Schleifen-Scheduling wird dabei an die vorgefundene Datenverteilung angepaßt, damit sich Lokalität beim Datenzugriff einstellt. Der Nachteil des Aligned-Scheduling besteht jedoch darin, daß es keine Möglichkeit vorsieht, eine dynamische Lastadaption vorzunehmen. Da weder auf effizienten Datenzugriff noch auf gleichmäßige Lastverteilung verzichtet werden kann, liegt es nahe, beide Verfahren miteinander zu verbinden. Das resultierende Verfahren soll im folgenden als **Aligned-Self-Service-Scheduling** bezeichnet werden, um zu verdeutlichen, daß es Anteile aus beiden Strategien vereinigt.

8.1 Der statische Anteil

Wenn man Lokalität beim Datenzugriff erreichen möchte, muß man zunächst die aktuell vorliegende Datenverteilung kennen. Der erste Schritt ist also, wie beim Aligned-Scheduling, das Auslesen der lokalen Page-Owner-Tabelle. Daran schließt sich die Bestimmung der globalen Seitenverteilungsinformation bei einem zentralen Prozessor an, damit der Fall fehlender und doppelt vorhandener Seiten behandelt werden kann. Für die Zuordnung von Seiten zu Prozessoren ist also nicht die Aussage der Page-Owner-Tabelle verbindlich, sondern die Verteilung, die der zentrale Prozessor in Kenntnis der globalen Seitenverteilungsinformation vorgibt. Nachdem eine eindeutige Zuordnung von Speicherseiten auf Prozessoren vorliegt, schließt sich die Invertierung der Align-Funktion an, so daß anschließend jeder Prozessor die Iterationen kennt, die auf seine lokal vorhandenen Seiten zugreifen. Anders als beim Aligned-Scheduling, bearbeiten die Prozessoren jetzt aber nicht direkt die lokal vorhandenen Iterationen, sondern an dieser Stelle beginnt der dynamische Anteil des Verfahrens.

8.2 Der dynamische Anteil

Bei dem dynamischen Lastverteilungsansatz, den das Self-Service-Scheduling realisiert, werden Arbeitspakete immer dann zugeteilt, wenn ein Prozessor mit den ihm zuvor zugewiesenen Aufgaben fertig ist. Dieser Ansatz wird beibehalten, auch wenn nicht mehr

Iterationsblöcke, sondern Bearbeitungsaufträge ausgehändigt werden. Ein solcher Bearbeitungsauftrag besagt, daß die Iterationen zu bearbeiten sind, die auf Daten einer bestimmten Speicherseite zugreifen. Zusammen mit einem Bearbeitungsauftrag wird die Nummer des Prozessors mitgeteilt, der über die betroffene Speicherseite verfügt. Ist der Iterationsraum groß genug, so können zur Reduktion des Nachrichtenaufkommens auch mehrere Bearbeitungsaufträge auf einmal ausgehändigt werden.

Wird dabei ein Bearbeitungsauftrag dem Besitzer einer Speicherseite zugewiesen, so liegt auch die Liste der zugehörigen Iterationen lokal vor, denn bei der Invertierung der Align-Funktion wurde diese Iterationsliste ja gerade bei dem Besitzer der Speicherseite abgelegt. Anderenfalls muß diese Liste noch von dem Besitzer der Seite angefordert werden. Wer dieser Besitzer ist, muß nicht extra herausgefunden werden, denn diese Information wurde ja zusammen mit dem Bearbeitungsauftrag gegeben.

Man erkennt, daß ein Auftrag zur Bearbeitung nicht-lokaler Daten deutlich mehr Zeit in Anspruch nimmt, als ein Auftrag, der lokal vorhandene Daten betrifft. Bevor mit der Berechnung begonnen werden kann, muß das Betriebssystem die adressierten Daten und der Scheduler die zugehörigen Iterationen zur Verfügung stellen. Es wird deshalb eine wichtige Aufgabe für den Scheduler sein, diese Bearbeitungsaufträge so zu verteilen, daß die vorgefundene Seitenverteilung berücksichtigt wird.

Nach der Vorarbeit, die im statischen Teil des Verfahrens geleistet worden ist, sind nun die Voraussetzungen für eine Berücksichtigung der Seitenverteilung gegeben. In der Wurzel des Schedulers liegt die Information darüber vor, bei welchem Prozessor sich eine bestimmte Speicherseite befindet. Darüber hinaus weiß jeder innere Knoten aufgrund des Aufbaus des Scheduler-Baumes, welche Prozessoren seine rechten bzw. linken Nachfolger sind. Mit Hilfe dieses Wissens kann er entscheiden, ob ein Bearbeitungsauftrag besser an den rechten oder an den linken Nachfolger weiterzuleiten ist. Fordert ein Nachfolger weitere Bearbeitungsaufträge an, so werden ihm in erster Linie Aufträge zugewiesen, die Speicherseiten in seinem Teilbaum betreffen. Erst wenn keine 'passenden' Aufträge mehr vorhanden sind, werden auch Aufträge vergeben, die Speicherseiten in dem anderen Teilbaum betreffen.

