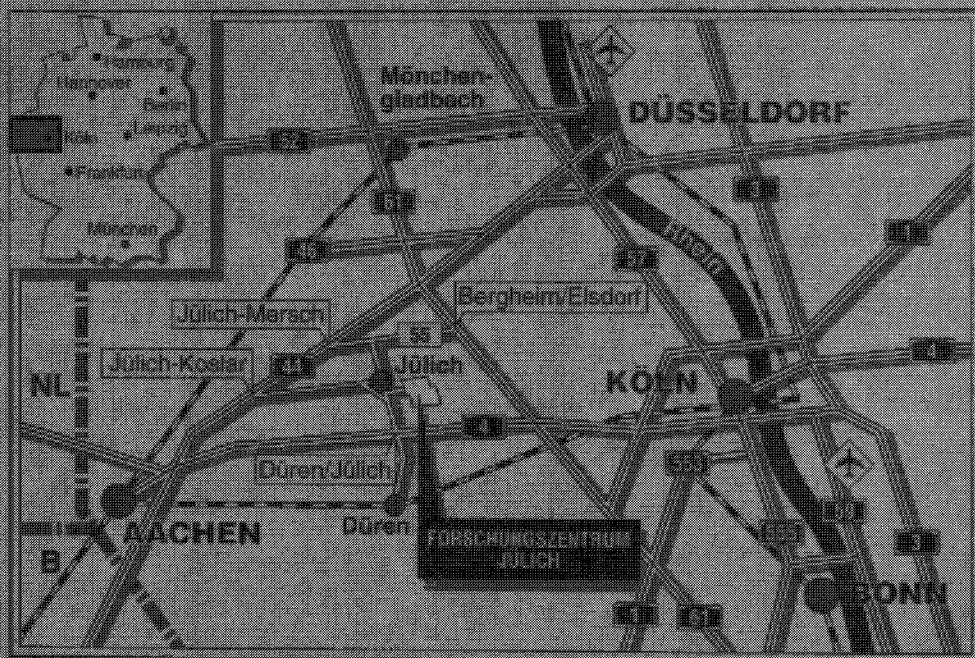


Zentralinstitut für Angewandte Mathematik

**Integration von benutzergesteuertem
Scheduling in einen Präprozessor für
SVM-Fortran**

Eric Plümakers



Berichte des Forschungszentrums Jülich ; 3111
 ISSN 0944-2952
 Zentralinstitut für Angewandte Mathematik JÜl-3111

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

Integration von benutzergesteuertem Scheduling in einen Präprozessor für SVM-Fortran

Eric Plümäkers

Kurzfassung

SVM-Fortran ist eine Erweiterung von Fortran77, die die Programmierung von Parallelrechnern mit virtuell gemeinsamem Speicher erlaubt. In SVM-Fortran ist die Steuerung der Datenlokalität über das Scheduling von Schleifen möglich. Der Benutzer spezifiziert für Schleifen eine Scheduling-Strategie. Die aus dem Scheduling resultierende Zuordnung der Schleifeniterationen zu den Prozessoren kann abgespeichert und bei nachfolgenden Schleifen mit gleichem Speicherzugriffsmuster wiederverwendet werden, um so eine Datenlokalität zu erreichen.

In dieser Arbeit werden die für reguläre Scheduling-Strategien wichtigen Parameter vorgestellt. Weiterhin wird ein Konzept erläutert, nach dem parallele Schleifen in eine neue Schleifenstruktur transformiert werden.

Abstract

SVM-Fortran is an extension to Fortran77 which allows the programming of parallel computers with shared virtual memory. Achieving data locality is possible in SVM-Fortran via loop scheduling. The user specifies a scheduling strategy for a loop. The resulting mapping of the scheduling from loop iterations to processors can be stored and reused in loops with similar memory access pattern.

In this paper we outline the important parameters for regular loop scheduling strategies and introduce a transformation concept for parallel loops.

Inhaltsverzeichnis

1	Einleitung	1
2	Programmierung von SVM-Systemen	3
2.1	Virtuell gemeinsamer Speicher	3
2.1.1	Kohärenzstrategien	4
2.1.2	Realisierung von SVM-Systemen auf Rechnern	5
2.2	Benutzergesteuertes Scheduling	7
2.3	SVM-Fortran	8
2.4	Einordnung der Diplomarbeit in das SVM-Fortran-Projekt	15
3	Datenverteilung und Scheduling in Fortran-Dialekten	17
3.1	Historische Entwicklung	18
3.2	Begriffe	19
3.2.1	Verteilungsstrategien	19
3.2.2	Alignment	23
3.2.3	Datenverteilungsmodell	23
3.3	Datenverteilung in HPF	25
3.4	Datenverteilung in Vienna Fortran	26
3.5	Datenverteilung in Fortran D	26
3.6	Scheduling in ANSI-X3H5	27
3.7	Vergleich der Konzepte	29
3.8	Weitere Arbeiten	31

4	Benutzergesteuertes Scheduling in SVM-Fortran	33
4.1	Direktes Schleifen-Scheduling	34
4.2	Vordefiniertes Schleifen-Scheduling	37
4.2.1	PROCESSORS-Direktive	39
4.2.2	TEMPLATE-Direktive	40
4.2.3	DISTRIBUTE- und REDISTRIBUTE-Direktive	41
4.2.4	Bewertung des vordefinierten Schleifen-Scheduling	46
4.3	Dynamisches Schleifen-Scheduling	48
5	Realisierung der Scheduling-Direktiven	52
5.1	Verteilte Datenstrukturen zur Administration	53
5.1.1	Template-Tabelle	54
5.1.2	Prozessorfeldtabelle	56
5.2	PROCESSORS-Direktive	57
5.3	TEMPLATE- und CREATE-Direktive	62
5.4	DISTRIBUTE- und REDISTRIBUTE-Direktive	64
5.5	UNDEF- und DESTROY-Direktive	69
5.6	PDO-Direktive	70
5.6.1	ON-HOME-Scheduling	70
5.6.2	Scheduling über DISTRIBUTE-ONCE	74
6	Implementierung	77
6.1	Integration in einen Präprozessor für SVM-Fortran	77
6.2	Beschreibung der Implementierung	80
6.2.1	Format der Datei <i>svm_workshar.dat</i>	80
6.2.2	Speicherplatzbedarf der internen Tabellen	84
6.3	Ergebnisse und Messungen anhand einiger Testbeispiele	84
6.3.1	Der Parallelrechner Intel Paragon XP/S 10 mit ASVM	84
6.3.2	Zeitaufwand der einzelnen Direktiven	85
6.3.3	Vergleich von direktem mit vordefiniertem Scheduling	86

7 Zusammenfassung und Ausblick	90
Literaturverzeichnis	93
A Syntax von SVM-Fortran	98

Kapitel 1

Einleitung

In den letzten Jahren gewinnen Parallelrechner aufgrund des großen Rechenbedarfs mehr und mehr an Bedeutung. Ein Architekturkonzept, das besondere Aufmerksamkeit erregt, ist das der massiv-parallelen Rechner. Diese Systeme zeichnen sich durch eine große Anzahl von Prozessoren aus, die über ein Verbindungsnetzwerk miteinander kommunizieren, und über keinen gemeinsamen, sondern einen verteilten Speicher verfügen.

Die Programmierung der massiv-parallelen Systeme kann schwierig und aufwendig sein, da der Programmierer die Verteilung der Daten auf die lokalen Speicher spezifizieren muß. Außerdem muß er bei nichtlokalen Datenzugriffen eines Prozessors die Routinen zur Kommunikation sowohl für das Senden als auch für das Empfangen der Daten explizit in das Programm einfügen.

Um diese Schwierigkeiten bei der Programmierung der massiv-parallelen Rechner zu überwinden, werden zur Zeit zwei unterschiedliche Ansätze in der Fachwelt diskutiert. Der eine Ansatz basiert auf der Datenverteilung, bei dem der Compiler die Datenverwaltung übernimmt. Der Benutzer spezifiziert für Programmdaten Verteilungsstrategien, die den Compiler veranlassen, die Daten auf die lokalen Speicher der Prozessoren zu verteilen. Der Compiler generiert dann im Programm bei einem nichtlokalen Datenzugriff die entsprechenden Kommunikationsschemata. Dadurch wird dem Benutzer die Sicht eines globalen Adreßraumes gegeben, der durch den Compiler auf die verteilten Speicher abgebildet wird. Die Verteilung der Iterationen einer parallel auszuführenden Schleife auf die Prozessoren wird dann durch die "*owner-compute-rule*" bestimmt, d.h. der Prozessor führt nach Möglichkeit die Iterationen aus, in denen auf Daten in seinem lokalen Speicher zugegriffen wird, wodurch nichtlokale Datenzugriffe reduziert werden.

Compiler für Sprachen, die auf dem Prinzip der Datenverteilung aufbauen wie High Performance Fortran (HPF) [29, 30], sind komplex, da die Abstraktionsebene, die dem Compiler zugrunde liegt, sehr niedrig ist. Compiler, basierend auf Datenverteilungen, sind deshalb im wesentlichen auf reguläre Datenverteilungen und Datenzugriffe beschränkt.

Ein anderer Ansatz ist der des virtuell gemeinsamen Speichers (*Shared Virtual Memory*, SVM). Dieses Konzept wurde 1986 von K. Li [43] vorgestellt. Es simuliert durch geeignete Algorithmen, entweder in Hardware oder Software implementiert, einen virtuell gemeinsamen Adreßraum auf massiv-parallelen Systemen mit real verteiltem Speicher. Die Zugriffe auf Daten erfolgen bei einem Rechner mit einem SVM-System wie bei einem Rechner mit einem gemeinsamen Speicher. Daher müssen Compiler von Sprachen für SVM-Systeme die Datenverwaltung nicht explizit durchführen, sondern können auf der höheren Abstraktionsebene des gemeinsamen Speichers aufsetzen.

Obwohl die Datenverwaltung vom Betriebssystem oder von der Hardware übernommen wird, darf der Benutzer die Datenverteilung nicht außer acht lassen. Da eine Lokalität beim Datenzugriff in einem SVM-System für die Ausführungszeit essentiell ist, stellt die Sprache SVM-Fortran dem Benutzer Sprachmittel zur Verfügung, mit denen er Datenlokalität erreichen kann. Die Sprache SVM-Fortran [11, 12] wurde zur Programmierung von SVM-Systemen am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich entwickelt. Eine Möglichkeit zur Steuerung der Lokalität von Daten ist das Scheduling. Im Rahmen dieser Arbeit wird **Scheduling** als die Zuweisung von Befehlssequenzen zu Prozessoren definiert. Dem Benutzer werden in SVM-Fortran verschiedene Sprachmittel angeboten, mit denen er den Prozessoren Befehlssequenzen zuweisen kann, so daß die Prozessoren hauptsächlich auf Daten in ihrem lokalen Speicher zugreifen.

In dieser Arbeit wird das benutzergesteuerte Scheduling von Schleifen vorgestellt, mit dessen Hilfe eine Zuordnung der Schleifeniterationen zu den Prozessoren abgespeichert werden kann. Diese Zuordnung kann bei nachfolgenden Schleifen, die auf dieselben Daten zugreifen, auch wenn die Schleifen sich in unterschiedlichen Unterprogrammen befinden, wiederverwendet werden, wodurch eine Lokalität der Daten über Unterprogramm Grenzen hinweg erhalten werden kann.

Nachdem in Kapitel 2 die wesentlichen Merkmale von SVM-Systemen und SVM-Fortran vorgestellt werden, wird in Kapitel 3 ein Vergleich bekannter Fortran-Dialekte zur Programmierung massiv-paralleler Systeme vorgenommen. Anschließend wird in Kapitel 4 das benutzergesteuerte Scheduling in SVM-Fortran detailliert diskutiert und in Kapitel 5 die Realisierung der zum benutzergesteuerten Schleifen-Scheduling mit Hilfe von Templates benötigten Konstrukte erläutert. In Kapitel 6 wird auf einige Details der Implementierung eingegangen, und es werden Performance-Ergebnisse vorgestellt. Schließlich werden in Kapitel 7 die Ergebnisse dieser Arbeit zusammengefaßt.

Kapitel 2

Programmierung von SVM-Systemen

In diesem Kapitel wird eine kurze Übersicht über die Gebiete gegeben, die für die Programmierung von SVM-Systemen wichtig sind. Die Schwerpunkte liegen auf der Erläuterung des Konzepts des virtuell gemeinsamen Speichers (*shared virtual memory*, SVM¹) und einer Zusammenfassung der wesentlichen Eigenschaften der auf die besonderen Erfordernisse von SVM-Systemen zugeschnittenen Sprache SVM-Fortran.

2.1 Virtuell gemeinsamer Speicher

Das Konzept des virtuell gemeinsamen Speichers [43] realisiert auf Rechnern mit verteiltem Speicher die Sicht eines gemeinsamen Speichers mit Hilfe eines virtuellen Adreßraumes, dargestellt in Abbildung 2.1. Falls ein Prozessor in einem SVM-System auf Daten zugreifen will, die sich nicht in seinem lokalen Speicher befinden, dann sorgt das SVM-System schließlich dafür, daß sich die benötigten Daten im lokalen Speicher des Prozessors befinden und er dann darauf zugreifen kann.

Daten werden in SVM-Systemen zu Seiten (*pages*) zusammengefaßt. Dadurch müssen administrative Informationen, wie beispielsweise welcher Prozessor eine Kopie einer Seite besitzt, nur pro Seite gespeichert werden und nicht für jedes einzelne Datum, wodurch der Speicher-*Overhead* reduziert werden kann. Seiten des SVM-Systems werden auf die einzelnen lokalen Prozessorspeicher verteilt. Bei einem nicht-lokalen Datenzugriff wird ein Seitenfehler (*page fault*) ausgelöst und die gesamte Seite mit dem entsprechenden Datum an den anfragenden Prozessor gesendet, was transparent für das Benutzerprogramm geschieht.

¹Die Namensgebung in der Literatur ist nicht einheitlich. Häufig werden Begriffe wie *virtual shared memory*, *distributed shared memory* und andere Synonyme verwendet.

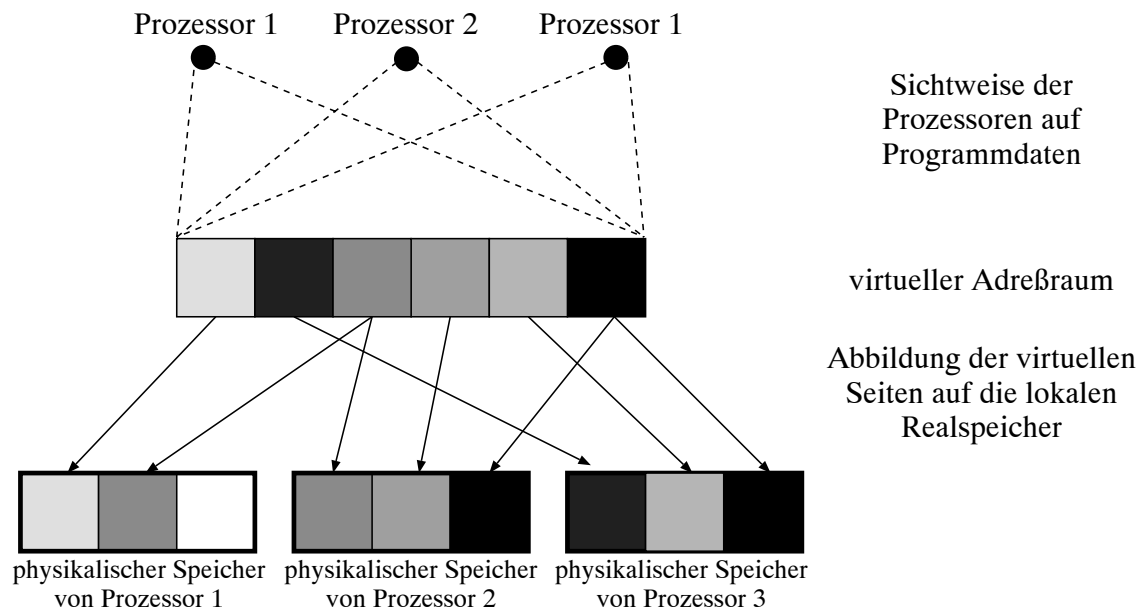


Abbildung 2.1: Sichtweise auf Programmdaten und ihre Speicherung

Beim seitenorientierten Zugriff wird zu jeder Seite das Zugriffsrecht für diese Seite, der Status, festgelegt. Eine Seite kann den Read-Only- oder Read-Write-Status haben. Bei einer Seite mit dem Read-Only-Status darf ein Prozessor nur lesend auf diese Seite zugreifen. Der Read-Write-Seitenstatus legt fest, daß der Prozessor sowohl lesend als auch schreibend auf diese Seite zugreifen darf. Bei einem nichtlokalen Datenzugriff kann unterschieden werden zwischen einem *read page fault* oder *write page fault*. Im ersten Fall erhält der Prozessor die angeforderte Seite mit dem Read-Only-Status, im zweiten Fall hingegen mit dem Read-Write-Status.

Ob von einer Seite mehrere Kopien existieren dürfen, hängt von der Kohärenzstrategie und dem Seitenstatus ab.

2.1.1 Kohärenzstrategien

Kohärenzstrategien garantieren, daß alle Prozessoren alle Schreiboperationen auf eine Seite in einer bestimmten logischen Reihenfolge durchführen. Die beiden bekanntesten Kohärenzstrategien sind:

1. strikte Kohärenz [21, 22]:

Diese Strategie verlangt, daß ein gelesenes Datum mit dem zuletzt geschriebenen Datum übereinstimmen muß, unabhängig davon, welcher Prozessor dieses Datum geschrieben hat. Um dies zu gewährleisten, existieren verschiedene Lösungsansätze. Als zwei wichtige seien hier die Invalidierungstechnik und die Write-Update-Technik genannt. In einem kurzen Überblick werden diese zwei Techniken vorgestellt:

- **Invalidierungstechnik** [36]:

Es können mehrere Kopien einer Seite mit Read-Only-Status existieren, eine Seite mit Read-Write-Status hingegen darf nicht kopiert werden, diese Seite existiert nur einmal im System. Ein nichtlokaler Datenzugriff ist entweder ein *read* oder *write page fault*. Bei einem *read page fault* wird eine Kopie einer Seite angefordert. Bei einem *write page fault* hingegen erhält der anfordernde Prozessor eine gültige Kopie der Seite mit Read-Write-Status, und alle Kopien der Seite werden invalidiert.

- **Write-Update-Technik** [57]:

Auch bei der Write-Update-Technik können mehrere Kopien einer Seite mit Read-Only-Status existieren, aber nur ein einziger Prozessor führt für diese Seite den Read-Write-Status. Falls der Prozessor, der für eine Seite den Read-Write-Status hat, schreiben will, so wird bei den Kopien der Seite das Leserecht vorübergehend entzogen. Anschließend schreibt der Prozessor auf die Seite, verschickt danach die Änderungen an die Kopien dieser Seite, und die Kopien erhalten wieder das Leserecht auf diese Seite. Bei einem nichtlokalen Datenzugriff fordert der Prozessor die Seite mit dem entsprechenden Status an.

2. schwache Kohärenz [26]:

Bei dieser Strategie muß ein gelesenes Datum nicht mit dem zuletzt geschriebenen Datum übereinstimmen. Daher ist es hier erlaubt, daß mehrere Kopien einer Seite mit Read-Write-Status existieren. An Synchronisationspunkten findet eine Aktualisierung aller Seiten mit schwacher Kohärenz statt, dabei werden alle modifizierten Kopien einer Seite vereinigt. Der Vorteil dieser Strategie ist, daß mehrere Prozessoren gleichzeitig auf dieselbe Seite schreiben können, ohne daß *page faults* entstehen.