Bei dem in Abbildung 8.1 gegebenen Beispiel sind die Speicherseiten 1 bis 11 auf vier Prozessoren verteilt. Die Zahlen in eckigen Klammern unterhalb des ersten Bildes geben diese Verteilung an. Obwohl in Wirklichkeit ein asynchroner Ablauf vorliegt, wurde hier zur Veranschaulichung eine Darstellung in vier Zeitschritten gewählt. Im ersten Zeitschritt sind noch alle elf Aufträge von der Wurzel des Scheduler-Baumes zu vergeben. Da die Wurzel die Information besitzt, welche Seiten sich in den beiden Unterbäumen befinden, kann sie die ersten Aufträge unter Berücksichtigung der Seitenverteilung aushändigen. Dies gelingt auch im zweiten und dritten Schritt, da die vorhandenen Speicherseiten gleichmäßig auf die beiden Unterbäume verteilt sind. Innere Knoten im Scheduler-Baum verfahren im Prinzip ebenso. Der Unterschied besteht darin, daß sie weniger Aufträge auf einmal aushändigen, weil das Gewicht ihrer Nachfolger geringer ist und darin, daß sie selbst von ihren Vorgängern weitere Aufträge anfordern müssen. Interessant ist der dritte dargestellte Zeitschritt. Hier wird an den Blattknoten Nr. 2 der Auftrag für die achte Speicherseite vergeben, obwohl sich diese Seite nicht bei ihm befindet. Dies geschieht mit dem Ziel einer ausgeglichenen Lastverteilung. Anderenfalls könnte Prozessor Nr. 2 nur die Iterationen der Speicherseite Nr. 3, die sich zunächst als einzige bei ihm befindet, bearbeiten.

Mit der beschriebenen Methode wird die Strategie verfolgt, so weit wie möglich lokal zu rechnen, und erst dann, wenn es der dynamische Lastausgleich erfordert, die Kosten für