Prinzipiell ist es möglich, in einem Programmlauf die Kohärenzstrategie für eine oder mehrere Seiten zu ändern. Ein typischer Ablauf wäre, für alle Seiten die strikte Kohärenz zu fordern. Bei einem konkurrierenden Zugriff mehrerer Prozessoren auf eine Seite in einem bestimmten Programmteil kann die Kohärenzstrategie für diese Seite dann in die schwache Kohärenz geändert werden. Am Ende dieses Programmteils wird ein Synchronisationspunkt eingefügt, wodurch eine einheitliche Sicht aller Prozessoren auf diese Seite hergestellt wird, und die strikte Kohärenz für diese Seite gefordert.

2.1.2 Realisierung von SVM-Systemen auf Rechnern

Die Realisierung eines SVM-Systems auf einem Rechner hängt von vielen Einflußgrößen ab, wie z.B. dem Verbindungsnetzwerk zwischen den Prozessoren, dem Seitenersetzungsmechanismus, der Speicherallokationsstrategie, der Seitengröße oder

der Managementalgorithmen zur Seitenverwaltung [47]. Die letztgenannte Einflußgröße wird einer genaueren Betrachtung unterzogen, da sie die Ausführungszeiten bei *page faults* entscheidend beeinflußt.

Realisierung der Managementalgorithmen

Managementalgorithmen in SVM-Systemen dienen der Verwaltung von Seiten inkl. der Einhaltung der Kohärenzstrategien. Managementalgorithmen können sowohl *software*- als auch *hardware*-basiert implementiert werden [44]. Der Vorteil der *software*-basierten Ansätze ist, daß sie flexibler sind, da aufwendige Protokolle zur Verwaltung der Seiten möglich sind. Demgegenüber sind die *hardware*-basierten Ansätze i. allg. effizienter realisierbar.

Der *software*-basierte Ansatz wird aufgrund der höheren Flexibilität in vielen SVM-Systemen zur Implementierung der Kohärenzstrategien gewählt. In diesem Ansatz unterscheidet man zwischen dem Besitzer, dem Teilhaber und dem Manager einer Seite. Jede Seite wird einem Prozessor, dem Manager dieser Seite, zugeordnet. Dieser Prozessor hat alle notwendigen Informationen zur Administration der Seite, wer z.B. Besitzer ist oder welcher Prozessor Kopien der Seite besitzt. Der Besitzer einer Seite ist der Prozessor, der das Read-Write-Recht dieser Seite besitzt. Dieser Besitzer hat unter anderem Informationen über die Teilhaber der Seite, die Teilhaber haben eine Kopie dieser Seite mit Read-Only-Status. Für die Zuordnung des Managers zu einem Prozessor sind in der Literatur drei Algorithmen [43] bekannt:

- **zentraler Manageralgorithmus:**

Ein Prozessor ist Manager aller Seiten. Alle Anfragen bei einem *page fault* werden an ihn gerichtet. Der Manager leitet die Anfrage an den Besitzer weiter.

- **statisch verteilter Manageralgorithmus:**

Jeder Prozessor im SVM-System verwaltet eine vorbestimmte Menge an Seiten, er ist der Manager dieser Seiten. Durch einen einfachen Algorithmus (z.B. eine Hash-Funktion) kann jeder Prozessor berechnen, wer der Manager einer Seite ist. Eine *page fault* Anfrage wird dann an den Manager verschickt, der die Anfrage an den Besitzer weiterleitet.

- **dynamisch verteilter Manageralgorithmus:**

Bei diesem Algorithmus findet keine Unterscheidung zwischen Manager und Besitzer statt. Jeder Prozessor hat eine lokale Tabelle, in der der wahrscheinliche Besitzer jeder Seite spezifiziert ist. Bei einem *page fault* richtet ein Prozessor seine Anfrage an den in dieser Tabelle befindlichen wahrscheinlichen Besitzer der Seite. Besitzt dieser Prozessor die Seite nicht, so leitet er die Anfrage an den Prozessor weiter, der als der wahrscheinliche Besitzer in seiner lokalen Tabelle angegeben ist. Dies ist der Anfang einer Folge von Anfragen, die schließlich zum tatsächlichen Besitzer führt. In [43] ist ein Beweis der Korrektheit dieser Strategie angegeben. Die lokalen Tabellen werden aktualisiert,

wenn ein Besitzer einen *write page fault* erhält oder die Teilhaber einer Seite eine Invalidierungsanfrage erhalten.

Der zentrale Manageralgorithmus ist zwar einfach zu realisieren, aber er stellt immer einen potentiellen Engpaß für das System dar, wenn viele *page faults* zur gleichen Zeit auftreten. In diesem Punkt sind die verteilten Manageralgorithmen diesem Algorithmus überlegen.

Beim statisch verteilten Manageralgorithmus müssen bei jeder Anfrage zwei Nachrichten verschickt werden, zum einen die von dem den *page fault* auslösenden Prozessor an den Manager und zum anderen die vom Manager an den Besitzer. Beim dynamisch verteilten Manager hingegen ist es möglich, daß mit dem Verschicken einer Nachricht schon der richtige Besitzer der Seite gefunden wird, wenn die Information über den wahrscheinlichen Besitzer mit der Realität übereinstimmt. Messungen haben ergeben [44], daß die Realisierung der Kohärenzstrategien mittels des dynamisch verteilten Manageralgorithmus der Realisierung mittels des statisch verteilten Manageralgorithmus in der Regel überlegen ist.

Beispiele für *software*-basierte Ansätze sind MaX [28, 52], MYOAN [14] und ASVM [59]. Ein Beispiel für einen *hardware*-basierten Ansatz ist der Rechner KSR1 der Firma Kendall Square Research [38].

2.2 Benutzergesteuertes Scheduling

Benutzergesteuertes Scheduling spielt in SVM-Fortran (siehe Abschnitt 2.3) eine wesentliche Rolle. Unter benutzergesteuertem Scheduling versteht man, daß der Benutzer festlegt, welche Ausführungsblöcke in einem Programm den Prozessoren, die das Programm ausführen, zugewiesen werden. Ein **Ausführungsblock** ist eine Sequenz von Befehlen. Beispiele für Ausführungsblöcke sind eine Schleife, die Iteration einer Schleife, die Anweisungen bei einer IF-Abfrage, ein Unterprogramm oder Wertzuweisungen.

Beim Scheduling durch einen Compiler wird die Zuordnung der Ausführungsblöcke auf die Prozessoren oft durch Anwendung einer vorgegebenen Strategie, wie z.B. einer statischen Blockverteilung zur Verteilung der Iterationen einer Schleife auf die Prozessoren, gelöst. Dies kann Probleme verursachen, wenn die Verteilung dieser Blöcke nicht auf das Programm und dessen Daten zugeschnitten ist. Es kann unter Umständen zu einer vermehrten Kommunikation zwischen den Prozessoren kommen und somit die erwartete Ausführungszeit ungünstig beeinflussen. In der Forschung existieren daher Bestrebungen, die Verteilungsstrategie der Ausführungsblöcke durch Analysewerkzeuge bestimmen zu lassen [27, 34, 45, 51], um so eine problemgerechte Verteilung zu erhalten. In der Praxis existiert bis heute jedoch kein automatisches Werkzeug, das dieses Problem zufriedenstellend löst.

Eine Alternative dazu ist das benutzergesteuerte Scheduling. Dem Benutzer werden verschiedene Verteilungsverfahren für die Zuordnung der Ausführungsblöcke auf die

Prozessoren zur Verfügung gestellt. Für jeden Block muß er dann entscheiden, welches Verfahren er wählt. Dies bedeutet einen größeren Aufwand für den Programmierer. Ein Vorteil besteht darin, daß der Programmierer oft eine globale Sicht des Programms und seiner Daten hat und aufgrund dessen Entscheidungen treffen kann, die dann zu einer effizienteren Programmausführung führen können. Der Compiler hingegen trifft diese Entscheidungen i. allg. anhand von lokalen Informationen und kann daher unter Umständen zu einer ineffizienteren Lösung gelangen.

SVM-Fortran bietet sowohl das automatische als auch das benutzergesteuerte Scheduling an. Dazu werden dem Benutzer Sprachmittel zur Verfügung gestellt, mit denen er die Zuweisung von Ausführungsblöcken auf Prozessoren vornehmen kann.

2.3 SVM-Fortran

SVM-Fortran [11, 12] ist eine Erweiterung von Fortran77 [2], die die Programmierung paralleler Programme erlaubt. Die Zielmaschinen dieser Sprache sind Parallelrechner mit virtuell gemeinsamem Speicher. Das Datenmodell, auf dem die Erweiterung basiert, ist das eines virtuell gemeinsamen Speichers, d.h. der Benutzer hat die Sicht eines einzigen globalen Adreßraumes, obwohl die Daten auf den lokalen Speichern der Prozessoren abgespeichert werden. Das zugrundeliegende Programmiermodell ist das *Single-Program-Multiple-Data*-Modell (SPMD-Modell), das 1985 zum ersten Mal in [19] vorgestellt wurde. Dieses Modell legt fest, daß auf allen Prozessoren dasselbe Programm ausgeführt wird. Dennoch kann der Kontrollfluß auf den Prozessoren unterschiedlich sein, wenn bei einem Programmteil angegeben wird, daß dieser von einem bestimmten Prozessor ausgeführt werden soll.

Im folgenden soll ein kurzer Überblick über die Sprache gegeben werden. Detaillierte Informationen sind in [11, 12] zu finden.

Prozessorfelder

In SVM-Fortran kann der Benutzer bestimmen, auf welchen Prozessoren ein Ausführungsblock, wie z.B. die Iterationen einer parallel auszuführenden Schleife, ausgeführt werden soll. Dies kann er durch Definition von **Prozessorfeldern** (*processor arrangements*) erreichen, die als Hilfsmittel dienen, um auf abstrakte Prozessoren zugreifen und ihnen Ausführungsblöcke zuweisen zu können. Zwischen den abstrakten Prozessoren in SVM-Fortran und den realen Prozessoren des Parallelrechners wird wiederum eine Abbildung hergestellt, so daß jedem abstrakten genau ein realer Prozessor entspricht und jedem realen Prozessor höchstens ein abstrakter zugeordnet wird.

In dem Beispiel in Abbildung 2.2 wird das Prozessorfeld P mit vier Elementen auf die abstrakten Prozessoren, die immer von null beginnend aufwärts durchnumeriert sind, abgebildet. Das Prozessorfeldelement P(1,1) wird beispielsweise auf den abstrakten Prozessor 0 abgebildet. Die abstrakten Prozessoren wiederum werden auf

CSVM\$ PROCESSORS:: P(2,2)

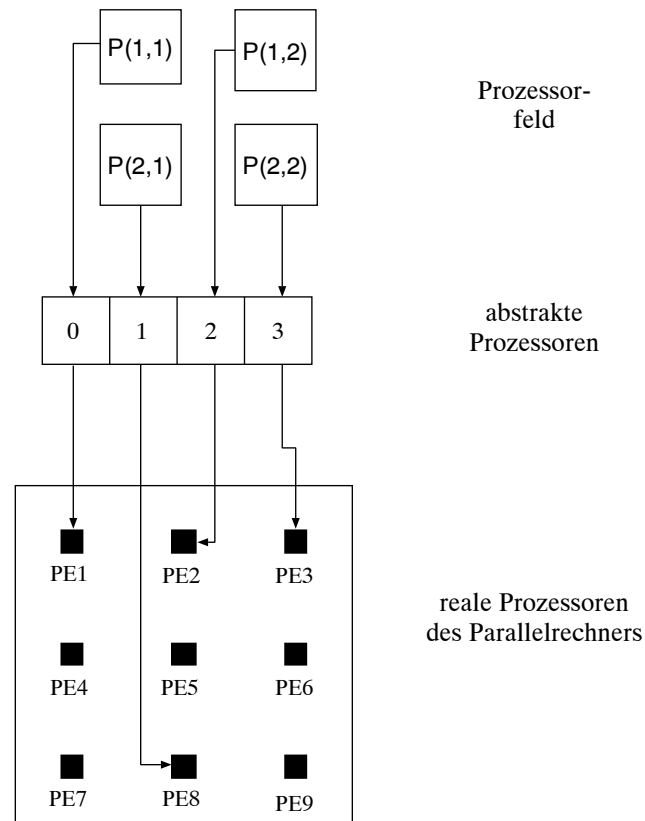


Abbildung 2.2: Zusammenhang zwischen Prozessorfeld, abstrakten und realen Prozessoren anhand eines Beispiels

die realen Prozessoren des Parallelrechners $PE = \{PE1, \dots, PE9\}$ ² abgebildet, z.B. wird der abstrakte Prozessor 0 auf den realen Prozessor PE1 abgebildet. Wird nun dem Prozessorfeldelement $P(1,1)$ ein Ausführungsblock zugewiesen, so wird dieser dann auf dem realen Prozessor PE1 ausgeführt.

Abstrakte Prozessoren werden zu **processor sets** zusammengefaßt. Wenn ein Ausführungsblock von einem *processor set*, das unter Umständen auch nur aus einem abstrakten Prozessor bestehen kann, ausgeführt wird, dann wird dies als das **active processor set (APS)** dieses Ausführungsblockes bezeichnet. Während der Programmausführung kann es mehrere *active processor sets* geben, falls mehrere Ausführungsblöcke parallel bearbeitet werden. Zu jedem dieser Blöcke existiert dann ein APS, wobei die Schnittmenge dieser *active processor sets* leer sein muß.

Alle Prozessoren, die einer Anwendung zugeordnet werden, bilden das **initial processor set**. Dies ist das APS bei Programmstart.

²Im folgenden werden reale Prozessoren mit PE bezeichnet.

Synchronisation

Innerhalb eines APS können die Prozessoren durch verschiedene Mechanismen synchronisiert werden, wie z.B. durch einen *Barrier*, eine kritische Sektion oder eine Lock-Synchronisation. Der *Barrier* synchronisiert alle Prozessoren im APS. Die Anweisungen in einer kritischen Sektion werden zur selben Zeit maximal von einem Prozessor abgearbeitet. Bei einer Lock-Synchronisation können die Anweisungen nur durchgeführt werden, wenn ein Prozessor exklusiven Schreib-Lese-Zugriff auf eine Speicherregion erhalten hat. Diesen exklusiven Zugriff behält der Prozessor so lange, bis er auf eine Unlock-Operation trifft.

Ausführungsmodus

Der Benutzer kann durch den Ausführungsmodus bestimmen, ob ein Ausführungsblock von nur einem Prozessor im APS des Ausführungsblockes oder von allen Prozessoren dieses APS ausgeführt werden soll. Es existieren zwei Modi:

- **exklusiv:**
Der Ausführungsblock, der als exklusiv gekennzeichnet ist, wird nur von einem einzigen Prozessor ausgeführt. Die anderen Prozessoren im APS des Ausführungsblockes überspringen diesen Block.
- **repliziert:**
Alle Prozessoren, die auf einen als repliziert markierten Ausführungsblock treffen, bearbeiten diesen Block redundant. Der Kontrollfluß der beteiligten Prozessoren kann jedoch verschieden sein.

Im folgenden Beispiel wird der Unterprogrammaufruf in der REPLICATED_REGION von jedem Prozessor redundant ausgeführt, die I-Schleife in der EXCLUSIVE_REGION hingegen wird nur von einem einzigen Prozessor ausgeführt.

```
PROGRAM EXAMPLE_1
INTEGER I

C *** Beginn des replizierten Ausführungsblockes
CSVM$ REPLICATED_REGION

C *** Aufruf des Unterprogramms SORT
CALL SORT()

CSVM$ REPLICATED_REGION_END

C *** Beginn des als exklusiv markierten Ausführungsblockes
CSVM$ EXCLUSIVE_REGION
```

```

        DO I = 1, 10
            ....
        ENDDO

CSVM$ EXCLUSIVE_REGION_END

END

```

Manchmal kann es sinnvoll sein, daß ein Unterprogramm in einer REPLICATED_REGION nicht redundant auf jedem Prozessor, sondern auf kooperative Weise ausgeführt wird: Falls sich im Unterprogramm mehrere Ausführungsblöcke befinden, die parallel ausgeführt werden können, dann werden diese auch parallel und nicht redundant ausgeführt. Dazu ist in SVM-Fortran die COORDINATED_CALL-Direktive vorgesehen.

Mit der Angabe des Ausführungsmodus ist die parallele Ausführung disjunkter Teile eines Ausführungsblockes nicht möglich. Dies kann nur mit Hilfe der parallelen Sektionen (*parallel sections*) oder parallelen Schleifen (*parallel loops*) erreicht werden. Dies sind die zwei einzigen Konstrukte, mit denen man Parallelismus bei der Ausführung eines Programms spezifizieren kann. Sie können in SVM-Fortran beliebig geschachtelt auftreten.

Parallele Sektionen

In parallelen Sektionen können Ausführungsblöcke, die einer Sektion zugeordnet werden, von mehreren Prozessoren parallel bearbeitet werden:

```

PROGRAM EXAMPLE_2
PARAMETER (N=1000000)
INTEGER SPROD, SUM(N), X(N), Y(N)

C *** Initialisiere die Vektoren X und Y
CALL INIT(X,Y,N)

C *** Fuehre zwei Teile des Programms parallel aus
CSVM$ PSECTION

C *** Erster Teil des parallel auszufuehrenden Programms
CSVM$ SECTION
C *** Addiere die Vektoren X und Y
DO 100 J=1,N
    SUM(J)=X(J) + Y(J)
100 CONTINUE

C *** Zweiter Teil des parallel auszufuehrenden Programms
CSVM$ SECTION
C *** Berechne das Skalarprodukt
SPROD=0

```

```

        DO 200 I=1,N
            SPROD=SPROD + X(I) * Y(I)
200    CONTINUE

CSVM$ PSECTION_END

        END

```

In diesem Beispiel wird mit der PSECTION-Direktive die Vektoraddition von der einen Hälfte der Prozessoren, die auf dieses Konstrukt treffen, ausgeführt und das Skalarprodukt von der anderen Hälfte. Die Zuordnung der beiden parallelen Sektionen zu den Prozessoren wird vom Compiler implizit durchgeführt, falls keine explizite Zuordnung durch den Benutzer erfolgt.

Parallele Schleifen

In einer parallelen Schleife können die Iterationen auf Prozessoren verteilt werden. Die Zuweisung der Iterationen einer Schleife auf die Prozessoren wird im folgenden als **Schleifen-Scheduling** bezeichnet.

```

PROGRAM EXAMPLE_3
PARAMETER (N=1024)
INTEGER I, J, SUM(N,N), DIFF(N,N), A(N,N), B(N,N)

C *** Allokiere die Variable I im privaten Speicher eines jeden Prozessors
CSVM$ PRIVATE:: I

C *** Initialisiere die Matrizen A und B
CALL INIT(A,B,N)

C *** Führe die I-Schleife parallel aus
CSVM$ PDO(LOOPS(I))
DO 10 I=1,N

    DO 20 J=1,N

C ***      Berechne die Summe und Differenz der Matrizen A und B
        SUM(I,J) = A(I,J) + B(I,J)
        DIFF(I,J)= A(I,J) - B(I,J)

20    CONTINUE

10    CONTINUE

END

```

In diesem Beispiel werden die Iterationen der äußeren Schleife auf alle Prozessoren, die auf die PDO-Direktive treffen, aufgeteilt. Diese Prozessoren führen dann den Schleifenkörper, die innere J-Schleife, lokal sequentiell aus.