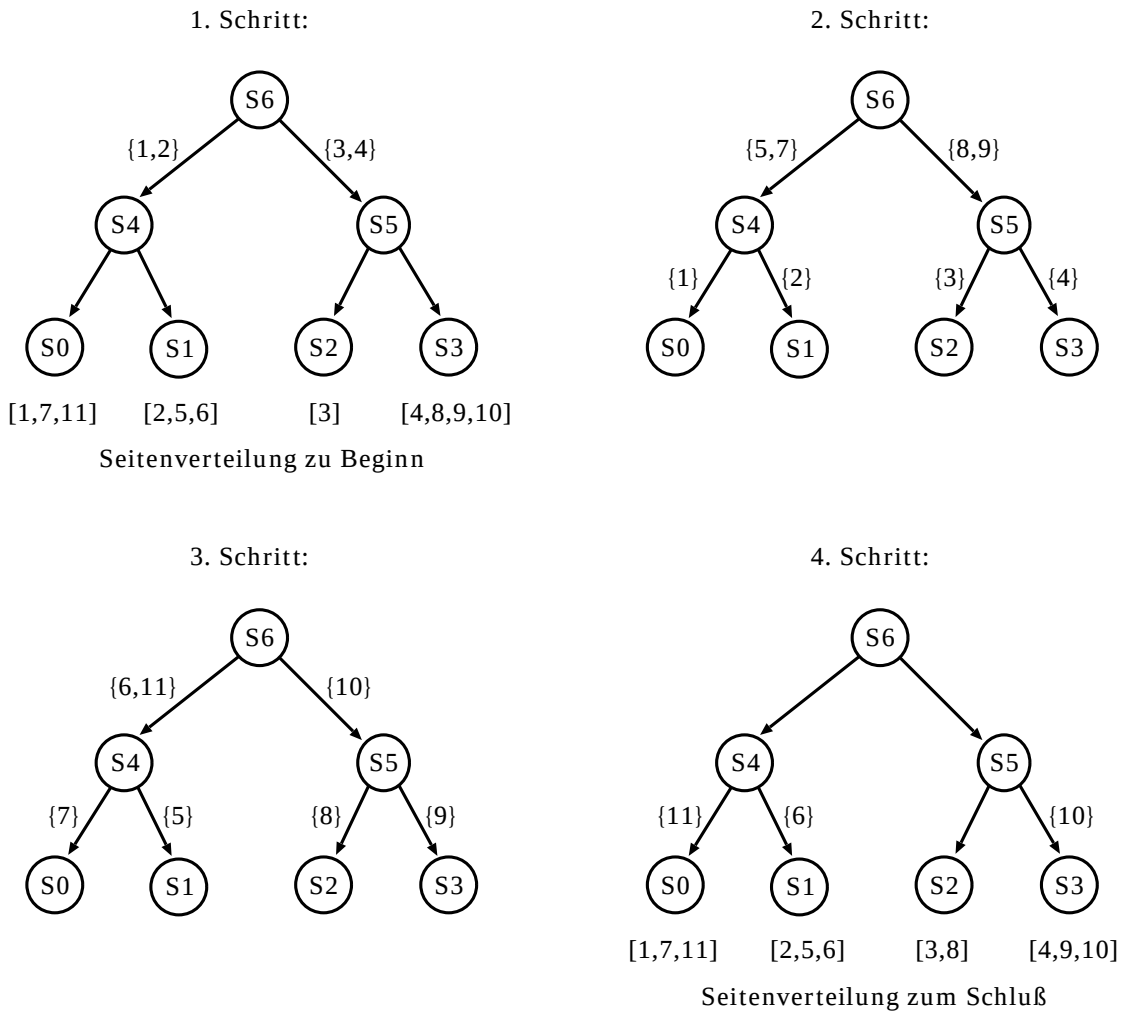


Abbildung 8.1: Ein möglicher Ablauf des Aligned-Self-Service-Scheduling: Die eckigen Klammern unter den Blattknoten geben an, welche Seiten sich bei einem Prozessor befinden. Die Markierung der Pfeile besagt, welche Bearbeitungsaufträge erteilt werden.

einen Datentransport zu investieren. An dieser Stelle kommt der Vorteil von SVM, nämlich die automatische Datenmigration, besonders zum Tragen. Allein durch den Zugriff auf eine nicht-lokale Speicherseite wird der Transportmechanismus angestoßen. Im Vergleich zu compiler-basierten Realisierungen eines gemeinsamen Adreßraumes [Han93] kann eine aufwendige Programmierarbeit zur Datenumverteilung hier eingespart werden.

8.3 Anwendung auf iterative Verfahren

Im Beispiel aus Abbildung 8.1 ist zu erkennen, daß sich die Datenverteilung der dynamischen Arbeitsverteilung anpaßt. Die Speicherseite Nr. 8 ist durch die Zuweisung eines entsprechenden Bearbeitungsauftrages von Prozessor 3 zu Prozessor 2 verlagert worden. Würde man nun dieselbe Schleife ein zweites Mal über das Aligned-Self-Service-Scheduling

verteilen, so wäre eine geeignete Datenverteilung schon gegeben. Es würde sich also ein Ablauf mit einer ausgeglichenen Lastverteilung ergeben, ohne daß eine erneute Datenumverteilung nötig wäre.

Diese Beobachtung eröffnet die Möglichkeit, das Aligned-Self-Service-Scheduling für technisch-wissenschaftliche Anwendungsprogramme zu optimieren, bei denen häufig parallele Schleifen mehrfach iteriert werden. Zeitaufwendige Abläufe sollen dabei so weit wie möglich aus der Bearbeitung einer parallelen Schleife nach außen verlagert werden. Arbeiten, die nicht jedesmal wieder neu durchgeführt werden müssen, sind:

1. Die Bestimmung der globalen Seitenverteilungsinformation
2. Die Invertierung der Align-Funktion
3. Die vollständige Datenumverteilung
4. Die vollständige Iterationslistenumverteilung

Eine Optimierung kann auf zwei unabhängigen Ebenen erfolgen. Zum einen ist unter Umständen eine dynamische Verteilung von Arbeitspaketen nicht bei jedem Iterationsschritt erforderlich, und zum anderen kann man bei der dynamischen Verteilung selbst Berechnungszeit einsparen, wenn man auf Informationen zurückgreift, die noch aus dem vorangegangenen Verteilungsprozeß zur Verfügung stehen.

8.3.1 Die DISTRIBUTE-ONCE-Direktive

Über die DISTRIBUTE-ONCE-Direktive kann der Programmierer angeben, daß beim ersten Ablauf einer parallelen Schleife ein dynamisches Lastausgleichsverfahren durchgeführt werden soll. Jeder Prozessor speichert dabei die Iterationen, die er bearbeitet hat. Beim zweiten und bei jedem weiteren Ablauf derselben Schleife führt dann jeder Prozessor genau diejenigen Iterationen aus, die er auch schon beim ersten Mal ausgeführt hat. Der Overhead, den ein dynamisches Verteilen der Arbeitslast mit sich bringt, kann so eingespart werden. Dieses Vorgehen ist aber nur dann erfolgversprechend, wenn der Zeitbedarf für die Bearbeitung zugewiesener Iterationen im Lauf der Berechnung nur marginal variiert. Um eine Änderung der Lastverteilung dennoch berücksichtigen zu können, ist es sinnvoll, eine dynamische Arbeitsverteilung nicht für jede Iteration, sondern erst nach einigen Iterationsschritten erneut vorzunehmen, wie es in Abbildung 8.2 dargestellt ist.