Das Schleifen-Scheduling kann aber auch mit Hilfe von **Templates** durchgeführt werden. Das Konzept der Templates ist aus HPF [29, 30] übernommen worden. Es wird in HPF zur Speicherung einer Verteilung der Daten auf die Prozessoren verwendet, wohingegen es in SVM-Fortran nur zum Schleifen-Scheduling verwendet wird.

In einem Template wird die Zuordnung der Template-Elemente zu einem Prozessor festgehalten. Bei der Schleifenausführung kann man diese Informationen nutzen und die Ausführung der Schleife gemäß einem Template durchführen³. Dies bedeutet, daß eine Schleifeniteration auf dem Prozessor ausgeführt wird, dem das entsprechende Template-Element zugeordnet wurde. Diese Art des Schleifen-Scheduling, in SVM-Fortran auch vordefiniertes Scheduling genannt, wird in Kapitel 4 detailliert erklärt.

Bei der PDO-Direktive hat der Benutzer die Möglichkeit, Variablen, die im Schleifenkörper auftreten, als Reduktionsvariablen zu definieren. Die unterstützten Reduktionsoperationen sind die Addition und Multiplikation sowie die Minimum- bzw. Maximumbildung.

```

Program EXAMPLE_4
PARAMETER (N=1024)
INTEGER I, SUM, MINIMUM, A(N), B(N)

C *** Allokiere die Variablen im privaten Speicher eines jeden Prozessors
CSVM$ PRIVATE:: I,J

C *** Initialisiere die Variablen SUM und MINIMUM und die Felder A und B
CALL INIT(A,B,N,SUM,MINIMUM)

C *** Fuehre die I-Schleife parallel mit den Reduktionsvariablen
C *** SUM und MINIMUM aus
CSVM$ PDO(LOOPS(I), REDUCTION(SUM, MINIMUM)
DO 10 I=1,N

C ***      Berechne die Summe aller Feldelemente von A und das Minimum
C ***      des Feldes B
SUM = SUM + A(I)
MINIMUM = MIN(MINIMUM, B(I))

10 CONTINUE

END

```

In der parallelen I-Schleife wird in der Variablen SUM die Summe des Feldes A berechnet und in MINIMUM das Minimum des Feldes B bestimmt. Eine Implementierungsstrategie der Reduktion wäre hierbei, daß jeder Prozessor, der einen Teil der parallelen Schleife ausführt, während der Schleifenausführung in seinem lokalen Speicher eine Zwischensumme bzw. das lokale Minimum der Feldelemente berechnet,

³In der Literatur wird dies als *ON-HOME*-Scheduling bezeichnet.

die der Prozessor überprüft hat. Nach der parallelen Schleife bilden dann die Prozessoren die Endsumme des Feldes A bzw. das Minimum des Feldes B durch geeignete Kommunikationsschemata, die vom Compiler automatisch generiert werden.

Datenspeicherung

In SVM-Fortran kann der Benutzer bestimmen, ob Daten im gemeinsamen Speicher oder im lokalen Speicher eines Prozessors abgelegt werden. Im Deklarationsteil einer Programmeinheit gibt der Benutzer an, wo diese Daten gespeichert werden sollen. Es bestehen zwei Möglichkeiten, Variablen im Speicher abzulegen:

- **private:**

Die als *private* deklarierten Daten werden im lokalen Speicher des Prozessors abgelegt. Nur der Prozessor, bei dem die Daten gespeichert werden, hat Zugriff darauf.

- **shared:**

Die als *shared* markierten Daten werden im gemeinsamen Speicher abgelegt. Alle Prozessoren, die sich im APS des Unterprogramms befinden, in dem die Daten deklariert werden, haben Lese- und Schreibzugriff auf diese Daten. Die restlichen Prozessoren können auf diese Daten nicht zugreifen.

In einem Beispiel soll der Unterschied zwischen *private* und *shared* Variablen verdeutlicht werden:

```

PROGRAM EXAMPLE_VARIABLES
PARAMETER (N=1000000)
C *** Die Default-Speichermethode ist shared.
INTEGER A(N), B(N), I

C *** Variable I wird im lokalen Speicher gespeichert
CSVM$ PRIVATE:: I

....

C *** Feld B wird nach Feld A kopiert; dabei fuehrt jeder Prozessor nur
C *** einen Teil des Kopiervorganges auf dem Feld im gemeinsamen Speicher durch
CSVM$ PDO(LOOPS(I), STRATEGY(BLOCK))
DO I = 1, N
    A(I) = B(I)
ENDDO

....

END
```

In Programm EXAMPLE_VARIABLES wird die Variable I im lokalen Speicher eines jeden Prozessors angelegt, da bei der PDO-Schleife die Variable auf jedem Prozessor unterschiedliche Werte hat. In der Schleife führt jeder Prozessor den Kopiervorgang auf disjunkten Bereichen des Feldes A aus, aber die Änderungen sind für die anderen Prozessoren auch sichtbar, da das Feld A im gemeinsamen Speicher allokiert ist.

2.4 Einordnung der Diplomarbeit in das SVM-Fortran-Projekt

Das SVM-Fortran-Projekt setzt sich aus mehreren Teilprojekten zusammen, die in Abbildung 2.3 skizziert sind.

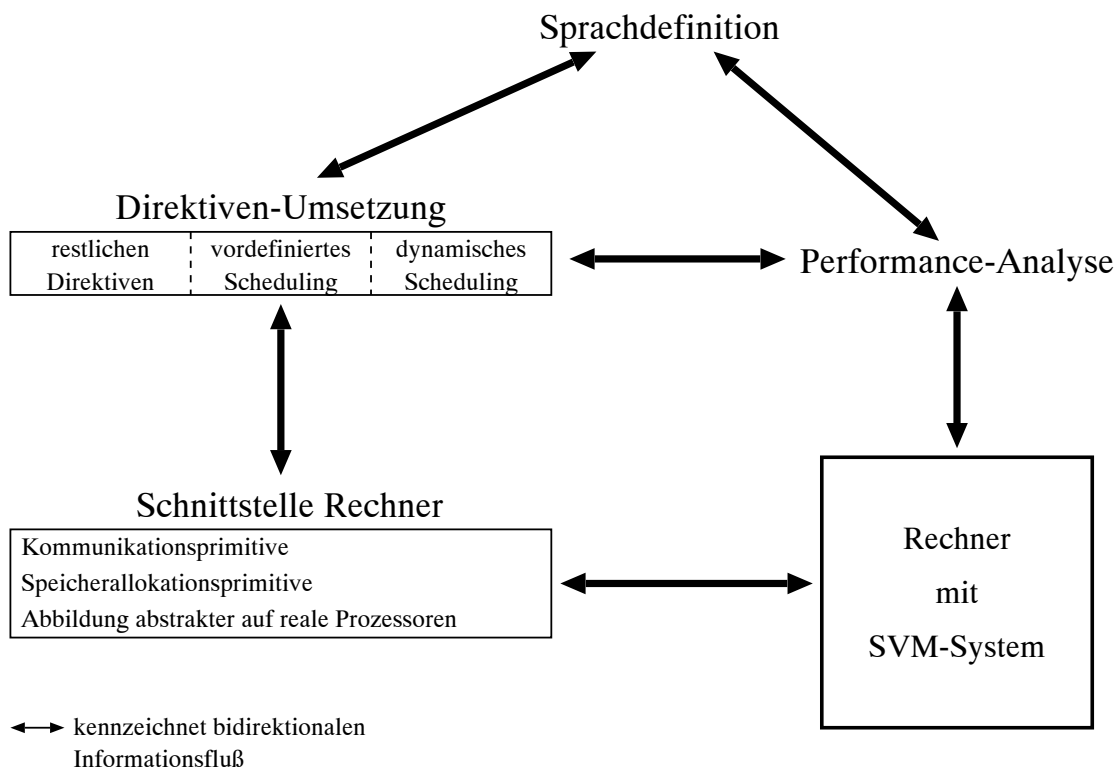


Abbildung 2.3: Das SVM-Fortran-Projekt

Nach der Sprachdefinitionsphase wurde die Implementierung der Sprache in drei Teilprojekte aufgeteilt. In einem Teilprojekt, das mit Hilfe dieser Diplomarbeit durchgeführt wurde, sollten die Prozessorfelder, Templates und das vordefinierte Scheduling implementiert werden. Dazu war es notwendig, eine geeignete und effiziente interne Repräsentation für Prozessorfelder und Templates zu finden, diese dann zu implementieren und in den existierenden Präprozessor für SVM-Fortran zu integrieren. Weiterhin sollte für die statische Verteilung der Templates auf die

Prozessorfelder, in SVM-Fortran auch *standard distribution* genannt, eine Repräsentation mit geringem Speicherplatzbedarf gefunden und implementiert werden. Für diese Verteilungen sollte auch das *ON-HOME*-Scheduling bei den parallelen Schleifen realisiert werden. Aufbauend auf diesem Teilprojekt werden in einem anderen Teilprojekt [50] die unregelmäßigen Verteilungsstrategien der Templates auf Prozessorfelder, ihre interne Darstellung sowie das dynamische Schleifen-Scheduling und die Abspeicherung dieser Verteilung im Template umgesetzt. In dem dritten Teilprojekt wurden die restlichen Direktiven implementiert und die Schnittstelle zum Rechner bereitgestellt. Diese Schnittstelle beinhaltet alle Kommunikations- und Speicherallokationsprimitive sowie die Abbildung der abstrakten auf die realen Prozessoren des Rechners. Da alle Teilprojekte eng miteinander verflochten sind und Informationen von einem Bereich in den anderen zur Übersetzungs- und Laufzeit übermittelt werden müssen, war eine präzise Definition der Schnittstellen nötig.

In einem weiteren Teilprojekt soll ein Werkzeug zur Performance-Analyse realisiert werden. Dieses Projekt gliedert sich wiederum in zwei Unterprojekte. In einem Projekt sollen die Daten, die zu einer Performance-Analyse wichtig sind, bei der Programmausführung aufgezeichnet werden. In dem anderen Projekt werden diese Daten dann durch geeignete Werkzeuge gefiltert und für den Benutzer aufbereitet, damit er Performance-Engpässe schnell identifizieren und entsprechend korrigieren kann.

Kapitel 3

Datenverteilung und Scheduling in Fortran-Dialekten

Da SVM-Fortran auf anderen Fortran-Dialekten aufbaut und einige Konzepte, die sich in anderen Dialekten bewährt haben, in SVM-Fortran integriert wurden, wird ein Vergleich zwischen den bekanntesten Sprachen zur Programmierung massiv-paralleler Rechner durchgeführt. Die Fortran-Dialekte bzw. Sprachen High Performance Fortran (HPF) [29, 30], Vienna Fortran [15, 60], Fortran D [25] und der zur Zeit als Standardisierungsvorschlag vorliegende Entwurf ANSI-X3H5 (*Parallel Processing Model for High Level Programming Languages*) für Rechner mit (virtuell) gemeinsamem Speicher [3, 5] werden bezüglich des Aspektes der Datenverteilung und des Scheduling miteinander verglichen. Andere Arbeiten werden am Ende des Kapitels kurz vorgestellt.

Unter Scheduling versteht man hier, wie in Kapitel 1 bereits erwähnt, die Zuordnung von Ausführungsblöcken zu Prozessoren.

Die Sprachen HPF, Vienna Fortran und Fortran D basieren auf dem Ansatz der Datenverteilung. Unter **Datenverteilung** versteht man die Abbildung der Daten, meist Felder, eines Programms auf die lokalen Speicher der Prozessoren. Ziel ist es hierbei, die Datenbewegungen zwischen den lokalen Speichern der Prozessoren zu minimieren. Daher ist es sinnvoll, den Prozessoren solche Ausführungsblöcke zuzuweisen, die möglichst nur die Daten benutzen, die sich im lokalen Speicher befinden. Dies ist in der Literatur wie zum Beispiel in [18] auch als *owner-compute-rule* bekannt.

Im folgenden wird zuerst ein kurzer historischer Überblick über die Ursprünge und die Entwicklung der Compiler und Fortran-Dialekte für Parallelrechner gegeben. Nach einer einführenden Erklärung der für den Vergleich der Sprachen wichtigen Konzepte werden die Fortran-Dialekte und -Erweiterungen bezüglich ihrer Scheduling- und Datenverteilungskonzepte abschließend verglichen.

3.1 Historische Entwicklung

Frühe Arbeiten zur automatischen Vektorisierung von Programmen fanden bei Massachusetts Computer Associates, auch später bekannt als Compass, in den frühen 70er Jahren statt. Dort wurde ein Transformationswerkzeug [1] für den Illiac IV [6] entwickelt, das Befehlsabhängigkeiten durch einfache und effektive Datenabhängigkeitstests herausfand. Ein Problem bei diesem Transformator war, daß er viele unterschiedliche Abhängigkeitstypen miteinander vermischte. Aus diesem Transformator wurde schließlich der VAST-Vektorisierer weiterentwickelt, der erfolgreich von der Firma Pacific Sierra Corporation vertrieben wurde. Andere Firmen wie beispielsweise Cray Research Inc. entwickelten ebenfalls erfolgreich vektorisierende Compiler.

Mitte der 70er Jahre wurde das Parafrase-Projekt gegründet, das unter anderem auch von der Universität von Illinois, die an der Entwicklung des *Parallelizer* mitgewirkt hatte, unterstützt wurde. In diesem Projekt sollten sequentielle Programme zu äquivalenten parallelisierten oder vektorisierten Programmen umgeformt werden. Die Notation der Abhängigkeiten war wesentlich genauer als beim *Parallelizer*. Es gab eine eindeutige Klassifizierung von Datenabhängigkeiten und sehr gute Tests, um Abhängigkeiten festzustellen. Aufgrund der Ergebnisse dieses Projektes wurden Compiler entwickelt wie zum Beispiel KAP, ein Vektorisierer der Firma Kuck and Associates Inc.

Ende der 70er Jahre wurde das Parallel Fortran Converter (PFC) Projekt [1] an der Rice University gestartet. Zuerst sollten die Ergebnisse des Parafrase-Projektes weiterentwickelt werden, aber man entschloß sich, einen ganz neuen Zugang zu wählen. Das PFC Projekt diente als Prototyp für den vektorisierenden Compiler für die IBM-Großrechner, und die in diesem Projekt entwickelte Theorie dieses Projektes bildete die Basis für den vektorisierenden Compiler von Convex.

Anfang der 80er Jahre wollte man aus der Vektorisierung die Parallelisierung von Programmen weiterentwickeln, da Ähnlichkeiten wie zum Beispiel die Notwendigkeit zur Datenabhängigkeitsanalyse vorhanden waren. Dies führte zu zahlreichen Compiler- und Sprachentwicklungen für Rechner mit verteiltem Speicher wie z.B. Vienna Fortran [15, 60] und Fortran D[25]. Um in diesem Sprachbereich eine einheitliche, auf vielen Maschinen unterstützte Sprache zu erhalten, wurde das High Performance Fortran Forum (HPFF) gegründet. Ein Sprachentwurf dieses Forums ist HPF [29, 30].

Für Rechner mit gemeinsamem Speicher fanden ebenfalls zahl- und erfolgreiche Sprach- und Compiler-Entwicklungen statt. Bei den Firmen Cray Research Inc., IBM, Silicon Graphics und anderen existieren parallelisierende Compiler für Rechner mit gemeinsamem Hauptspeicher. Ein großes Problem für den Programmierer war jedoch die Inkompatibilität der Spracherweiterungen untereinander, sowohl in syntaktischer als auch in semantischer Hinsicht. Um eine größere Kompatibilität und Konsistenz zwischen den Sprachen zu erhalten, wurde 1993 ein Standardisierungsvorschlag zur Programmierung von Parallelrechnern mit gemeinsamem Speicher von

der ANSI veröffentlicht, der Standardisierungsvorschlag X3H5 "Parallel Processing Model for High Level Programming Languages" [3, 5].

3.2 Begriffe

In diesem Abschnitt werden einige grundlegenden Begriffe definiert, die bei der Beschreibung der Fortran-Dialekte benötigt werden. Zum einen sind dies die Verteilungsstrategien für Daten oder Ausführungsblöcke und zum anderen wichtige Begriffe im Zusammenhang mit Datenverteilungen.

3.2.1 Verteilungsstrategien

Bei der Verteilung von Daten auf Prozessoren bzw. dem Scheduling von Ausführungsblöcken gibt es zwei häufig benutzte Basisstrategien: die Blockverteilung und die zyklische Verteilung.

Zur formalen Beschreibung seien die folgenden Bezeichnungen eingeführt:

Sei P die Anzahl der Prozessoren. Ohne Beschränkung der Allgemeinheit seien die Prozessoren von 1 bis P durchnummeriert. A sei ein Tupel mit $A = (a_1, \dots, a_n)$, wobei n die Anzahl der Komponenten von A festlegt. A kann sowohl Daten eines Programms wie z.B. ein Feld als auch Ausführungsblöcke wie z.B. eine Schleife repräsentieren. Die Tupelkomponenten sind dann die einzelnen Feldelemente bzw. Schleifeniterationen. A' sei ein weiteres Tupel mit $A' = (a_i, a_{i+1}, \dots, a_{r-1}, a_r)$ mit $1 \leq i \leq r \leq n$, im folgenden **Segment** genannt, welches aus mehreren aufeinanderfolgenden, zusammenhängenden Komponenten des Tupels A besteht. Ein Segment wird einem Prozessor zugeordnet, d.h. seine Tupelkomponenten werden diesem Prozessor zugeteilt.

Blockverteilung

Bei der Blockverteilung werden die Komponenten eines Tupels A blockweise auf die Prozessoren verteilt.

Formal läßt sich dies wie folgt beschreiben:

Sei q definiert durch $q := \left\lfloor \frac{n}{P} \right\rfloor$.

Falls $q = \frac{n}{P}$ ist, dann wird A in P gleichgroße Segmente der Größe¹ q aufgeteilt, die auf die Prozessoren wie folgt verteilt werden:

Prozessor Nr.	Komponenten
1	$a_1, \dots, a_q$
2	$a_{q+1}, \dots, a_{2 \cdot q}$
$\vdots$	$\vdots$
P	$a_{n-q+1}, \dots, a_n$

Falls $n \nmid P$, dann wird in der Literatur zwischen zwei Möglichkeiten der Verteilung unterschieden. Die erste Möglichkeit, die z.B. in Vienna Fortran [15, 60] vorgeschlagen wird, sieht wie folgt aus:

$$\begin{aligned}
 n &= q \cdot P + r \\
 &= (q+1) \cdot r + q \cdot (P-r), \text{ mit } 0 < r < P.
 \end{aligned}$$

Die letzte Schreibweise drückt aus, daß A in r Segmente der Größe $q+1$ und $P-r$ Segmente der Größe q aufgeteilt wird. Die Größe eines Segmentes unterscheidet sich daher von der der anderen Segmente um höchstens eins. Die Verteilung der Segmente auf die Prozessoren sieht dann wie folgt aus:

Prozessor Nr.	Komponenten	Größe des Segmentes
1	$a_1, \dots, a_{q+1}$	$q+1$
2	$a_{q+2}, \dots, a_{2 \cdot (q+1)}$	$q+1$
$\vdots$	$\vdots$	$\vdots$
r	$a_{(r-1) \cdot (q+1) + 1}, \dots, a_{r \cdot (q+1)}$	$q+1$
$r+1$	$a_{r \cdot (q+1) + 1}, \dots, a_{r \cdot (q+1) + q}$	q
$r+2$	$a_{r \cdot (q+1) + q + 1}, \dots, a_{r \cdot (q+1) + 2 \cdot q}$	q
$\vdots$	$\vdots$	$\vdots$
P	$a_{n-q+1}, \dots, a_n$	q

¹Die Größe eines Segmentes sei definiert als die Anzahl der Tupelkomponenten.

Bei der zweiten Möglichkeit, wie sie in HPF [29, 30] vorgestellt ist, wird wie folgt verfahren:

Setze $p^* := \left\lfloor \frac{n}{q+1} \right\rfloor + 1$. Dann erhalten alle Prozessoren p mit $1 \leq p < p^*$ Segmente der Größe $q+1$. Der Prozessor p^* erhält die restlichen $n \bmod (q+1)$ Tupelkomponenten von A , falls $n > P$, ansonsten erhält dieser Prozessor kein Segment. Falls $p^* < P$ ist, dann erhalten die Prozessoren p' mit $p^* < p' \leq P$ kein Segment.

Falls $p^* = P$ ist, dann werden den Prozessoren folgende Tupelkomponenten zugewiesen:

Prozessor Nr.	Komponenten	Größe des Segmentes
1	$a_1, \dots, a_{q+1}$	$q+1$
2	$a_{q+2}, \dots, a_{2 \cdot (q+1)}$	$q+1$
$\vdots$	$\vdots$	$\vdots$
$P = p^*$	$a_{(P-1) \cdot (q+1) + 1}, \dots, a_n$	$n \bmod (q+1)$

Falls $p^* < P$ und $n > P$, dann sieht die Zuordnung der Tupelkomponenten zu den Prozessoren folgendermaßen aus:

Prozessor Nr.	Komponenten	Größe des Segmentes
1	$a_1, \dots, a_{q+1}$	$q+1$
2	$a_{q+2}, \dots, a_{2 \cdot (q+1)}$	$q+1$
$\vdots$	$\vdots$	$\vdots$
p^*	$a_{(p^*-1) \cdot (q+1) + 1}, \dots, a_n$	$n \bmod (q+1)$
$p^* + 1$	–	0
$\vdots$	$\vdots$	$\vdots$
P	–	0

Ist im Fall $p^* < P$ aber $n < P$, dann unterscheidet sich die Verteilung nur für Prozessor p^* : Prozessor p^* wird keine Komponente des Tupels zugewiesen.

An einem Beispiel soll der Unterschied zwischen der Blockverteilung in Vienna Fortran und HPF verdeutlicht werden. Es seien $n = 11$ Komponenten auf $P = 8$ Prozessoren zu verteilen. Die Komponenten werden wie folgt verteilt:

Prozessor	HPF	Vienna Fortran
1	1,2	1,2
2	3,4	3,4
3	5,6	5,6
4	7,8	7
5	9,10	8
6	11	9
7	–	10
8	–	11

Zyklische Verteilung

Zyklische Verteilungen mit einem Parameter l mit $l \geq 1$ ordnen l aufeinanderfolgende Komponenten des Tupels A jeweils einem Segment zu und verteilen diese Segmente dann nach dem *round-robin*-Verfahren auf die Prozessoren. Eine Zuordnung von 20 Elementen auf drei Prozessoren mit dem Parameter $l = 2$ lässt sich folgendermaßen darstellen:

Prozessor	Elemente			
1	1, 2,	7, 8,	13, 14,	19, 20
2	3, 4,	9, 10,	15, 16	
3	5, 6,	11, 12,	17, 18	

Formal lässt sich dies wie folgt beschreiben:

Allen Komponenten a_i von A mit $1 \leq i \leq n$ wird der Prozessor p ($1 \leq p \leq P$) mit

$$p := \text{mod} \left(\left\lfloor \frac{i-1}{l} \right\rfloor, P \right) + 1$$

zugeordnet. Die Verteilung auf die Prozessoren sieht allgemein wie folgt aus:

Prozessor Nr.	Komponenten		
1	$a_1, \dots, a_l,$	$a_{P \cdot l + 1}, \dots, a_{(P+1) \cdot l}$	$\dots$
2	$a_{l+1}, \dots, a_{2 \cdot l},$	$a_{(P+1) \cdot l + 1}, \dots, a_{(P+2) \cdot l}$	$\dots$
3	$a_{2l+1}, \dots, a_{3 \cdot l},$	$a_{(P+2) \cdot l + 1}, \dots, a_{(P+3) \cdot l}$	$\dots$
$\vdots$		$\vdots$	
P	$a_{(P-1) \cdot l + 1}, \dots, a_{P \cdot l},$	$a_{(2 \cdot P - 1) \cdot l + 1}, \dots, a_{2P \cdot l},$	$\dots$

3.2.2 Alignment

In einigen Sprachen zur Datenverteilung müssen Felder nicht unbedingt direkt auf Prozessoren verteilt werden, sondern sie können auch durch ein *Alignment* relativ zueinander ausgerichtet werden. Ein Feld ist hierbei das Zielfeld, nach dem das andere Feld ausgerichtet wird. Sowohl bei dem Zielfeld als auch bei dem Feld, nach dem ausgerichtet wird, können Bereiche anstelle von ganzen Feldern zueinander ausgerichtet werden. Beispielsweise könnte das Feld A[10] mit dem Feld C[20] wie folgt ausgerichtet sein:

ALIGN A[i] WITH C[2*i]

Dies bedeutet, daß A[1] mit C[1], A[2] mit C[4], A[3] mit C[6], usw. ausgerichtet wird.

Der Vorteil besteht darin, daß sich bei der Verteilung der Felder auf die Prozessoren die einzelnen Feldelemente der Ausrichtung entsprechend auf demselben Prozessor befinden. Dadurch können Operationen zwischen diesen Feldelementen effizienter ausgeführt werden. Außerdem kann die Kommunikation zwischen den Prozessoren vermindert werden [39]. Für das Beispiel bedeutet dies, daß A[1] und C[1], A[2] und C[4], A[3] und C[6] etc. jeweils auf demselben Prozessor abgespeichert werden.

3.2.3 Datenverteilungsmodell

Den Fortran-Dialekten HPF, Vienna Fortran und Fortran D liegen ähnliche Datenverteilungsmodelle zugrunde, die sich aus mehreren Grundabbildungsfunktionen auf verschiedenen Stufen zusammensetzen. Im folgenden wird ein allgemeines Datenverteilungsmodell erläutert. Bei den Dialekten werden dann im einzelnen die Unterschiede zu diesem Modell aufgezeigt.

Das Datenverteilungsmodell, dargestellt in Abbildung 3.1, setzt sich prinzipiell aus drei Stufen zusammen. In der ersten Stufe kann man mit der *Alignment*-Abbildungsfunktion, wie oben beschrieben, Felder relativ zueinander ausrichten. Dies kann

im Modell nur mit Hilfe eines Templates geschehen, d.h. Felder können nur über ein Template mittels der *Alignment*-Abbildungsfunktion ausgerichtet werden. Templates können wiederum auf Prozessoren verteilt werden, wodurch dann die mit ihnen ausgerichteten Felder ebenfalls auf dieselben Prozessoren verteilt werden. Ein Vorteil des Template-Konzeptes besteht darin, daß bei gleicher Verteilung mehrerer Felder die Verteilung nur einmal angegeben und abgespeichert werden muß.

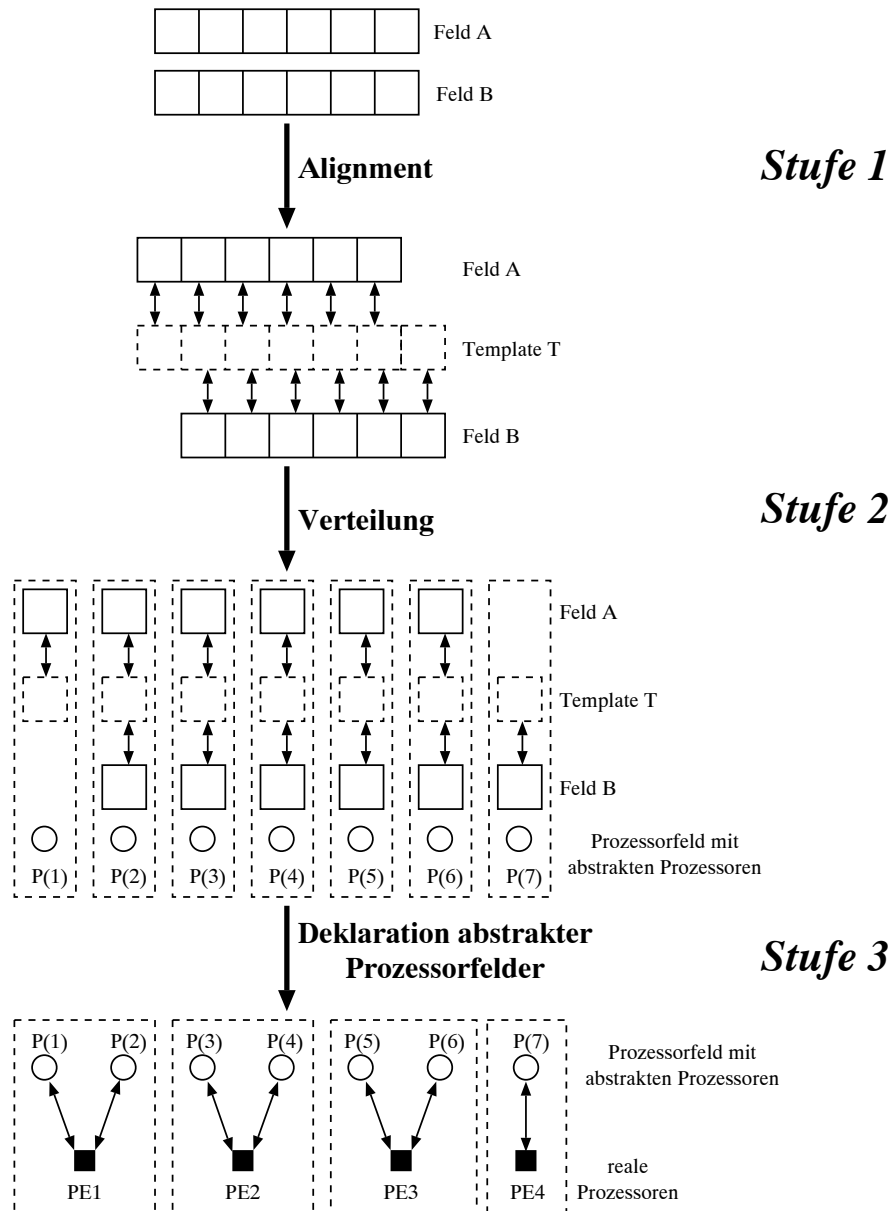


Abbildung 3.1: Ein allgemeines Datenverteilungsmodell

Bei der Verteilung der Templates auf die Prozessoren wird in der zweiten Stufe des Datenverteilungsmodells das Hilfsmittel der Prozessorfelder verwendet. Prozessorfelder sind Felder mit abstrakten Prozessoren. Sie vereinfachen die Programmierung, da man Prozessorfelder definieren kann, die der Problemstruktur angepaßt sind und

nicht die reale Prozessorstruktur auf Rechnerebene im Programm berücksichtigen müssen. Dadurch wird auch eine Portierung der Programme auf unterschiedliche Rechnerarchitekturen erleichtert.

Die Abbildung der Prozessorfelder auf die realen Prozessoren in der dritten Stufe des Datenverteilungsmodells wird beim Übersetzen oder bei der Ausführung des Programms durchgeführt. Wie diese Verteilung konkret erfolgt, ist von der Implementierung der Compiler oder Laufzeitsysteme abhängig und kann vom Programmierer im allgemeinen nicht beeinflußt werden.

3.3 Datenverteilung in HPF

HPF [29, 30] ist eine Erweiterung von Fortran 90 [35]. HPF ist in der Absicht entwickelt worden [30], eine gemeinsame Programmierbasis auf möglichst vielen Parallelrechnern zu bieten, insbesondere auf **SIMD** (Single Instruction Multiple Data) und **MIMD** (Multiple Instruction Multiple Data) Systemen².

In HPF wird ein Datenverteilungsmodell verwendet, welches dem in Abschnitt 3.2.3 vorgestellten Modell entspricht. Die Datenverteilung findet in HPF auch in drei Stufen statt. Zuerst werden Felder gemäß eines Templates ausgerichtet. Falls Felder direkt auf die Prozessoren verteilt werden, was in HPF erlaubt ist, dann wird beim Übersetzen des Programms oder zur Laufzeit automatisch ein Template angelegt, das die Verteilung festhält, so daß die erste Stufe, wenn auch implizit, immer ausgeführt wird.

Bei der Verteilung der Templates oder Felder auf die Prozessorfelder in der zweiten Stufe gibt es verschiedene Strategien. Die Blockverteilung und die zyklische Verteilung, wie in Abschnitt 3.2.1 definiert, sind in HPF standardmäßig vorhanden. Zusätzlich kann man bei der Blockverteilung noch einen Parameter angeben, der die Größe der Segmente angibt. Diese Art der Blockverteilung ist jedoch redundant zur zyklischen Verteilung.

Auf die dritte Stufe im Datenverteilungsmodell, der Zuordnung des Prozessorfeldes auf die realen Prozessoren, kann der Programmierer nur Einfluß nehmen, indem er die maximale Anzahl der Prozessoren festlegt, die ein Programm ausführen soll.

Ein weiterer wesentlicher Aspekt bei der Datenverteilung ist der Parameterübergabemechanismus bei Funktionsaufrufen. Bei HPF können formale Argumente in Funktionen das Template als aktuelles Argument erben. Dabei wird eine Kopie des Templates angelegt. Das formale Argument erbt die Dimension, die Ausdehnung und die Verteilung des aktuellen Argumentes. Falls sich die Verteilung innerhalb der Funktion ändert, dann kann dies jedoch nicht an das aktuelle Argument zurückgegeben werden. Dies entspricht dem Call-By-Value-Übergabemechanismus.

²Diese Klassifizierung der Parallelrechner wurde auf diese Art und Weise von Flynn [23] erstmals eingeführt.

3.4 Datenverteilung in Vienna Fortran

Vienna Fortran [15, 60] ist eine maschinenunabhängige Spracherweiterung von Fortran77, die es dem Benutzer erlaubt, Programme für Systeme mit verteiltem Speicher zu schreiben, wobei Vienna Fortran es ermöglicht, das System als einen Rechner mit einem globalen Speicheradreibraum anzusehen.

Das Datenverteilungsmodell, das der Spracherweiterung zugrunde liegt, ist ähnlich dem in Abschnitt 3.2.3 beschriebenen Modell. Vienna Fortran unterstützt das Konzept der Templates jedoch nicht, hier werden in Stufe 1 die Felder direkt zueinander ausgerichtet. Die Ausrichtung geschieht wie in Abschnitt 3.2.2 beschrieben.

Zusätzlich gibt es ein Functional Alignment. Beim **Functional Alignment** wird die Ausrichtung der Felder in einer Funktion durchgeführt. Die zueinander auszurichtenden Felder werden der Funktion als Parameter übergeben. Wenn man die Ausrichtung zweier Felder ändern will, dann braucht man dies nur lokal in der Funktion durchzuführen, was eine bessere Wartbarkeit des Programms mit sich bringt.

Die Verteilung der Felder auf die Prozessorfelder kann blockweise oder zyklisch erfolgen. Sie erfolgt ähnlich wie in Stufe 2 des Datenverteilungsmodells, hier werden die Felder nicht indirekt über ein Template, sondern direkt auf die Prozessorfelder verteilt. Eine weitere Verteilungsstrategie ist die indirekte Verteilung.

Eine **indirekte Verteilung** ist eine Verteilung eines Feldes $B = (b_1, \dots, b_n)$ mittels eines Abbildungsfeldes $C = (c_1, \dots, c_m)$, welches zur Laufzeit definiert werden kann. Dabei muß $n \leq m$ sein, und c_i mit $1 \leq i \leq n$ muß ein Element eines Prozessorfeldes repräsentieren. Jedes Element b_i mit $1 \leq i \leq n$ wird dann auf den durch c_i definierten Prozessor abgespeichert.

Die Zuordnung der Prozessorfelder auf die realen Prozessoren kann man in Vienna Fortran nicht beeinflussen. Es ist gestattet, für dieselben realen Prozessoren mehrere Prozessorfelder zu definieren. Dies kann ein Programm vereinfachen, wenn verschiedene Sichten auf dieselbe physikalische Prozessorstruktur benötigt werden.

Der Parameterübergabemechanismus bei Funktionsaufrufen ist bei Vienna Fortran sehr flexibel, da man durch Setzen von Attributen erreichen kann, daß sich Änderungen der Verteilung eines Feldes in der aufgerufenen Funktion nur auf das formale Argument auswirken bzw. auch an das aktuelle Argument weitergegeben werden können. Vienna Fortran unterstützt also den Call-By-Value- und Call-By-Reference-Übergabemechanismus.

3.5 Datenverteilung in Fortran D

Fortran D [25] wurde entwickelt, um ein maschinenunabhängiges, paralleles Programmiermodell zu erhalten, das man für Rechnerarchitekturen verwenden kann, die einen gemeinsamen oder verteilten Speicher haben.

Das in dieser Sprache verwendete Datenverteilungsmodell unterscheidet sich im

wesentlichen von dem in Abschnitt 3.2.3 vorgestellten Modell darin, daß es nicht möglich ist, Prozessorfelder zu definieren. Außerdem heißen die Templates in Fortran D Dekompositionen (*decompositions*). Funktional besteht aber zwischen Dekompositionen und Templates kein Unterschied. In Fortran D müssen die Felder mit einer Dekomposition ausgerichtet werden, es besteht hier nicht die Möglichkeit, Felder direkt zueinander auszurichten wie beispielsweise in HPF. Werden zwei Felder mit derselben Dekomposition ausgerichtet, dann sind diese automatisch zueinander ausgerichtet.

Die Dekompositionen können auf die realen Prozessoren verteilt werden. Die Verteilungsstrategie kann eine wie in Abschnitt 3.2.1 definierte HPF-Blockverteilung, eine zyklische oder eine irreguläre Verteilung sein.

Eine **irreguläre Verteilung** bildet eine Dekomposition mittels eines Abbildungsfeldes, das der Dekomposition übergeben wird, auf die Prozessoren ab. Die Funktionalität ist die gleiche wie bei der indirekten Verteilung in Vienna Fortran.

Da in Fortran D eine Prozessorfelddeklaration nicht möglich ist, kann die Verteilung der Dekomposition auf die Prozessoren nur erfolgen, indem man in einer Dimension einer zu verteilenden Dekomposition die maximale Anzahl der Prozessoren angibt, auf die diese Dimension verteilt werden soll. Das Produkt der Prozessoren in jeder Dimension darf die Gesamtzahl der dem Programm zur Verfügung stehenden Prozessoren nicht überschreiten.

In Fortran D hat man beim Parameterübergabemechanismus bei Funktionsaufrufen lediglich die Möglichkeit, verteilte Felder an formale Argumente weiterzugeben. Da die Verteilung der Felder in einer Dekomposition festgehalten wird und der Gültigkeitsbereich der Dekomposition nur auf die umfassende Prozedur beschränkt ist, können Änderungen der Verteilung eines Feldes, die innerhalb einer Funktion stattfinden, nicht an das aktuelle Argument zurückgegeben werden. Dies entspricht dem Call-By-Value-Übergabemechanismus, der in HPF ebenfalls nur anwendbar ist.