8.3.2 Optimierung für wiederholtes dynamisches Scheduling

Ist eine dynamische Arbeitsverteilung erforderlich, damit eine veränderte Lastverteilung berücksichtigt werden kann, so kann der Ablauf der Arbeitsverteilung dadurch optimiert werden, daß einmal berechnetes 'Wissen' gespeichert wird, um es gegebenenfalls wiederverwenden zu können.

Bei einem iterativen Verfahren mit dynamischer Lastadaption genügt es, wenn die Bestimmung der globalen Seitenverteilungsinformation und die Invertierung der Align-Funktion nur ein einziges Mal während der Initialisierung vorgenommen werden. Da sich die Align-Funktion für eine Schleife nicht ändern kann, bleibt auch die Liste der Iterationen, die

```
 1 : 100      CONTINUE
 2 :          DO J = 1, NUM_SAME_SCHEDULE
 3 : CSVM$        PDO (STRATEGY, DISTRIBUTE_ONCE(TEMPL(I),
 4 : CSVM$  +                          GRAB_ALIGNED(A(F(I)))))
 5 :              DO I = 1, N
 6 :                A(F(I)) = ...
 7 :              ENDDO
 8 :          ENDDO
 9 : CSVM$    UNDEF :: TEMPL
10 :          IF (RESIDUUM .GT. EPSI) THEN GOTO 100
```

Abbildung 8.2: Dynamische Arbeitsverteilung in einem iterativen Programm

gemeinsam auf eine bestimmte Speicherseite zugreifen, unverändert. Lediglich die Position dieser Liste wird sich verlagern, wenn auch die betroffene Speicherseite selbst verschoben wird. Falls die Scheduler-Knoten jedesmal, wenn sie einen Bearbeitungsauftrag aushändigen, auch ihre gespeicherte Seitenverteilung aktualisieren, kann die Bestimmung der globalen Seitenverteilungsinformation zu Beginn eines weiteren Ablaufs des Aligned-Self-Service-Scheduling eingespart werden, weil diese Information aus der vorangegangenen Iteration noch gegeben ist.

Darüber hinaus ist festzuhalten, daß das Aligned-Self-Service-Scheduling darauf ausgelegt ist, eine Seitenumverteilung und damit verbunden eine Umverteilung der zugehörigen Iterationslisten nur dann durchzuführen, wenn dies zur Realisierung einer gleichmäßigen Lastverteilung unverzichtbar ist.

Insgesamt erweist sich das Aligned-Self-Service-Scheduling als besonders geeignet für iterative Programme, bei denen wiederholt Konfigurationen zu berechnen sind, die den Zustand eines untersuchten dynamischen Systems repräsentieren. In den meisten technisch-wissenschaftlichen Anwendungen liegt dabei ein stetiger Übergang von einer Konfiguration auf die nächste vor. Wenn sich also die Folgekonfiguration nur wenig von ihrer Vorgängerin unterschiedet, kann man davon ausgehen, daß eine ausgeglichene Lastverteilung, die sich in Verbindung mit einer Lokalität beim Datenzugriff eingestellt hat, während einiger Iterationsschritte erhalten bleibt. Eine aufwendige Neuverteilung ist deshalb immer erst nach einigen Iterationen erforderlich. Ist eine Neuverteilung zur Lastadaption unvermeidlich, so kann sie konservativ erfolgen, d.h. es wird nur dort umverteilt, wo es zur Wahrung einer gleichmäßigen Lastverteilung unvermeidbar ist. Die meisten Arbeitspakete werden dort bearbeitet, wo sie schon zuvor bearbeitet wurden, damit Datenübertragungszeiten minimiert werden.

9 Zusammenfassung und Ausblick

Eine hohe Auslastung aller Prozessoren während des gesamten Programmlaufs ist unabdingbar, um das Potential moderner massiv-paralleler Rechnersysteme voll zur Beschleunigung eines einzelnen Anwendungsprogramms nutzen zu können. Bei einigen Anwendungsprogrammen gelingt dies relativ leicht, indem eine statische Einteilung in gleich große Arbeitspakete vorgenommen werden kann. Bei Simulationsprogrammen für komplexe nichtlineare Systeme ist dies jedoch nicht immer möglich, weil deren Dynamik oft dazu führt, daß sich Bereiche hoher Rechenintensität kontinuierlich verlagern.

In dieser Situation kann eine Berücksichtigung der dynamischen Arbeitslastverteilung nur durch eine ebenfalls dynamische Zuteilung von Arbeitspaketen auf Prozessoren erfolgen.

Dynamische Lastausgleichsverfahren für massiv-parallele Anwendungsprogramme, die heute schon existieren, sind immer explizit für das gerade betrachtete Problem konzipiert. Um den Programmierer von der Aufgabe zu entlasten, für jedes neue Problem auch ein neues Scheduling-Verfahren selbst entwickeln zu müssen, wurden in dieser Arbeit für die Programmiersprache SVM-Fortran dynamische Loop-Scheduling-Verfahren entwickelt.

Die ungleiche Verteilung der Arbeitslast auf einzelne Bereiche resultiert zumeist aus Anwendungen mit irregulären Datenverteilungen, bei denen indirekt auf Datenfelder zugegriffen werden muß. Um in SVM-Fortran eine parallele DO-Schleife mit indirektem Feldzugriff effizient ausführen zu können, müssen die Schleifeniterationen dort ausgeführt werden, wo auch die referenzierten Feldelemente lokal vorhanden sind. Es wurde deshalb eine Datenstruktur realisiert, die eine beliebige Zuordnung von Schleifeniterationen auf Prozessorelemente speichern kann.

Die Umsetzung des dynamischen Schleifen-Scheduling auf massiv-parallelen Systemen basiert wesentlich auf dem Prinzip des Self-Service-Scheduling [FeRu90]. Bei diesem Ansatz greift ein Prozessor, der seinen letzten Bearbeitungsauftrag vollständig erledigt hat, auf einen gemeinsamen Vorrat zu, um ein neues zur Bearbeitung anstehendes Arbeitspaket zu erhalten.

Dieses Prinzip hat sich auf Parallelrechnern mit physikalisch gemeinsamem Hauptspeicher bewährt, da dort ein gemeinsames Register oder eine Stelle im gemeinsamen Hauptspeicher genutzt werden kann, um die Menge verbliebener Arbeitspakete effizient zu verwalten. Diese Mittel stehen in netzwerkgekoppelten Systemen jedoch nicht zur Verfügung. Die Kommunikation zwischen Prozessoren dauert zu lange, so daß ein feingranulares Schleifen-Scheduling zunächst als nicht realisierbar erscheint. Darüber hinaus würde in massiv-parallelen Rechnern der konkurrierende Zugriff auf eine einzige Menge zu verteilender Arbeitspakete einen Engpaß darstellen. Gelöst wurden diese Probleme durch den Übergang auf einen hierarchischen Scheduler. Diese als Baum organisierte Verteilungsstruktur gestattet die verteilte Verwaltung der Menge unerledigter Arbeitspakete. Der

Scheduler wurde mit Hilfe des asynchronen Message-Passing realisiert, so daß Berechnung und Kommunikation überlappen können. Dadurch wird es möglich, daß die relativ langsame Kommunikation den Fortgang der Berechnung typischerweise nur noch während der Startphase verzögert. Mit diesem hierarchischen Scheduler liegt eine Struktur vor, die leicht auf sehr große Prozessorzahlen übertragen werden kann. Die Realisierung ist nicht zwingend auf das Message-Passing festgelegt; es kann der jeweils schnellste Kommunikationsmechanismus genutzt werden. Voraussetzung bleibt jedoch die Möglichkeit der Programmierung unabhängiger Threads, so daß das Scheduling und die Berechnungen in den Arbeitspaketen zeitlich entkoppelt ablaufen können.

Ein weiterer Aspekt dieser Arbeit war die Berücksichtigung der besonderen Anforderungen, die Systeme mit virtuell gemeinsamen Speicher an das Schleifen-Scheduling stellen. Da Daten in diesen Systemen typischerweise in Seiten-Einheiten verwaltet werden, sollten Arbeitspakete, die gemeinsam auf Daten in einer Speicherseite zugreifen, auch gemeinsam einem Prozessor zugewiesen werden. Ein Datentransport kann nur dann vermieden werden, wenn dort gerechnet wird, wo die benötigten Daten lokal vorliegen. Das Aligned-Scheduling ist ein Verfahren, das diese Forderungen erfüllt. Dieses bisher lediglich theoretisch untersuchte Konzept [Lin94] konnte im Rahmen der hier vorliegenden Arbeit realisiert werden. Probleme der konkreten Umsetzung wurden aufgezeigt und Lösungsmöglichkeiten beschrieben.