3.6 Scheduling in ANSI-X3H5

Das dem Standardisierungsvorschlag ANSI-X3H5 [3, 5] zugrundeliegende Programmiermodell ist das eines (virtuell) gemeinsamen Speichers. In dem Standardisierungsvorschlag ist es nicht vorgesehen, die Verteilung der Daten auf die lokalen Speicher der Prozessoren zu beeinflussen, stattdessen basiert dieser Ansatz auf dem der Arbeitsverteilung, d.h. hier kann das Scheduling gesteuert werden.

Am Anfang werden zuerst einige Begriffe erläutert, die in dem Standardisierungsvorschlag benutzt werden. Ein **Prozeß** (*process*) kann aus einem virtuellen Prozessor, einer Datenumgebung und einem Befehlszeiger bestehen. Ein **aktiver Prozeß** (*active process*) hat alle drei Komponenten, ein **blockierter Prozeß** (*blocked process*) hat einen Befehlszeiger und eine Datenumgebung, aber keinen virtuellen Prozessor, und ein **nach Arbeit suchender Prozeß** (*looking-for-work process*) hat einen virtuellen Prozessor und eine Datenumgebung, aber keinen Befehlszeiger. Ein **Team**

besteht aus einem oder mehreren Prozessen, die ein paralleles Konstrukt gemeinsam bearbeiten. Alle Team-Mitglieder sind gleichwertig und können von dem Benutzer einzeln nicht angesprochen werden, das heißt, der Benutzer kann nicht angeben, daß ein bestimmtes Team-Mitglied einen Ausführungsblock wie zum Beispiel eine Schleifeniteration bearbeiten soll.

Program begin ein einzelner Prozessor ist hier aktiv

Durchführung serieller Aktionen

Parallel construct begin Team wird hier gebildet

Initialisierungen, von allen Team-Mitgliedern ausgeführt

Worksharing construct begin Scheduling auf die Team-Mitglieder

jedes Team-Mitglied führt seine Ausführungsblöcke durch

Worksharing construct end

hier können sich Aktionen befinden, die von allen Team-Mitgliedern redundant ausgeführt werden

Grouping construct begin Eintritt von Team-Mitgliedern

solange Ausführungsblöcke im Arbeitsteilungskonstrukt
vorhanden sind

replizierter Code - Initialisierungen für Arbeitsteilungskonstrukt

Worksharing construct begin

Worksharing construct end

Grouping construct end

Parallel construct end Team wird aufgelöst

Durchführung serieller Aktionen

Program end

Abbildung 3.2: Beispiel Ausführungsmodell

Bei dem Ausführungsmodell in dem Standardisierungsvorschlag, beispielhaft dargestellt in Abbildung 3.2, beginnt zunächst ein einzelner Prozeß mit der Ausführung des Programms. Wenn dieser Prozeß auf ein paralleles Konstrukt (*parallel construct*) trifft, wird ein Team von Prozessoren gebildet.

In dem parallelen Konstrukt befinden sich dann die Konstrukte zur parallelen Ausführung. Wenn in dem parallelen Konstrukt ein Arbeitsteilungskonstrukt (*work-sharing construct*) vorhanden ist, dann führen einzelne Team-Mitglieder die Ausführungsblöcke in diesem parallel aus. Um eine parallele Ausführung von Ausführungsblöcken in einem Programm überhaupt zu erreichen, muß ein Arbeitsteilungskonstrukt in einem parallelen Konstrukt auftreten. Weiterhin kann in diesem auch ein Gruppierungskonstrukt (*grouping construct*) auftreten. Dieses stellt einen Mechanismus zur Verfügung, um replizierten Code und Arbeitsteilungskonstrukte miteinander zu gruppieren bzw. zu assoziieren.

Bei den **Arbeitsteilungskonstrukten** unterscheidet man iterative und nichtiterative. Die iterativen Arbeitsteilungskonstrukte enthalten einen Ausführungsblock, der mehrfach ausgeführt werden kann wie z.B. eine Schleife, wohingegen die nichtiterativen

Konstrukte disjunkte Ausführungsblöcke enthalten. In beiden Fällen können die Ausführungsblöcke von unterschiedlichen Team-Mitgliedern parallel ausgeführt werden. Weiterhin kann man spezifizieren, ob die Ausführung der einzelnen Blöcke wie im seriellen Fall geschehen soll. Dies bedeutet, daß die Ausführungsblöcke den Team-Mitgliedern in derselben Reihenfolge zugewiesen werden, wie sie im seriellen Fall abgearbeitet würden. Die Zuordnung eines speziellen Ausführungsblockes zu einem Team-Mitglied ist nicht vorgesehen, man kann nur angeben, ob ein Block parallel oder sequentiell ausgeführt werden soll. Dadurch ist es nicht möglich, ein Scheduling auf die Team-Mitglieder nach den in Abschnitt 3.2.1 definierten Strategien durchzuführen.

Bei den **Gruppierungskonstrukten** werden replizierter Code und Arbeitsteilungskonstrukte miteinander verbunden. In dem replizierten Code könnten beispielsweise Initialisierungen vorgenommen werden, die zur Ausführung des Arbeitsteilungskonstruktes nötig sind. Wenn nun in dem Arbeitsteilungskonstrukt keine zu bearbeitenden Ausführungsblöcke vorhanden sind, dann durchlaufen die Team-Mitglieder, die auf dieses Gruppierungskonstrukt treffen, dieses nicht mehr.

3.7 Vergleich der Konzepte

Zuerst werden die Fortran-Dialekte HPF, Vienna Fortran und Fortran D bzgl. der Datenverteilungsstrategie untereinander verglichen. Danach werden diese Konzepte denen des Scheduling im Standardisierungsvorschlag gegenübergestellt und Vor- und Nachteile aufgezeigt.

Bei allen drei Fortran-Dialekten ist es möglich, Felder zueinander auszurichten und anschließend auf Prozessoren zu verteilen. Bei HPF und Fortran D geschieht dies über Templates bzw. Dekompositionen. Zuerst werden die Felder mit den Templates ausgerichtet und dann auf die Prozessoren verteilt. Bei Vienna Fortran hingegen werden die Felder direkt zueinander ausgerichtet und dann auf die Prozessoren verteilt. Dies ist keine wesentliche Einschränkung, da es möglich ist, Programme von Sprachen mit Template-Konzept in Sprachen ohne Template-Konzept, aber mit Alignment, zu transformieren [16].

Bei der Verteilung der Felder oder Templates auf die Prozessoren unterstützen alle drei Dialekte standardmäßig die Blockverteilung und die zyklische Verteilung. Bei HPF und Fortran D ist die Blockverteilung derart definiert, daß bei der Datenverteilung eine einfachere Berechnung durchgeführt werden kann. Der Nachteil liegt darin, daß auf einigen Prozessoren evtl. gar keine Daten abgespeichert werden, was bei der Ausführung einer parallelen Schleife durch die *owner-compute-rule* zu einer schlechten Lastbalance führen kann. Dies geschieht bei der Blockverteilung, wie sie bei Vienna Fortran vorgenommen wird, nicht. Hier unterscheidet sich die Anzahl der Iterationen, die die Prozessoren ausführen müssen, um maximal eins.

Fortran D und Vienna Fortran haben zusätzlich noch eine irreguläre bzw. indirekte Verteilung, die durch ein Abbildungsfeld realisiert wird. Dies ermöglicht dem Programmierer, unregelmäßige Verteilungen zur Laufzeit zu definieren. HPF hat kein

Konzept, das eine Verteilung dieser Art vorsieht, es soll aber in Zukunft dahingehend erweitert werden [17].

Die Verteilung der Felder oder Templates auf die Prozessoren wird bei HPF und Vienna Fortran durch Prozessorfelder geleistet. Die Felder werden auf die Prozessorfelder verteilt, und ein Compiler oder Laufzeitsystem übernimmt die Aufgabe, die Prozessorfelder auf die realen Prozessoren abzubilden. In Fortran D ist eine Definition der Prozessorfelder nicht möglich. Hier werden Dimensionen von Dekompositionen auf eine maximale Anzahl von Prozessoren aufgeteilt, was eine erhebliche Einschränkung bzgl. der Kontrolle über eine konkrete Zuordnung eines Feldelementes zu einem bestimmten Prozessor darstellt.

Der Parameterübergabemechanismus bietet bei Vienna Fortran die größte Funktionalität. Dort hat man die wie in Abschnitt 3.4 beschriebenen Möglichkeiten, bei Fortran D und HPF hingegen ist man sehr eingeschränkt, dort kann man nur eine Verteilung an ein formales Argument weitergeben. Wenn sich diese Verteilung in der aufgerufenen Funktion ändern sollte, dann kann dies nicht an das aktuelle Argument der aufrufenden Funktion zurückgegeben werden.

Bei den Sprachen, die auf einer Datenverteilung basieren, können Ausführungsblöcke wie z.B. die einzelnen Schleifeniterationen mit einem ON-HOME-Konstrukt verteilt werden. Dieses Konstrukt verteilt die Schleifeniterationen in der gleichen Weise auf Prozessoren wie ein Template oder Feld, das dem ON-HOME-Konstrukt als Argument übergeben wird und das bereits auf Prozessoren verteilt sein muß. Dadurch kann man die Verteilung der Ausführungsblöcke mit den gleichen Strategien wie die Datenverteilung vornehmen. Dies bedeutet, man kann Ausführungsblöcke blockweise, zyklisch und auf eine durch den Benutzer bestimmte Weise verteilen.

Während es bei dem Standardisierungsvorschlag zwar möglich ist, Parallelität zu spezifizieren, wird auf die Strategie des Scheduling nicht eingegangen. Dort kann nur angegeben werden, ob Ausführungsblöcke auf Team-Mitglieder verteilt werden oder nicht, aber es kann keine Strategie spezifiziert werden, nach der dies geschehen soll. Diese Entscheidung wird auf einer anderen Ebene getroffen, z.B. durch einen Compiler oder ein Laufzeitsystem. Dies kann den Benutzer entlasten, aber er verliert dadurch die Kontrolle über die Bearbeitungsreihenfolge der einzelnen Ausführungsblöcke und die Zuweisung dieser zu den Team-Mitgliedern. Folglich kann der Benutzer beispielsweise nicht steuern, daß einem Team-Mitglied nur die Ausführungsblöcke zugewiesen werden, die die Daten im lokalen Speicher eines Team-Mitgliedes benutzen. Der Benutzer kann also die Speicherzugriffe nicht beeinflussen, bei dem Konzept der Datenverteilung hingegen kann er durch eine geeignete Datenverteilung in Verbindung mit dem ON-HOME-Konstrukt die Speicherzugriffe auf die einzelnen Prozessoren beeinflussen und somit unter Umständen beispielsweise *Cache-Faults* oder Seitenfehler reduzieren.

Zusammenfassend kann man sagen, daß der Benutzer bei den vorgestellten Fortran-Dialekten mit der Datenverteilung mehr Kontrolle über die Speicherzugriffe und über das Scheduling hat als beim Standardisierungsvorschlag. Dies bedeutet aber zugleich, daß der Benutzer im allgemeinen ein detailliertes Verständnis des Programms haben muß sowie vollständige Informationen dem Compiler oder Laufzeit-

system bzgl. der Verteilung von Feldern übergeben muß. Bei dem Standardisierungsvorschlag hingegen werden wichtige Details, wie etwa Verteilungsstrategien, nicht spezifiziert und sind somit implementierungsabhängig.

3.8 Weitere Arbeiten

Ein anderer Fortran-Dialekt ist **Fortran M** [24], der eine Erweiterung von Fortran77 ist. Er enthält Konstrukte, die *message passing* unterstützen. In diesem Dialekt sind **Prozesse** Programmodule, die aus Daten und Ausführungsblöcken bestehen und nebenläufig ausgeführt werden können. Prozesse können über Kanäle miteinander kommunizieren. Die Kommunikationsmöglichkeiten sind das Senden und Empfangen von Nachrichten. Prozesse können über spezielle Konstrukte aktiviert werden, die in SVM-Fortran der PSECTION- bzw. der PDO-Direktive entsprechen. Weiterhin stellt die Sprache für die Datenverteilung und das Scheduling ähnliche Konstrukte wie in HPF zur Verfügung.

Mit Fortran M ist die Erstellung paralleler Programme aufwendig, da der Benutzer die Programmierung der Kommunikation selbst durchführen muß.

Fx Fortran [54, 55] ist eine Fortran-Erweiterung, die das Konzept der *parallel section* beinhaltet. Dieses Konstrukt erlaubt die parallele Ausführung von Unterprogrammen, die als *Tasks* bezeichnet werden. *Tasks* kommunizieren über Argumente. Diese können der *Task* bei der Aktivierung übergeben werden und, nachdem eine *Task* terminiert ist, von ihr empfangen werden. Jede Aktivierung einer *Task* wird von einer *Input*- und *Output*-Direktive begleitet, die die geteilten Argumente spezifiziert.

Fx Fortran ist für solche Probleme gut geeignet, die sich in *Tasks* aufteilen lassen, die nur am Anfang und am Ende Kommunikation benötigen. Ansonsten muß eine *Task* in kleinere Einheiten aufgeteilt, eine Schnittstelle innerhalb dieser *Tasks* definiert und diese Einheiten dann neuen *Tasks* zugeordnet werden. Dabei ist zu berücksichtigen, daß *Overhead* für die Aktivierung und Terminierung einer zusätzlichen *Task* anfällt.

Die **Parti**-Routinen [20, 53] ermöglichen es, parallele Schleifen, die auf Felder mit unregelmäßigen Verteilungsmustern zugreifen, effizient zu bearbeiten. Es wurde eine aus Primitiven bestehende Bibliothek entwickelt, die die Benutzung von unregelmäßig verteilten Feldern, deren Verteilung zur Laufzeit erst bekannt ist, unterstützt. Die Verteilung eines Feldes auf die Prozessoren wird in einem Abbildungsfeld (*mapping array*) festgehalten, wobei der Wert eines Abbildungsfeldelementes die Nummer des Prozessors angibt, wo das Feldelement des ursprünglichen Feldes mit demselben Index gespeichert wird. Dieses Abbildungsfeld hat denselben Speicherplatzbedarf wie das ursprüngliche Feld, aber es erlaubt die Speicherung von unregelmäßigen Verteilungen. Die Ausführung einer Schleife geschieht in zwei Phasen. In der ersten Phase, der *Inspector*-Phase, werden Berechnungen zur Reduzierung der nichtlokalen Speicherzugriffe durchgeführt. In der zweiten Phase werden die

se Ergebnisse genutzt, die Feldelemente umverteilt und die Schleife ausgeführt. Im Anschluß wird die ursprüngliche Verteilung wiederhergestellt.

Kapitel 4

Benutzergesteuertes Scheduling in SVM-Fortran

SVM-Fortran unterstützt das benutzergesteuerte Scheduling durch die Bereitstellung der PSECTION- und PDO-Direktive. Die PSECTION-Direktive dient dazu, parallele Sektionen zu spezifizieren. Diese Direktive wird im folgenden nicht mehr betrachtet, da ihre Implementierung nicht Gegenstand der Arbeit ist.

Mit der PDO-Direktive kann der Benutzer die Iterationen einer Schleife als parallel ausführbar markieren. Zunächst wird die Syntax der PDO-Direktive mit den wichtigsten Optionen dargestellt, die in diesem Kapitel erläutert werden. Die vollständige Syntax befindet sich in Anhang A.

Syntax:

<i>parloop-directive</i>	is	PDO [(<i>parloop-option-list</i>)]
<i>parloop-option</i>	is	LOOPS (<i>loop-var-list</i>)
	or	NOBARRIER
	or	REDUCTION (<i>variable-name-list</i>)
	or	PROCESSORS (<i>processors-name</i> [(<i>dist-spec-list</i>)])
	or	STRATEGY (<i>strat-spec</i>)
<i>strat-spec</i>	is	<i>strat-spec1-list</i>
	or	ON_HOME (<i>template-name</i> (<i>templ-spec-list</i>))
<i>strat-spec1</i>	is	*
	or	BLOCK
	or	CYCLIC [(<i>int-expr</i>)]
<i>templ-spec</i>	is	<i>int-expr</i>
	or	*
<i>loop-var</i>	is	<i>variable-name</i>

Mit der PDO-Direktive können Schleifen als parallel ausführbar markiert werden. Wenn mehrere Schleifen gleichzeitig parallelisiert werden sollen, dann müssen sie perfekt geschachtelt sein, d.h. zwischen den einzelnen Schleifen dürfen sich keine

ausführbaren Befehle befinden.

Mit der LOOPS-Option werden die zu parallelisierenden Schleifen durch Angabe der Laufvariablen in der Option spezifiziert. Diese Variablen müssen im lokalen Speicher eines jeden Prozessors gespeichert und daher mit der PRIVATE-Direktive deklariert worden sein. Mit der NOBARRIER-Option kann angegeben werden, daß vor und nach der parallelen Schleife keine Synchronisation zwischen allen Prozessoren, die diese Schleife ausführen, stattfinden soll. Standardmäßig wird diese Synchronisation durchgeführt.

Mit der REDUCTION-Option können, wie in Abschnitt 2.3 bereits beschrieben, Reduktionsoperationen in der Schleife durchgeführt werden. In der PROCESSORS-Option kann man angeben, welche Prozessoren eine Schleife ausführen sollen. Diese müssen im APS enthalten sein.

```

PROGRAM PDO_LOOP_1
  INTEGER I

C *** Speichere Variable I im lokalen Speicher der Prozessoren
CSVM$ PRIVATE::I

C *** PDO-Direktive
CSVM$ PDO(LOOPS(I))
      DO 10 I = 1, 1000
          . . . . .
10 CONTINUE

      END

```

In dem Programm PDO_LOOP_1 wird die I-Schleife parallel ausgeführt. Vor und nach der Schleife findet eine Synchronisation aller Prozessoren, die auf diese Direktive treffen, statt. Die Schleife wird mit einer *Default*-Strategie auf die Prozessoren im APS verteilt. Sollen die Schleifen jedoch mit einer anderen als der *Default*-Strategie verteilt werden, so kann dies durch einen entsprechenden Eintrag in der STRATEGY-Option erreicht werden. Weiterhin kann an dieser Stelle auch definiert werden, ob das Schleifen-Scheduling direkt, dynamisch oder über ein Template geschehen soll. Im folgenden werden diese drei Arten des Scheduling erläutert und gegenübergestellt.

4.1 Direktes Schleifen-Scheduling

Beim direkten Schleifen-Scheduling werden die Schleifeniterationen entweder direkt auf die Prozessoren des Schleifen-APS oder auf die Prozessoren, die in der PROCESSORS-Option spezifiziert sind, verteilt. Die Zuordnung der Iterationen auf die Prozessoren wird nicht in einer Datenstruktur abgespeichert, sondern es wird

Code generiert, der die Verteilungsstrategie implementiert. Als Strategie kann die Blockverteilung, die zyklische Verteilung oder die Replikation gewählt werden¹. Die Blockverteilung ist identisch mit der HPF-Blockverteilung. Diese und die zyklische Verteilung wurden bereits in Abschnitt 3.2.1 erläutert. Bei der Replikation, die durch einen Stern markiert ist, wird eine Schleife sequentiell auf jedem Prozessor ausgeführt.