Basierend auf dem Self-Service-Scheduling und dem Aligned-Scheduling konnte schließlich mit dem Aligned-Self-Service-Scheduling ein Verfahren gefunden werden, das die Vorteile eines lokalen Datenzugriffs mit einer dynamischen Lastanpassung vereinigt. Dieses Verfahren hat sich als besonders geeignet erwiesen für iterative Programme, bei denen wiederholt Konfigurationen zu berechnen sind, die den Zustand eines untersuchten dynamischen Systems repräsentieren. Es hat sich herausgestellt, daß eine aufwendige Umverteilung der Arbeitspakete nur relativ selten erforderlich ist. Der Ablauf konnte durch die Wiederverwendung von Informationen, die noch aus dem bisherigen Programmlauf zur Verfügung stehen, optimiert werden. Die meisten Arbeitspakete können dort bearbeitet werden, wo sie schon zuvor bearbeitet wurden, Datenübertragungszeiten werden so minimiert.

Insgesamt waren bei den beschriebenen Verfahren drei Ziele miteinander in Einklang zu bringen:

- ausgeglichene Lastverteilung;
- geringer Overhead;
- Berücksichtigung von Lokalität beim Datenzugriff.

Es hat sich gezeigt, daß die Berücksichtigung von Lokalität beim Datenzugriff das wichtigste Ziel ist, da die Page-Fault-Zeiten im Vergleich zur Taktzeit des Prozessors sehr hoch sind. Darauf aufbauend kann dann eine ausgeglichene Lastverteilung realisiert werden. Der Overhead wurde durch eine Parallelisierung der notwendigen Abläufe und durch eine weitgehende Wiederverwendung einmal berechneter Information so weit wie möglich reduziert. Dennoch kann der Zeitbedarf zusätzlicher Arbeit nicht vernachlässigt werden. Ein weiteres Problem besteht darin, daß die Zeiten für einen Seitentransport mit in die Berechnungszeit eingehen. Ist innerhalb eines Arbeitspaketes nur so wenig Arbeit zu leisten, daß dessen Berechnung nicht wesentlich länger dauert als ein Page-Fault, so kann sich schon bei der nächsten Iteration ein anderes Laufzeitverhalten einstellen. In der Zeit, die zuvor

damit verbracht werden mußte, auf eine Speicherseite zu warten, kann der Prozessor jetzt ein weiteres Arbeitspaket bearbeiten. Dies führt dazu, daß erst einige Konvergenzschritte notwendig sind, bis sich schließlich eine ausgeglichene Lastverteilung einstellen wird.

Auch wenn die realisierten Verfahren nicht so feingranular eingesetzt werden können wie vergleichbare Verfahren bei Rechnern mit physikalisch gemeinsamem Speicher, erscheint deren Anwendung dennoch lohnend. Bei Problemen mit einer dynamischen Veränderung der Arbeitslastverteilung kann auch ein nicht optimales dynamisches Lastausgleichsverfahren vorhandene Ressourcen besser nutzen als ein statisches.

Die vorgestellten Verfahren konnten im Rahmen dieser Arbeit theoretisch und an Hand einfacher Testbeispiele bewertet werden. Die weitere Anwendung auf Programme im Produktionsbetrieb erscheint deshalb sinnvoll, um zu untersuchen, ob die Hoffnungen auf einen effizienten Ablauf auch den Praxisanforderungen standhalten.

Aufgrund der Komplexität der notwendigen Verfahren konnte dynamisches Loop-Scheduling zunächst nur für eine Ebene von geschachtelten Schleifen verwirklicht werden. Dies reicht für einen großen Teil der Anwendungen aus. Falls dennoch Bedarf auch für geschachteltes dynamisches Loop-Scheduling deutlich werden sollte, können entsprechende Erweiterungen vorgenommen werden.

Mittelfristig werden massiv-parallele Rechnersysteme nicht mehr ausschließlich dazu benutzt werden, einzelne Anwendungsprogramme in möglichst kurzer Zeit zu bearbeiten, sondern der durchsatzorientierte Aspekt in Rechenzentrumsumgebungen wird stärkere Beachtung finden. Wenn dazu die Voraussetzungen auf der Seite des Betriebssystems geschaffen sein werden, können in einem weiteren Schritt die Ergebnisse dieser Arbeit dazu beitragen, ein kooperatives Scheduling für massiv-parallele Systeme mit verteiltem Speicher zu realisieren.

Literaturverzeichnis

[Amd67] G.M. Amdahl, *Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities*, Proc. AFIPS, Vol. 30, Atlantic City (N.J.), AFIPS Press 1967, 483-485

[BCZ90] J. Bennnet, J. Carter, and W. Zwaenepoel, *Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence*, Proc. 2nd ACM SIGPLAN Symp. on Principles and Practice of Par. Programming, Seatle, WA, 1990, 168-176

[Bel92] G. Bell, *Ultracomputers: A Teraflop Before its Time*, CACM, Vol. 35, No. 8, 1992, 27-47

[BePo89] C.F. Beckmann and C.D. Polychronopoulos, *The Effect of Barrier Synchronization and Scheduling Overhead on Parallel Loops*, Proc. Int. Conf. Parallel Processing, Vol. II, St. Charles, IL, 1989, 200-204

[BGM95] R. Berrendorf, M. Gerndt, and M. Mairandres, *Programming Shared Virtual Memory on the Intel Paragon^{TM} Supercomputer*, Proc. ICS'95, 1995 to appear

[BGNP94] R. Berrendorf, M. Gerndt, W.E. Nagel, and J. Prümmer, *SVM-Fortran (Overview)*, Forschungszentrum Jülich, KFA-ZAM-IB-9322, 1994

[CMZ93] B. Chapman, P. Mehrotra, and H. Zima, *High Performance Fortran Without Templates: An Alternative Model for Distribution and Alignment*, University of Vienna, Techn. Report ACPC/TR 93-17, 1994

[CMZ94] B. Chapman, P. Mehrotra, and H. Zima, *Extending HPF For Advanced Data Parallel Applications*, University of Vienna, Technical Report ACPC/TR 94-7, 1994

[Con90] *CONVEX Fortran Language Reference Manual*, CONVEX Computer Corporation, 1990

[DSB86] M. Dubois, C. Scheurich, and F. Briggs, *Memory Access Buffering in Multiprocessors*, Proc. 13th Ann. Int. Symp. on Computer Architecture, 1986, 434-442

[DSB88] M. Dubois, C. Scheurich, and F. Briggs, *Synchronization, Coherence and Event Ordering in Multiprocessors*, IEEE Computer, Vol. 21, No. 2, 1988, 9-21

[FeRu90] D.G. Feitelson and L. Rudolph, *Distributed Hierarchical Control for Parallel Processing*, IEEE Computer, Vol. 23, No. 5, 1990, 65-77

[GaPe94] K.E. Gates and W.P. Petersen, *A Technical Description of Some Parallel Computers*, International Journal of High Speed Computing, Vol. 6, No. 3, 1994, 399-449

[GGH91] K. Gharachorloo, A. Gupta, and J. Hennessy, *Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors*, SIGPLAN Notices Vol. 26, No. 4, 1991, 245-256

[Gil94] W.K. Giloi, *Multiple Execution Thread Architecture - A New Approach*, Proc. Parcella'94, 4th Int. Workshop on Parallel Processing by Cellular Automata and Arrays, Potsdam, 1994, Akademie Verlag, 27-37

[GrWi93] E.D. Granston and H.A.G. Wijshoff, *Managing Pages in Shared Virtual Memory Systems: Getting the Compiler into the Game*, Proc. Int. Conf. Supercomputing, Tokyo, 1993, 11-20

[Han93] R. von Hanxleden, *Handling Irregular Problems with Fortran D – A Preliminary Report*, Proc. 4th Workshop on Par. Computers, Delft, Netherlands, 1993, also via ftp: pub/CRPC-TRs/reports/CRPC-TR-93339-S at softlib.rice.edu

[Hel90] H. Hellwagner, *A Survey of Virtually Shared Memory Schemes*, SFB-Bericht Nr. 342/33/90 A, Sonderforschungbereich 342, Technische Universität München, 1990

[HKKK92] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.W. Tseng *An Overview of the Fortran D Programming System*, Proc. 4th Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, 1991

[HPF93] High Performance FORTRAN Forum: *High Performance FORTRAN Language Specification - PART I, Version 1.0*, ACM Fortran Forum, Vol. 12, No. 4, 1993

[HPF94] High Performance FORTRAN Forum: *High Performance FORTRAN Language Specification - PART II, Version 1.0*, ACM Fortran Forum, Vol. 13, No. 2, 1994

[HSF91] S. Flynn Hummel, E. Schonberg, and L.E. Flynn, *Factoring: A Practical and Robust Method for Scheduling Parallel Loops*, Proc. Supercomputing '91, Albuquerque, NM, 1991, ACM, 610-619

[HSF92] S. Flynn Hummel, E. Schonberg, and L.E. Flynn, *Factoring: A Practical and Robust Method for Scheduling Parallel Loops*, CACM, Vol. 35, No. 8, 1992, 90-101

[HuLi89] P. Hudak and K. Li, *Memory Coherence in Shared Virtual Memory Systems*, ACM Trans. Comp. Syst., Vol. 7, No. 4, 1989, 321-359