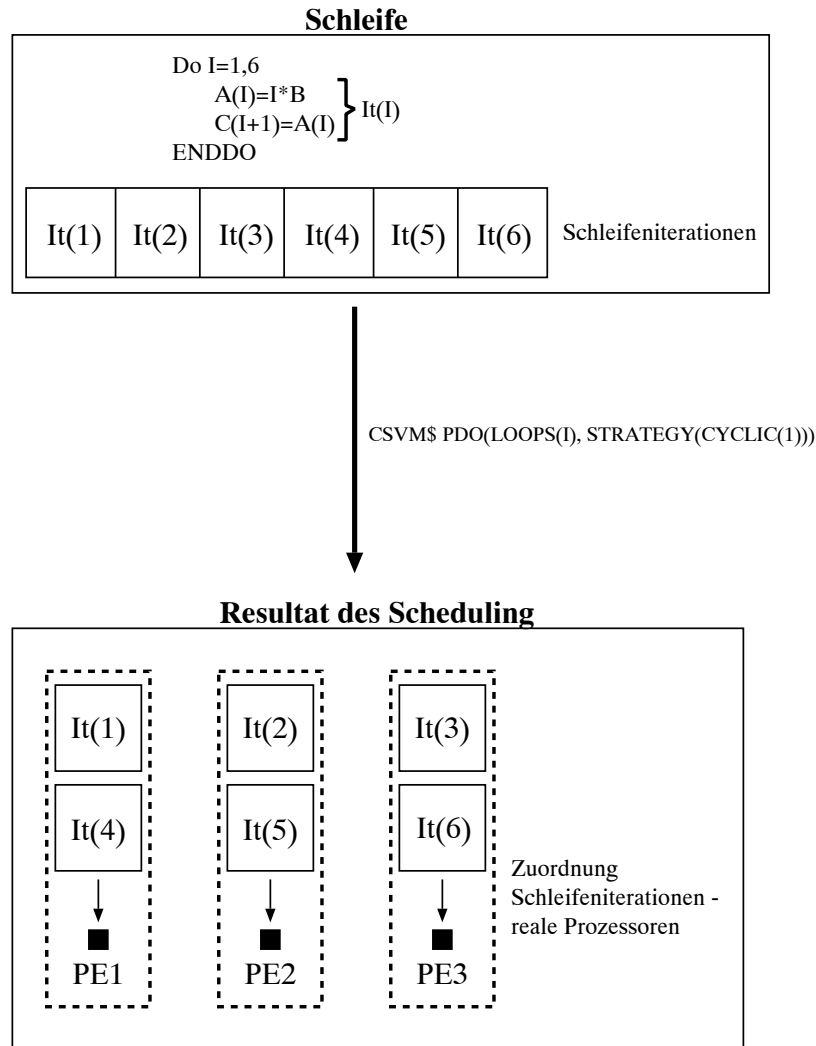


Abbildung 4.1: Direktes Schleifen-Scheduling

In Abbildung 4.1 kann man anhand eines Beispiels die Abbildung für eine zyklische Verteilung einer Schleife sehen. Die sechs Schleifeniterationen werden zyklisch auf die realen Prozessoren $PE = \{PE1, PE2, PE3\}$ verteilt, so daß Prozessor PE1 die Iterationen eins und vier, Prozessor PE2 die Iterationen zwei und fünf und Prozessor PE3 die Iterationen drei und sechs ausführt.

Beim Scheduling von mehreren perfekt geschachtelten Schleifen ist zu beachten,

¹Die Syntax dieser Strategien ist in der *strat-spec1-list* der PDO-Direktive festgehalten.

daß zwischen den Schleifen, die in der LOOPS-Option angegeben sind, und den Scheduling-Strategien, die in der STRATEGY-Option angegeben werden, eine Zuordnung stattfindet, so daß die Schleife, deren Laufvariable als erstes in der LOOPS-Option auftritt, mit der ersten Scheduling-Strategie in der STRATEGY-Option verteilt wird und so fort.

```

CSVM$ PDO( LOOPS(      I,      J,      K      ),
              STRATEGY( BLOCK, CYCLIC(1), BLOCK )
            )
DO I = 1, 1000
DO J = -100,100
DO K = 1, 20000
.....
ENDDO
ENDDO
ENDDO

```

Abbildung 4.2: Zuordnung Schleifenlaufvariable - Scheduling-Strategie

In Abbildung 4.2 wird die I-Schleife auf alle Prozessoren des APS, das zu Beginn dieser Schleife aktiv ist, blockverteilt. Die J-Schleife wird dann zyklisch und die K-Schleife wiederum blockweise auf die Prozessoren verteilt.

Ein Vorteil des direkten Schleifen-Scheduling ist, daß die Verteilung nicht abgespeichert wird. Dadurch wird im Hauptspeicher kein Platz belegt. Weiterhin ist die Strategie, nach der die Schleife verteilt werden soll, zur Übersetzungszeit bekannt, und es kann effizienter Code generiert werden.

Ein Nachteil dieses Scheduling ist, daß es nur lokal verwendet werden kann, d.h. bei jeder Schleife im Programm wird das Scheduling erneut durchgeführt. Falls eine Schleife mehrmals durchlaufen wird und die Schleifengrenzen bei jeder Schleifenausführung variieren, dann kann es vorkommen, daß Iterationen mit gleichem Speicherzugriffsmuster auf unterschiedlichen Prozessoren ausgeführt werden. Dies kann zur Folge haben, daß Daten, die in einer Iteration benötigt werden, von einem Prozessor zum anderen migrieren. Dadurch entstehen *page faults*, die man bei Wiederverwendung einer Verteilung vermeiden kann.

Da das direkte Schleifen-Scheduling nicht Gegenstand dieser Diplomarbeit ist, wird es in den folgenden Abschnitten nicht weiter behandelt werden.

4.2 Vordefiniertes Schleifen-Scheduling

Beim vordefinierten Schleifen-Scheduling liegt die Verteilung der Iterationen auf Prozessoren vor der Schleifenausführung fest. In der STRATEGY-Option der PDO-Direktive kann der Benutzer diese Art des Scheduling spezifizieren, indem er dort das ON-HOME-Konstrukt angibt. Diesem Konstrukt wird als Argument ein Template übergeben, mit dessen Hilfe das Scheduling durchgeführt werden soll. Zwischen einem Template und den Iterationen einer Schleife kann man eine Abbildung folgendermaßen herstellen: Zu jeder Iteration, die durch den Wert ihrer Laufvariablen bestimmt ist, muß genau ein Template-Element mit demselben Index existieren. Das Template abstrahiert von den Iterationen einer Schleife in dem Sinne, daß in dem Template nur die Zuordnung von einem Template-Element zu einem Prozessor in einer effizienten internen Repräsentation festgehalten wird. Bei der Schleifenausführung wird diese Information dann genutzt und die Iteration auf dem entsprechenden Prozessor abgearbeitet.

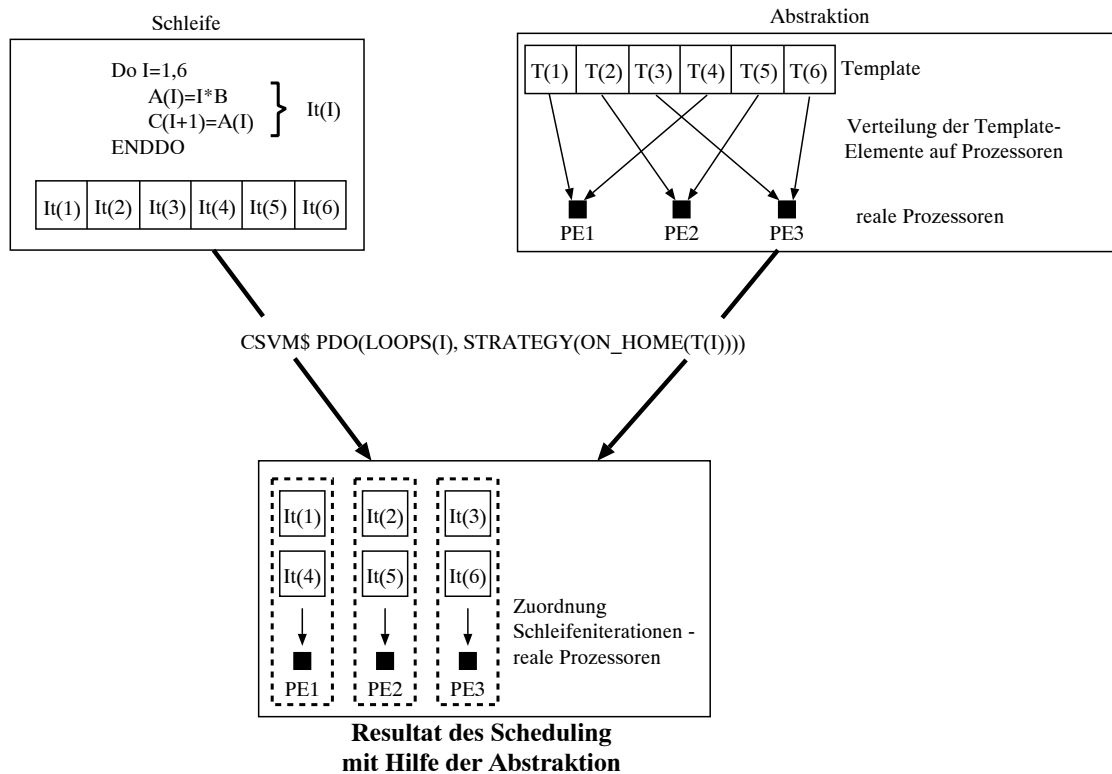


Abbildung 4.3: Vordefiniertes Schleifen-Scheduling

In Abbildung 4.3 wird die Vorgehensweise beim vordefinierten Scheduling verdeutlicht. Die Schleife mit sechs Iterationen soll mit dem ON-HOME-Konstrukt in der STRATEGY-Option der PDO-Direktive auf die drei Prozessoren $PE1$, $PE2$ und $PE3$ verteilt und ausgeführt werden. Dazu wird das Template T , dessen Elemente $T(I)$ mit $1 \leq I \leq 6$ bereits den Prozessoren über eine DISTRIBUTE- oder REDISTRIBUTE-Direktive (vgl. Abschnitt 4.2.3) zugeordnet worden sind, benutzt.

Template-Element T(1) wurde Prozessor PE1 zugeordnet, deshalb wird Iteration It(1) auf diesem ausgeführt, T(2) wurde PE2 zugeordnet, deshalb wird Iteration It(2) auf PE2 ausgeführt und so fort.

Das Scheduling von mehreren perfekt geschachtelten Schleifen geschieht hier über die Template-Dimensionen, d.h. jede Schleife, deren Schleifenlaufvariable in der LOOPS-Option spezifiziert ist, wird in der Weise verteilt, wie die Template-Dimension, in der dieselbe Schleifenlaufvariable angegeben wurde, verteilt worden ist.

```

CSVM$ PDO( LOOPS(
                                I,  J,  K ),
                                STRATEGY( ON_HOME( T( I, K, *, J ))
                                )
DO I = 1, 1000
    DO J = 1, 20000
        DO K = -100,100
            .....
        ENDDO
    ENDDO
ENDDO

```

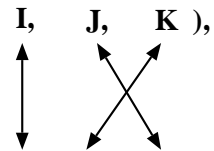


Abbildung 4.4: Scheduling mehrerer perfekt geschachtelter Schleifen über Templates

In Abbildung 4.4 wird die I-Schleife so verteilt, wie die erste Template-Dimension auf die Prozessoren verteilt worden ist, die J-Schleife so, wie die vierte Template-Dimension, und die K-Schleife so, wie die zweite Template-Dimension. Zusätzlich bearbeiten alle Prozessoren, auf die die dritte Template-Dimension abgebildet worden ist, das Schleifennest kooperativ.

Die Strategien, mit denen man die Schleifeniterationen den Prozessoren zuordnen kann, ergeben sich aus den Möglichkeiten, die man bei der Verteilung der Template-Elemente auf die Prozessoren in der DISTRIBUTE-Direktive hat (vgl. Abschnitt 4.2.3). Zusätzlich beeinflussen die Prozessorfeld- und Template-Deklaration indirekt auch die Scheduling-Strategien beim vordefinierten Schleifen-Scheduling. Daher werden diese drei Direktiven kurz vorgestellt. Abschließend werden die Vor- und Nachteile des vordefinierten Schleifen-Scheduling-Konzeptes diskutiert.

4.2.1 PROCESSORS-Direktive

Mit der PROCESSORS-Direktive, deren Syntax sich in Anhang A befindet, können Prozessorfelder deklariert werden. Der Gültigkeitsbereich eines Prozessorfeldes erstreckt sich über die Programmeinheit, in der die Deklaration ausgeführt worden ist. Bei der Deklaration, die nur im Deklarationsteil einer Programmeinheit verwendet werden darf, wird das Prozessorfeld auf das APS, das bei Ausführung der PROCESSORS-Direktive gültig ist, abgebildet. Falls mehrere Prozessorfelder in einer Programmeinheit deklariert werden, dann werden sie in der Reihenfolge ihres Auftretens auf das APS abgebildet. Bei der Deklaration wird der Rang des Feldes und die Ausdehnung in jeder Dimension definiert. Eine Option bei der Prozessorfelddeklaration ist das RESHAPE-Konstrukt. Mit ihm kann man auf ein *processor set*², auf das ein Prozessorfeld abgebildet worden ist, unterschiedliche geometrische Sichten erhalten. Ein neues Prozessorfeld, das im RESHAPE-Konstrukt spezifiziert wird, wird auf dasselbe *processor set* abgebildet. Deshalb muß die Anzahl der abstrakten Prozessoren, auf die beide Prozessorfelder abgebildet werden, gleich sein. Dennoch kann das RESHAPE-Prozessorfeld einen anderen Rang oder in den Dimensionen eine andere Ausdehnung haben.

An einem kleinen Beispiel sollen die oben erwähnten Konzepte erläutert werden:

```

PROGRAM EXAMPLE_PROCESSORS

C *** Prozessorfelddeklarationen
CSVM$ PROCESSORS:: Q(100)
CSVM$ PROCESSORS:: P1(16) RESHAPE:: P2(4,4), P3(2,2,4)

C *** Anweisungsteil
...

END

```

Zunächst wird ein Prozessorfeld Q mit 100 Prozessoren deklariert. Dieses Prozessorfeld wird auf die abstrakten Prozessoren des APS des Programms EXAMPLE_PROCESSORS von 0 bis 99 abgebildet.

Die nächste Prozessorfelddeklaration bewirkt, daß ein eindimensionales Prozessorfeld P1 mit 16 Elementen angelegt und auf die nächsten 16 abstrakten Prozessoren des APS, also von 100 bis 115, abgebildet wird. Weiterhin wird das zweidimensionale und dreidimensionale Prozessorfeld P2 bzw. P3 angelegt. Diese Felder werden mit der aus Fortran bekannten *column major order* auf dieselben Prozessoren des APS abgebildet wie P1. Es besteht also kein Unterschied, ob ein Ausführungsblock beispielsweise dem Prozessorfeldelement P1(3), P2(3,1) oder P3(1,2,1) zugewiesen wird, denn alle drei Prozessorfeldelemente werden auf den abstrakten Prozessor 102 im APS abgebildet, d.h. derselbe reale Prozessor würde diesen Block ausführen.

²Dies ist eine Anzahl abstrakter Prozessoren (siehe Abschnitt 2.3).

Ein Prozessorfeld kann an eine Funktion oder ein Unterprogramm als Parameter weitergegeben werden. Das aktuelle Prozessorfeldargument muß bzgl. Rang und Ausdehnung in jeder Dimension mit der Prozessorfelddeklaration des formalen Prozessorfeldargumentes übereinstimmen. Mit dem formalen Prozessorfeldargument dürfen nur *Barrier* im Gültigkeitsbereich des Prozessorfeldes ausgeführt werden.

4.2.2 TEMPLATE-Direktive

Bei der Template-Deklaration, die mit der TEMPLATE-Direktive durchgeführt wird und nur im Deklarationsteil einer Programmeinheit verwendet werden darf, bestimmt man den Rang eines Templates. Neben der festen Bestimmung der Ausdehnung in jeder Dimension besteht die Möglichkeit, Platzhalter einzusetzen, für die später mit der CREATE-Direktive im Anweisungsteil feste Werte eingesetzt werden können. Dadurch hat man die Möglichkeit, die Ausdehnung eines Templates von Berechnungsergebnissen abhängig zu definieren. Wenn beispielsweise die Grenzen einer Schleife von Berechnungen im Programm abhängen und diese Schleife mit einem Template verteilt werden soll, dann kann man ein Template mit der entsprechenden Ausdehnung definieren.

```

PROGRAM EXAMPLE_TEMPLATE
PARAMETER (N=1024)
INTEGER EXTENSION

C *** Template-Deklaration
CSVM$ TEMPLATE:: T1(-N:N), T2(:, :)

C *** Anweisungsteil (unter anderem wird die Variable EXTENSION berechnet)
...
EXTENSION = ...
...

C *** Definition der Ausdehnung des Templates T2 in jeder Dimension
CSVM$ CREATE:: T2(EXTENSION, 2*EXTENSION)

.....

END

```

Im Programm EXAMPLE_TEMPLATE wird das eindimensionale Template T1 mit einer Ausdehnung von -1024 bis 1024 und das zweidimensionale Template T2 ohne Definition der Ausdehnung in den zwei Dimensionen deklariert. Die Ausdehnung von T2 wird in jeder Dimension abhängig von der Variablen EXTENSION, die im Anweisungsteil berechnet wird, in der CREATE-Direktive festgelegt. Nach der Ausführung der Direktive hat T2 in der ersten Dimension die Ausdehnung von 1 bis EXTENSION und in der zweiten Dimension von 1 bis 2·EXTENSION.

Ein Template kann auch an ein Unterprogramm über die Parameterliste weitergegeben werden. Das aktuelle Template muß mit der Deklaration des formalen Templates bzgl. Rang und Ausdehnung in jeder Dimension übereinstimmen. Das formale Argument kann zum Scheduling von Schleifen benutzt werden, oder es wird auf ein Prozessorfeld verteilt. Diese neue Verteilung wird an das aktuelle Argument automatisch zurückgegeben, was dem *Call-By-Reference*-Übergabemechanismus entspricht.

4.2.3 DISTRIBUTE- und REDISTRIBUTE-Direktive

Die Verteilung eines Templates auf ein Prozessorfeld wird in der DISTRIBUTE- bzw. REDISTRIBUTE-Direktive realisiert. Der Unterschied zwischen diesen Direktiven besteht darin, daß die REDISTRIBUTE-Direktive ein bereits verteiltes Template wieder umverteilen kann und daß diese Direktive nur im Anweisungsteil verwendet werden darf, wohingegen die DISTRIBUTE-Direktive nur im Deklarationsteil einer Programmeinheit auftreten darf. Wenn im folgenden nur noch die DISTRIBUTE-Direktive erwähnt wird, dann gelten dieselben Aussagen auch für die REDISTRIBUTE-Direktive.

Eine Restriktion bei der Anwendung dieser Direktive ist, daß ein Template auf ein Prozessorfeld nur dann verteilt werden darf, wenn das APS der Template-Deklarationsumgebung mit dem APS übereinstimmt, das bei der Direktive aktiv ist.

Die Syntax der Direktive mit den für diesen Abschnitt wichtigen Teilen wird zunächst abgebildet und danach die Semantik erläutert. Das vollständige Syntaxdiagramm kann aus Anhang A entnommen werden.

Syntax:

<i>distribute-directive</i>	is	DISTRIBUTE <i>dist-strategy</i> :: <i>template-name-list</i>
<i>redistribute-directive</i>	is	REDISTRIBUTE <i>dist-strategy</i> :: <i>template-name-list</i>
<i>dist-strategy</i>	is	<i>standard-distribution</i>
<i>standard-distribution</i>	is	(<i>dist-format-list</i>) ONTO <i>processor-section</i>
<i>dist-format</i>	is	*
	or	BLOCK
	or	GENERAL_BLOCK (<i>int-array</i>)
	or	CYCLIC [(<i>int-expr</i>)]
<i>processor-section</i>	is	<i>processors-name</i>
	or	<i>processors-name</i> (<i>dist-spec-list</i>)
<i>dist-spec</i>	is	[<i>lower-bound</i>] : [<i>upper-bound</i>]
	or	<i>int-expr</i>
	or	*

In welcher Weise ein Template auf ein Prozessorfeld verteilt werden kann, hängt hauptsächlich von den Verteilungsarten der Template-Dimension und von den zur Verfügung stehenden Prozessoren ab. Dies wird in zwei Listen, der *dist-format-list* und der *dist-spec-list*, die in jeder Direktive definiert werden müssen, festgehalten.

Die *dist-format-list* gibt an, wie ein Template dimensionsweise verteilt wird, und die *dist-spec-list* beinhaltet die Informationen, auf welche Prozessoren ein Template verteilt wird. Diese zwei Listen und die Abhängigkeiten zwischen ihnen werden nachfolgend beschrieben.

Bei der Verteilung einer Template-Dimension gibt es vier verschiedene Möglichkeiten:

1. **Sequentialisierung:**

Hierbei wird eine Template-Dimension nicht auf eine Prozessorfelddimension verteilt, sondern alle Prozessoren erhalten alle Template-Elemente in dieser Dimension. Im Syntaxdiagramm ist dies durch einen Stern gekennzeichnet.

2. **BLOCK-Verteilung:**

Hierbei wird eine Template-Dimension mit der HPF-Blockverteilung auf die Prozessoren einer Prozessorfelddimension verteilt.

3. **GENERAL-BLOCK-Verteilung:**

Diese Verteilungsart funktioniert ähnlich wie die BLOCK-Verteilung, außer daß hier die Blockgröße, die einem Prozessor zugewiesen wird, in einem Feld angegeben wird. Die Anzahl der Feldelemente muß gleich der Anzahl der Prozessoren minus eins sein. Falls eine Template-Dimension auf 10 Prozessoren verteilt werden soll, dann muß das Feld neun Feldelemente haben. Dem ersten Prozessor wird die Anzahl von Template-Elementen zugewiesen, die im Feld mit Index 1 spezifiziert ist, dem zweiten die, die im Feld mit Index 2 spezifiziert ist und so fort. Dem letzten, dem zehnten Prozessor, werden die restlichen Elemente zugewiesen. Weiterhin darf die Summe der einzelnen Feldelemente nicht größer sein als die Anzahl der Template-Elemente in dieser Dimension.

4. **CYCLIC-Verteilung:** Die Template-Dimension wird zyklisch auf die Prozessoren einer Prozessorfelddimension verteilt.

Diese vier Möglichkeiten können in der *dist-format-list* spezifiziert werden, um zu bestimmen, wie eine Template-Dimension verteilt werden kann. Die Anzahl der Einträge in dieser Liste muß folglich mit dem Rang des Templates übereinstimmen. Der erste Eintrag gibt dann an, wie die erste Template-Dimension verteilt werden soll, der zweite wie die zweite Template-Dimension usw.

Bisher wurde beschrieben, wie man eine Template-Dimension verteilen kann. Nun soll erläutert werden, auf welches Prozessorfeld man das Template verteilen kann und welche Möglichkeiten dabei bestehen. In der *dist-spec-list* wird pro Dimension des Prozessorfeldes, auf das ein Template verteilt werden soll, spezifiziert, ob Teile des Templates auf die Prozessoren in dieser Dimension verteilt werden sollen oder nicht. Es gibt dabei insgesamt drei Möglichkeiten:

1. Sektionen:

Es wird eine Template-Dimension, die mit einer BLOCK-, CYCLIC- oder GENERAL-BLOCK-Verteilung verteilt werden soll, auf eine Prozessorfelddimension verteilt, die eine Sektion spezifiziert. Dabei erhalten nicht alle Prozessoren Teile des Templates, sondern nur die, die durch die Sektion bestimmt sind. Eine Prozessorfelddimension bestehe beispielsweise aus 10 Prozessoren, von 1 bis 10 durchnummeriert. Wenn in einer Sektion (3 : 9) spezifiziert wird, dann erhalten nur die Prozessoren von 3 bis 9 Teile des Templates:

```

PROGRAM EXAMPLE_DISTRIBUTE1

C *** Prozessorfelddeklaration
CSVMS$ PROCESSORS:: P(10)

C *** Template-Deklaration
CSVMS$ TEMPLATE:: T(70)

C *** Template T wird auf Prozessorfeld P verteilt
CSVMS$ DISTRIBUTE (BLOCK) ONTO P(3:9) :: T

    ....

END

```

Die Template-Elemente werden, wie in der folgenden Tabelle dargestellt, auf die Prozessoren verteilt:

Prozessor	Elemente
P(1)	–
P(2)	–
P(3)	$T(1), \dots, T(10)$
P(4)	$T(11), \dots, T(20)$
P(5)	$T(21), \dots, T(30)$
P(6)	$T(31), \dots, T(40)$
P(7)	$T(41), \dots, T(50)$
P(8)	$T(51), \dots, T(60)$
P(9)	$T(61), \dots, T(70)$
P(10)	–

2. Integer-Ausdrücke:

Das gesamte Template, nicht eine Template-Dimension, wird nur auf dem Prozessor, der durch den *Integer*-Ausdruck in dieser Dimension spezifiziert ist, verteilt. Die anderen Prozessoren dieser Dimension erhalten keine Template-Elemente.

```

PROGRAM EXAMPLE_DISTRIBUTE2

C *** Prozessorfelddeklaration
CSVM$ PROCESSORS:: P(5,2)

C *** Template-Deklaration
CSVM$ TEMPLATE:: T(40)

C *** Template T wird auf Prozessorfeld P verteilt
CSVM$ DISTRIBUTE (BLOCK) ONTO P(2:5,2) :: T

....

END

```

Nachfolgend ist die resultierende Verteilung der Template-Elemente auf das Prozessorfeld P, das in der zweiten Dimension einen *Integer*-Ausdruck enthält, abgebildet. Das zweidimensionale Prozessorfeld ist als Matrix in der Tabelle dargestellt.

	P(.,1)	P(.,2)
P(1,.)	–	–
P(2,.)	–	$T(1), \dots, T(10)$
P(3,.)	–	$T(11), \dots, T(20)$
P(4,.)	–	$T(21), \dots, T(30)$
P(5,.)	–	$T(31), \dots, T(40)$

3. Replikation:

Das gesamte Template wird auf alle Prozessoren in dieser Dimension redundant verteilt. Diese Dimension wird dann durch den Stern gekennzeichnet.

```

PROGRAM EXAMPLE_DISTRIBUTE3

C *** Prozessorfelddeklaration
CSVM$ PROCESSORS:: P(4,3)

C *** Template-Deklaration
CSVM$ TEMPLATE:: T(45)

C *** Template T wird auf Prozessorfeld P verteilt
CSVM$ DISTRIBUTE (BLOCK) ONTO P(*,1:3) :: T

....

END

```

In der Tabelle ist die Verteilung auf das Prozessorfeld P, das in der ersten Dimension repliziert wird, festgehalten:

	P(.,1)	P(.,2)	P(.,3)
P(1,.)	$T(1), \dots, T(15)$	$T(16), \dots, T(30)$	$T(31), \dots, T(45)$
P(2,.)	$T(1), \dots, T(15)$	$T(16), \dots, T(30)$	$T(31), \dots, T(45)$
P(3,.)	$T(1), \dots, T(15)$	$T(16), \dots, T(30)$	$T(31), \dots, T(45)$
P(4,.)	$T(1), \dots, T(15)$	$T(16), \dots, T(30)$	$T(31), \dots, T(45)$

Mit diesen drei Möglichkeiten kann der Benutzer spezifizieren, auf welche Prozessoren eines Prozessorfeldes ein Template verteilt werden soll. Daher muß die Anzahl der Listeneinträge in der *dist-spec-list* gleich dem Rang des Prozessorfeldes sein. Zur Vereinfachung braucht der Benutzer beim Prozessorfeld die *dist-spec-list*, die pro Dimension angibt, auf welche Prozessoren ein Template verteilt werden soll, nicht anzugeben. Der Compiler interpretiert dies dann so, als ob der Benutzer in jeder Dimension eine Sektion spezifiziert hat, deren Grenzen die Prozessorfeldgrenzen in dieser Dimension sind. In dem nachfolgend beschriebenen Programm EXAMPLE_DISTRIBUTE4 erhält jeder der 100 Prozessoren vom Prozessorfeld P 100 Template-Elemente.

```

PROGRAM EXAMPLE_DISTRIBUTE4

C *** Prozessorfelddeklaration
CSVM$ PROCESSORS:: P(100)

C *** Template-Deklaration
CSVM$ TEMPLATE:: T(10000)

C *** Template T wird auf Prozessorfeld P verteilt
CSVM$ DISTRIBUTE (BLOCK) ONTO P :: T

....

END

```

Zwischen der *dist-spec-list* und der *dist-format-list*, die angibt, wie ein Template dimensionsweise verteilt wird, läßt sich ein Zusammenhang herstellen. Die Anzahl der Template-Dimensionen, die mit einer BLOCK-, GENERAL-BLOCK- oder CYCLIC-Verteilung verteilt werden, muß gleich der Anzahl der Sektionen in dem Prozessorfeld sein, auf das das Template verteilt werden soll. Die Zuordnung der Template-Dimensionen auf die Prozessorfelddimensionen findet folgendermaßen statt: Die erste Template-Dimension, die eine der drei oben genannten Verteilungsschemata hat, wird auf die erste Dimension mit einer Sektionsspezifikation abgebildet, die zweite Template-Dimension, die keine Sequentialisierung spezifiziert, wird auf die zweite Dimension mit einer Sektionsspezifikation abgebildet usw.

```

PROGRAM EXAMPLE_DISTRIBUTE5

C *** Prozessorfelddeklaration
CSVM$ PROCESSORS:: P(4,2,5,10,3)

C *** Template-Deklaration
CSVM$ TEMPLATE:: T(100,200,300,400)

C *** Template T wird auf Prozessorfeld P verteilt
CSVM$ DISTRIBUTE (BLOCK,          CYCLIC(1), * , BLOCK      )
CSVM$+      ONTO P( 1:4 , * , 1:5 , 1:10, 1) :: T

      ....

END

```

Im Programm EXAMPLE_DISTRIBUTE5 wird ein vierdimensionales Template auf ein fünfdimensionales Prozessorfeld abgebildet. Die erste Template-Dimension wird blockweise auf die erste Prozessorfelddimension und die zweite Template-Dimension zyklisch auf die dritte Prozessorfelddimension verteilt. Die zweite Prozessorfelddimension wird repliziert, d.h. das Template wird auf den Prozessoren in dieser Dimension redundant gespeichert. Die dritte Template-Dimension wird sequentialisiert, jeder Prozessor erhält alle Template-Elemente in dieser Dimension, und die vierte Dimension wird blockweise der vierten Prozessorfelddimension zugeordnet. In der fünften Prozessorfelddimension wird nur dem Prozessor 1 das Template zugewiesen.

4.2.4 Bewertung des vordefinierten Schleifen-Scheduling

Im folgenden werden kurz die Vor- und Nachteile des vordefinierten Schleifen-Scheduling vorgestellt. Ein Vorteil ist, daß die Verteilung im Template abgespeichert wird. Dies hat verschiedene Konsequenzen:

- Dieselbe Verteilung kann im Gegensatz zum direkten Scheduling für mehrere Schleifen verwendet werden. Falls die Iterationen der unterschiedlichen Schleifen dasselbe Speicherzugriffsmuster haben, so hat dies eine größere Lokalität der Daten zur Folge, da Daten sich bereits im lokalen Speicher des Prozessors befinden.

Als Beispiel sei hier ein Programm mit zwei Schleifen aufgeführt. Zum einen werden die Schleifeniterationen mit dem direkten Scheduling und zum anderen mit dem vordefinierten Scheduling auf die Prozessoren verteilt. Anschließend wird das Speicherzugriffsmuster der Schleifen für beide Verfahren für den Fall vorgestellt, daß das Programm mit vier Prozessoren gestartet wird.

```

PROGRAM DIRECT_SCHEDULING
REAL *8 A(4096)

....

CSVM$ PDO(LOOPS(I),
          STRATEGY(BLOCK))
DO I = 1, 4096
  A(I) = ...
ENDDO

....

CSVM$ PDO(LOOPS(J),
          STRATEGY(BLOCK))
DO J = 2, 4096
  A(J) = ...
ENDDO

....

END

```

```

PROGRAM PREDEFINED_SCHEDULING
REAL *8 A(4096)
CSVM$ PROCESSORS::P(4)
CSVM$ TEMPLATE::T(4096)
CSVM$ DISTRIBUTE (BLOCK) ONTO P :: T

....

CSVM$ PDO(LOOPS(I),
          STRATEGY(ON_HOME(T(I))))
DO I = 1, 4096
  A(I) = ...
ENDDO

....

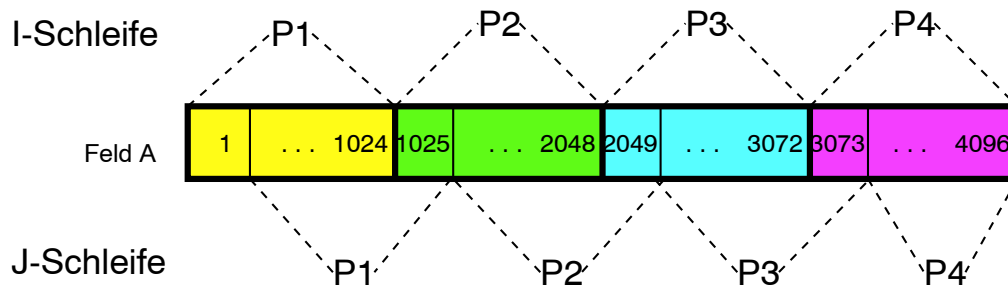
CSVM$ PDO(LOOPS(J),
          STRATEGY(ON_HOME(T(J))))
DO J = 2, 4096
  A(J) = ...
ENDDO

....

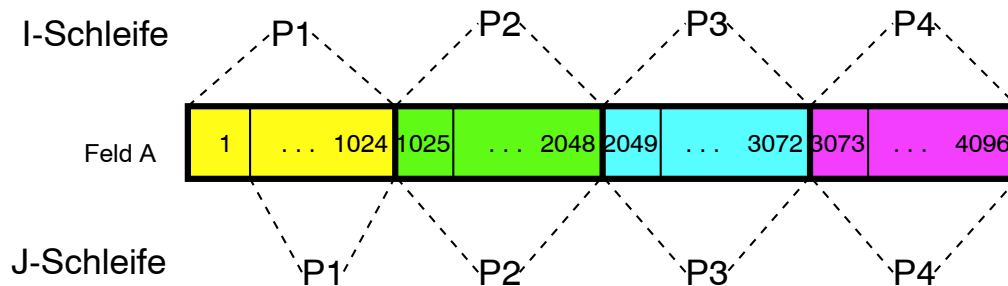
END

```

Das Speicherzugriffsverhalten sieht im Programm DIRECT_SCHEDULING für das Feld A folgendermaßen aus:



Das Speicherzugriffsverhalten im Programm PREDEFINED_SCHEDULING sieht demgegenüber für das Feld A wie folgt aus:



Eine Seite habe die Größe 8 KB. Dies hat zur Folge, daß 1024 Feldelemente des Feldes A auf eine Seite gespeichert werden. Demnach greift beim direkten Scheduling jeder Prozessor in der J-Schleife auf zwei Seiten zu, was einen konkurrierenden Zugriff von zwei Prozessoren auf jede Seite zur Folge hat. Beim vordefinierten Scheduling hingegen greift jeder Prozessor sowohl in der I- als auch in der J-Schleife auf dieselbe Seite zu, was eine größere Lokalität und weniger *page faults* mit sich bringt.

- Eine globale Steuerung des Scheduling kann auch über Unterprogrammgrenzen hinweg stattfinden, da Templates als Parameter an Funktionen übergeben werden können.
- Mit Hilfe der REDISTRIBUTE-Direktive kann ein Template zur Laufzeit umverteilt werden. Dadurch kann eine bessere Lastbalance zwischen Prozessoren erreicht werden.
- Berechnungen für die Schleifenverteilung können unabhängig von der Schleife berechnet werden. Dadurch müssen diese im Gegensatz zum direkten Scheduling nur einmal durchgeführt werden. Daraus folgt, daß ein Großteil des *Overheads*, der bei der Berechnung der Schleifenverteilung anfällt, beim Schleifen-Scheduling entfällt, dafür aber während der DISTRIBUTE-Direktive durchgeführt werden muß.

Ein Nachteil des vordefinierten Scheduling ist, daß das Abspeichern der Templates und der Verteilung eines Templates Platz im Hauptspeicher belegt. Außerdem steht zur Übersetzungszeit in der Regel nicht fest, mit welcher Verteilungsstrategie die Schleifeniterationen mit Hilfe des vordefinierten Scheduling auf die Prozessoren verteilt werden. Dadurch kann kein für die Strategie speziell optimierter Code generiert werden, sondern es muß Code erzeugt werden, der, über einen Parameter gesteuert, alle Strategien ausführen kann (siehe Abschnitt 5.6.1).

4.3 Dynamisches Schleifen-Scheduling

Beim dynamischen Schleifen-Scheduling werden die Iterationen während der Schleifenausführung den Prozessoren zugeordnet. Der Benutzer kann zwischen mehreren Strategien wie z.B. *Self Scheduling*, *Guided Self Scheduling* oder *Factoring* wählen³. Jeder Prozessor, der die parallele Schleife ausführt, fordert neue Iterationen an, sobald er nicht mehr ausgelastet ist. Im Gegensatz zu statischen Verfahren wie die BLOCK-, GENERAL-BLOCK- oder CYCLIC-Verteilungsstrategie gewährleisten diese Verfahren eine gute Lastbalance, auch wenn die Varianz der Ausführungszeiten der einzelnen Iterationen groß ist. Das dynamische Scheduling kann in Verbindung mit dem direkten oder vordefinierten Scheduling eingesetzt werden.

³Diese Strategien werden im Detail in [50] erklärt

Beim direkten Scheduling kann eine Schleife mit einer dynamischen Strategie direkt auf die Prozessoren verteilt werden. Die Effizienz dieses Verfahrens muß noch untersucht werden, da bei den dynamischen Scheduling-Verfahren ein nicht unerheblicher *Overhead* durch das dynamische Verteilen der Iterationen anfällt.