[HwBr84] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984, 494-497

[IBM91] *IBM VS FORTRAN Version 2: Programming Guide for CMS and MVS (Release 5)*, IBM, SC-26-4222-5, 1991

[Int94] *Paragon User's Guide*, Intel Supercomputer Systems Division, 1994

[KrWe85] C.P. Kruskal and A. Weiss, *Allocating Independent Subtasks on Parallel Processors*, IEEE Trans. Softw. Eng., Vol. SE-11, No. 10, 1985, 1001-1016

[KSR] Kendall Square Research, *KSR 1 Technical Summary*, 1992

[Li86] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. thesis, Yale, 1986

[Lin94] M. Linn, *Untersuchungen zur Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamem Speicher*, Bericht des Forschungszentrums Jülich Jül-2931, 1994

[LNVD91] M. Linn, W.E. Nagel, M. Vaeßen, and U. Detert, *UNICOS 5.1 vs. UNICOS 6.0: Performance Evaluation of Autotasking*, Proc. Cray User Group Meeting, Fall 1991, Santa Fe, also: Interner Bericht KFA-ZAM-9112

[Nag90] W.E. Nagel, *Exploiting Autotasking on a CRAY Y-MP: An improved Software Interface to Multitasking*, Parallel Computing, Vol. 13, No. 2, 1990, 225-234

[Nag93] W.E. Nagel, *Ein verteiltes Scheduler-System für Mehrprozessorrechner mit gemeinsamem Speicher: Untersuchungen zur Ablaufplanung von parallelen Programmen*, Bericht des Forschungszentrums Jülich Jül-2850, 1993

[Oed93] W. Oed, *Das Cray Research massiv-parallele Prozessorsystem CRAY T3D*, Internal Report, Cray Research GmbH, Riesstraße 25, 80922 München, 1993

[PDM92] D.M. Pase, T. MacDonald, and A. Meltzer, *MPP Fortran Programming Model*, Cray Research Inc., 655F Lone Oak Drive, Eagan, Minnesota 55121, 1992

[Plu94] E. Plümäkers, *Integration von benutzergesteuertem Scheduling in einem Präprozessor für SVM-Fortran*, Bericht des Forschungszentrums Jülich, 1995 to appear (Diplomarbeit RWTH Aachen)

[PoKu87] C.D. Polychronopoulos and D.J. Kuck, *Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers*, IEEE Trans. Comput., Vol. C-36. No. 12, 1987, 1425-1439

[SWLE92] M. Simmons, H. Wassermann, O. Lubeck, C. Eoyang, R. Mendez, H. Harada, and M. Ishiguro, *A Performance Comparison of Four Supercomputers*, Comm. ACM, Vol. 35, No. 8, 1992, 116-123

[TaYe86] P. Tang and P.C. Yew, *Processor Self-Scheduling for Multiple Nested Loops*, Proc. ICPP'86, St. Charles (IL), 1986, 528-534

[WiRe93] M.H. Willebeek-LeMair and A.P. Reeves, *Strategies for Dynamic Load Balancing on Highly Parallel Computers*, IEEE Trans. Par. Distr. Syst., Vol. 4, No. 9, 1993, 979-993

[Zei93] S. Zeisset, *Evaluation and Enhancement of the Paragon Multiprocessor's Shared Virtual Memory System*, Technische Universität München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Diplomarbeit, 1993

[ZBCM92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, *Vienna Fortran - A Language Specification*, ICASE Internal Report 21, ICASE, Hampton, VA., 1992

Dank

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

Dem Inhaber des Lehrstuhls, Herrn Professor Hoßfeld, danke ich dafür, daß er mir dieses interessante Thema anvertraut hat. Die Arbeit hat wesentlich davon profitiert, daß ich in seinem Institut eine sehr gute technische Ausstattung nutzen konnte und auch auf eine kompetente Beratung nicht verzichten mußte. Herrn Dr. Nagel danke ich für seine Betreuung. Erst durch sein kritisches Urteil konnte die Arbeit die vorliegende Form annehmen. Ich danke Herrn Berrendorf dafür, daß er immer kurzfristig notwendige Erweiterungen des Compilers zur Verfügung gestellt hat. Herrn Gerndt danke ich für konstruktive Gespräche. Heinz Bast hat mich, wie alle anderen Nutzer der Intel Paragon auch, stets kompetent und ausdauernd unterstützt. Zu der äußerst angenehmen und konstruktiven Arbeitsatmosphäre hat entscheidend das sehr gute Verhältnis zu allen Diplomanden und Doktoranden beigetragen.

Schließlich möchte ich besonders herzlich meiner Freundin Holle dafür danken, daß sie insbesondere während der arbeitsreichen Schlußphase bereit war, im privaten Bereich auf vieles zu verzichten.