Um die Verteilung einer Schleife, deren Iterationen mit einer dynamischen Strategie auf die Prozessoren verteilt worden sind, wiederverwenden zu können, kann das dynamische Schleifen-Scheduling in Verbindung mit dem vordefinierten Scheduling verwendet werden. Während des dynamischen Scheduling wird die Verteilung in einem Template festgehalten. Bei ähnlichen Schleifen kann man diese Verteilung im Template dann nutzen und diese Schleifen nach gleichem Muster verteilen. Die Vorteile bestehen in einer guten Lastbalance auf allen Prozessoren und bei Schleifen mit ähnlichem Speicherzugriffsmuster in einer Reduzierung von nichtlokalen Datenzugriffen. Dies kann evtl. dazu führen, daß der *Overhead*, der beim dynamischen Verteilen der Schleifeniterationen anfällt, in den nachfolgenden Schleifen kompensiert wird, was in [50] untersucht wird.

Um die dynamischen Scheduling-Verfahren mit dem Template in SVM-Fortran anwenden zu können, existiert das DISTRIBUTE-ONCE-Konstrukt. Dieses Konstrukt kann in der STRATEGY-Option der PDO-Direktive angegeben werden.

Syntax:

```

strat-spec    is  strat-spec1-list
                or  ON_HOME ( template-name ( templ-spec-list ) )
                or  DISTRIBUTE_ONCE ( template-name
                                     ( templ-spec-list ), ( strat-spec1-list ) )

strat-spec1  is  *
                or  BLOCK
                or  CYCLIC [ ( int-expr ) ]
                or  GRAB [ ( int-const ) ]

```

Die Semantik dieses Konstruktes ist, daß ein Template, das diesem Konstrukt übergeben wird und nicht auf ein Prozessorfeld verteilt worden ist, mit den in *strat-spec1-list*⁴ angegebenen Strategien verteilt wird. In der PROCESSORS-Option der PDO-Direktive kann spezifiziert werden, auf welches Prozessorfeld das Template verteilt wird. Die Verteilung wird dann so, wie bei der DISTRIBUTE-Direktive beschrieben, durchgeführt. Das Scheduling dieser Schleife findet danach genauso statt wie beim vordefinierten Schleifen-Scheduling mit dem ON-HOME-Konstrukt. Falls das Template jedoch mit einer dynamischen Strategie (*grab*) in einer Dimension verteilt werden soll, dann wird die Verteilung während der Schleifenausführung im Template abgespeichert, wie oben beschrieben. Nach Ausführung des DISTRIBUTE-ONCE-Konstruktes ist das Template in allen Dimensionen verteilt.

Ist das Template jedoch vor der parallelen Schleife, die mit dem DISTRIBUTE-ONCE-Konstrukt in der STRATEGY-Option der PDO-Direktive verteilt werden

⁴Die Semantik der regelmäßigen Verteilungsstrategien wurde bereits in Abschnitt 4.1 erklärt. Mit der GRAB-Option kann eine dynamische Scheduling-Strategie spezifiziert werden. Der Parameter, der dem GRAB-Konstrukt übergeben wird, definiert die Scheduling-Strategie.

soll, schon auf ein Prozessorfeld verteilt worden, dann wird diese Schleife genauso verteilt wie mit dem ON-HOME-Konstrukt des vordefinierten Schleifen-Scheduling.

```

PROGRAM EXAMPLE_DYNAMIC_SCHEDULING
  INTEGER I,J

C *** Speichere die Variablen I und J im lokalen Speicher eines jeden Prozessors
CSVM$ PRIVATE:: I, J

C *** Prozessorfelddeklaration
CSVM$ PROCESSORS:: P(10,10)

C *** Template-Deklaration
CSVM$ TEMPLATE:: T(1000, 2000)

C *** ..... (Anweisungen)

C *** Verteile eine Schleife mit dem DISTRIBUTE_ONCE-Konstrukt
CSVM$ PDO(LOOPS(I,J),
CSVM$+   STRATEGY(DISTRIBUTE_ONCE(T(I,J), (BLOCK, GRAB(1)))),
CSVM$+   PROCESSORS(P(1:10,1:10))
CSVM$+ )
      DO I = 1, 1000
        DO J = 1, 2000

            ....

        ENDDO
      ENDDO

C *** ..... (Anweisungen)

C *** Verteile eine zweite Schleife mit dem DISTRIBUTE_ONCE-Konstrukt
CSVM$ PDO(LOOPS(I,J),
CSVM$+   STRATEGY(DISTRIBUTE_ONCE(T(I,J), (GRAB(3), CYCLIC(1)))),
CSVM$+   PROCESSORS(P(1:10,1:10))
CSVM$+ )
      DO I = 1, 1000
        DO J = 1, 2000

            ....

        ENDDO
      ENDDO

C *** ..... (Anweisungen)

END

```

Im Programm `EXAMPLE_DYNAMIC_SCHEDULING` wird ein zweidimensionales Prozessorfeld `P` und Template `T` deklariert. Danach wird ein aus zwei Schleifen bestehendes paralleles Schleifennest mit dem `DISTRIBUTE-ONCE`-Konstrukt in der `PDO`-Direktive ausgeführt. Da `T` nicht auf ein Prozessorfeld verteilt worden ist, wird die erste Dimension blockweise auf die erste Prozessorfelddimension abgebildet und die Iterationen der I-Schleife gemäß dieser Template-Dimension auf die Prozessoren verteilt. Da in der zweiten Dimension eine dynamische Verteilungsstrategie angegeben ist, wird die J-Schleife der Strategie entsprechend während der Schleifenausführung verteilt und diese Verteilung im Template abgespeichert.

Bei der zweiten Ausführung eines parallelen Schleifennestes mit Hilfe des Templates `T` ist das Template bereits auf ein Prozessorfeld verteilt worden. Deshalb wird die vordefinierte Verteilung genommen und die I-Schleife blockweise und die J-Schleife mit der dynamischen Strategie auf die Prozessoren verteilt. Die Angaben in der *strat-spec1-list* (`GRAB(3)`, `CYCLIC(1)`) werden ignoriert. Wäre das Template jedoch nicht auf ein Prozessorfeld verteilt worden, so würden die Scheduling-Strategien entsprechend angewandt und im Template abgespeichert.

Kapitel 5

Realisierung der Scheduling-Direktiven auf Parallelrechnern

Nach einer einführenden Übersicht über die allgemeine Umsetzung von SVM-Fortran-Direktiven wird beschrieben, wie die Administration der Prozessorfelder und Templates realisiert wird. Im Anschluß daran werden die einzelnen Direktiven und ihre Umsetzung detailliert erläutert.

Prinzipiell werden die Direktiven durch Wertzuweisungen in Fortran und Aufrufe an Unterprogramme der Laufzeitbibliothek ersetzt. Die Wertzuweisungen vor einem Unterprogrammaufruf sind in Fortran notwendig, da Ausdrücke, die Variablen enthalten wie $(N/2)$, erst zur Laufzeit ausgewertet werden können, bevor der Wert des Ausdrucks dann an das Unterprogramm übergeben werden kann. Die eigentliche Arbeit bei der Umsetzung der Direktiven geschieht in den Unterprogrammen der Laufzeitbibliothek. Die PROCESSORS-Direktive

```
C SVM$ PROCESSORS:: P(1:(N/2))
```

wird etwa substituiert durch

```
C *** Deklaration der neu verwendeten Variablen im Deklarationsteil
C *** der Programmeinheit, wo die Prozessorfelddeklaration ausgeführt wird
      INTEGER SVM_FIELD(7,3), SVM_FIRST_PROC_IN_APS, SVM_LAST_PROC_IN_APS

C *** generierter Code fuer die Umsetzung der Direktive
      SVM_FIRST_PROC_IN_APS = -1
      SVM_LAST_PROC_IN_APS = -1
      SVM_FIELD(1,1) = 1
      SVM_FIELD(1,2) = 1
      SVM_FIELD(1,3) = (N/2)
C *** Aufruf eines Unterprogramms der Laufzeitbibliothek
      CALL SVM_PROC_FILL(1,SVM_FIRST_PROC_IN_APS,SVM_LAST_PROC_IN_APS,
                        1,SVM_FIELD)
```

An diesem Beispiel sieht man exemplarisch, wie eine Prozessorfelddeklaration im Fortran-Programm umgesetzt wird. Zuerst müssen die neuen Variablen, die bei der Umsetzung der Direktive benötigt werden, in der Programmeinheit deklariert werden. Dann wird die Direktive durch Wertzuweisungen und einen Unterprogrammaufruf ersetzt. Die Bedeutung der einzelnen Variablen und die Funktionsweise des aufgerufenen Unterprogramms `SVM_PROC_FILL` werden bei der `PROCESSORS`-Direktive in Abschnitt 5.2 erklärt. An dieser Stelle soll dem Leser lediglich ein intuitives Verständnis vermittelt werden, wie die Umsetzung der Direktiven, die ähnlich wie bei der `PROCESSORS`-Direktive durchgeführt wird, prinzipiell geschieht.

5.1 Verteilte Datenstrukturen zur Administration der Prozessorfelder und Templates

Die Datenstrukturen, in denen die Informationen zu den Prozessorfeldern und Templates abgespeichert werden, werden auf den Prozessoren, die zum *initial processor set* gehören, vor Programmausführung im privaten Speicher eines jeden Prozessors allokiert. Im folgenden wird die Datenstruktur, die die Prozessorfeldinformationen enthält, als **Prozessorfeldtabelle** und die Datenstruktur, die die Template-Informationen beinhaltet, als **Template-Tabelle** bezeichnet. Die Größe der Tabellen, das ist die Anzahl der Einträge, richtet sich nach den im Programm vorkommenden Prozessorfeld- bzw. Template-Deklarationen. Sie ist gleich der Anzahl der im gesamten Programm vorkommenden Deklarationen und verändert sich während der Programmausführung nicht. Ein Eintrag in diesen Tabellen enthält alle Informationen zu einem Prozessorfeld bzw. Template. Zu jedem Prozessorfeld bzw. Template existiert ein Index, der auch als **Prozessorfeldidentifikator** bzw. **Template-Identifikator** bezeichnet wird, der die Position in der Tabelle angibt, wo sich der Eintrag mit den entsprechenden Informationen zum Prozessorfeld bzw. Template befindet. Die Vergabe dieser Identifikatoren findet zur Übersetzungszeit statt und ändert sich danach nicht mehr.

Zunächst werden die Vor- und Nachteile des vorgestellten Konzeptes zur Verwaltung von Templates und Prozessorfelder skizziert und mit einer möglichen Alternative verglichen. Anschließend werden die Informationen, die zu einem Prozessorfeld bzw. Template abgespeichert werden, detailliert vorgestellt.

Ein Vorteil der verteilten Datenhaltung ist, daß sich die Tabellen im privaten Speicher eines jeden Prozessors befinden. Dadurch ist ein schneller Zugriff auf die Daten möglich.

Ein weiterer Vorteil bei der im privaten Speicher abgelegten Template-Tabelle ist, daß bei den verschiedenen Verteilungsstrategien jeder Prozessor in einer effizienten Repräsentation in der Template-Tabelle zusätzlich speichert, welche Template-Elemente ihm mittels einer `DISTRIBUTE`-Direktive zugewiesen worden sind. Würde diese Tabelle hingegen nur einmal gespeichert, dann kann dies in dieser Form nicht

so einfach geschehen. Ein anderer Vorteil ist, daß die Tabellen nur einmal vor Programmstart allokiert und nach Beendigung des Programms freigegeben werden. Sie werden nicht bei jeder Deklaration erneut allokiert, und der Speicherplatz wird nicht am Ende des Gültigkeitsbereichs freigegeben, was eine längere Laufzeit zur Folge hätte.

Ein Nachteil ist, daß die Informationen zu einem Prozessorfeld beispielsweise redundant auf den verschiedenen Prozessoren gespeichert werden. Dadurch wird mehr Speicherplatz benötigt, als wenn die Informationen nur einmal abgespeichert würden.

Ein anderes Konzept wäre, die Daten zu den Prozessorfeldern und Templates im gemeinsamen Speicher abzulegen. Dies hätte den Vorteil, daß die Daten zu einem Prozessorfeld bzw. Template nur einmal abgespeichert würden. Dadurch würde weniger Speicherplatz verbraucht als bei der realisierten Variante. Ein Nachteil dieses Konzeptes ist aber, daß der Zugriff auf die Daten die Laufzeit ungünstig beeinflusst, weil die Daten im gemeinsamen Speicher sind und mehrere Prozessoren auf diese zugreifen müssen, wodurch Seitenfehler gehäuft auftreten würden. Hinzu kommt noch, daß Kommunikation stattfinden muß, da den Prozessoren im APS, die eine Deklaration ausführen, mitgeteilt werden muß, wo sich die Information zu der Deklaration im gemeinsamen Speicher befindet.

Da die Nachteile bei der Speicherung der Prozessorfeld- und Template-Informationen im gemeinsamen Speicher überwiegen und da der Speicherplatzbedarf der Prozessorfeld- und Template-Tabelle nicht so groß ist, wurde das Konzept gewählt, bei dem die Tabellen im lokalen Speicher eines jeden Prozessors angelegt werden.

5.1.1 Template-Tabelle

Die Informationen, die die Template-Tabelle enthält, sind in Abbildung 5.1 dargestellt. In dieser Tabelle sind M Templates gespeichert. Für jedes Template wird der Template-Name, der Unterprogrammname der Template-Deklarationsumgebung, der Rang des Templates und die Ausdehnung des Templates in jeder Dimension festgehalten.

In den nächsten zwei Feldern wird der lokale und globale Template-Status gespeichert. Ein Template kann drei Zustände annehmen:

- **template_empty:**
Der Rang des Templates wurde definiert. Die Ausdehnung in einer Dimension ist noch nicht definiert worden.
- **template_declared:**
Der Rang und die Ausdehnung in jeder Dimension wurden definiert.
- **template_defined:**
Der Rang und die Ausdehnung in jeder Dimension wurden definiert. Außerdem wurde das Template auf ein Prozessorfeld verteilt. Diese Verteilung wurde

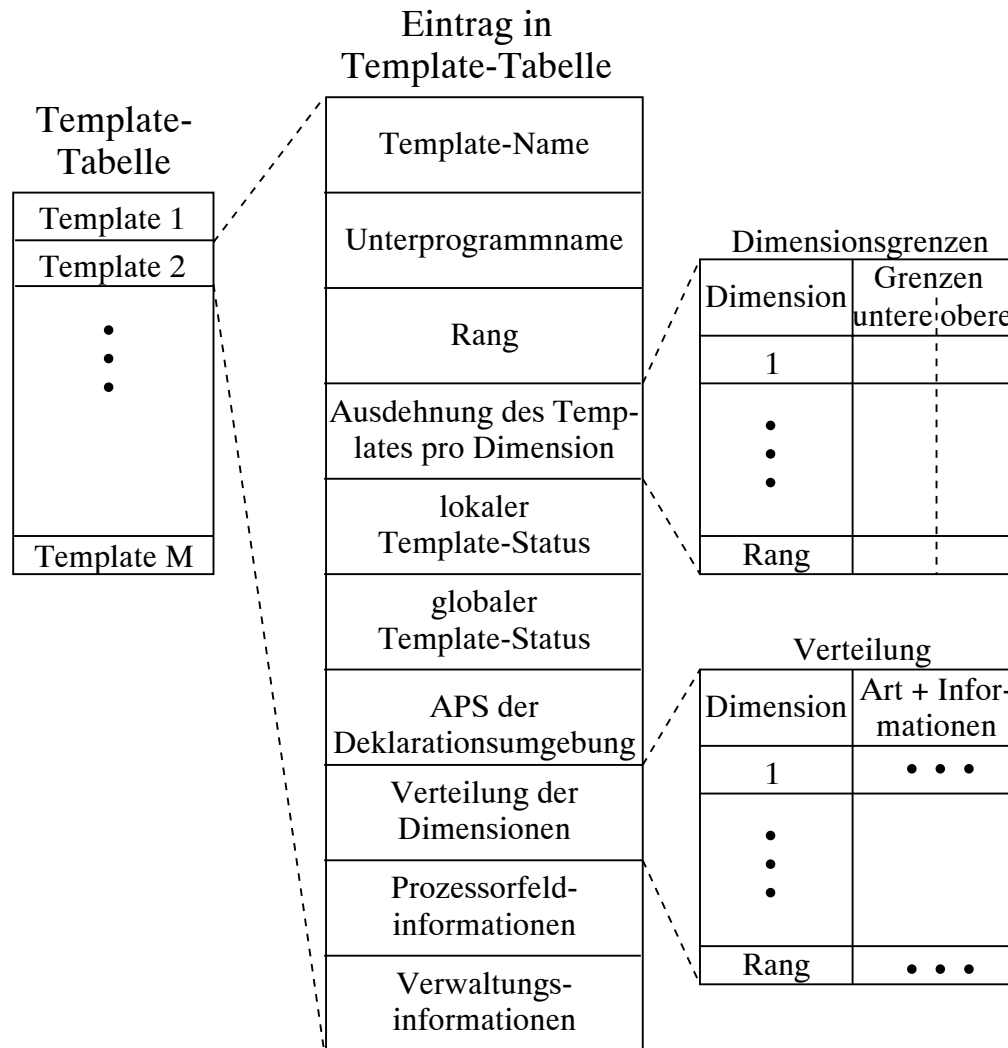


Abbildung 5.1: Inhalt der Template-Tabelle

zusätzlich im Template abgespeichert. Welche Parameter abgespeichert werden, wird in Abschnitt 5.4 bei der RE-/DISTRIBUTE-Direktive diskutiert.

Ein Template hat einen dieser drei Zustände. Eine Zustandsänderung des Templates kann bei der Ausführung einer Direktive, die das Template betrifft, stattfinden. In Abbildung 5.2 sind alle möglichen Zustandsübergänge und die Direktiven, die diese auslösen, festgehalten. Die Funktionsweise der einzelnen Direktiven wurde schon erklärt oder wird in diesem Abschnitt noch erklärt werden.

Der lokale Zustand des Templates stimmt immer mit dem globalen Zustand überein, außer in dem Fall, daß ein Template auf ein Prozessorfeld verteilt worden ist und einem Prozessor, der zum Prozessorfeld gehört, kein Template-Element zugewiesen worden ist. In diesem Fall ist der Zustand des Templates lokal auf diesem Prozessor *template_declared*, der globale Zustand hingegen ist *template_defined*. Dieser Prozessor muß aber bei einigen Direktiven wie der PDO-Direktive in Verbindung mit dem

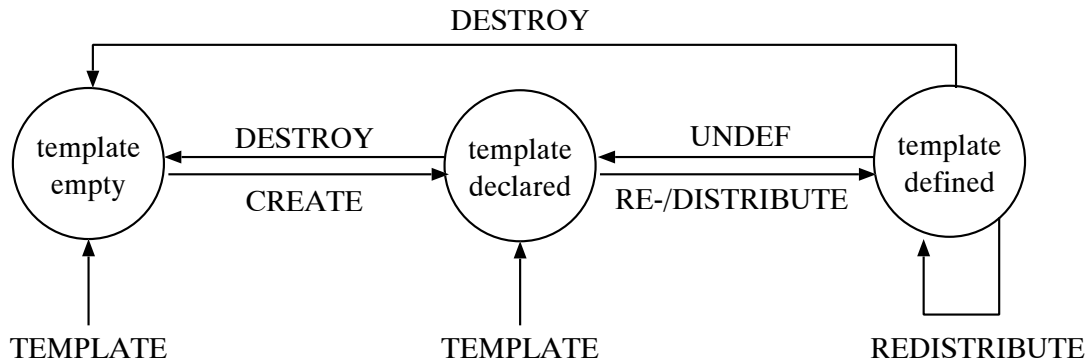


Abbildung 5.2: Zustandsdiagramm für Template

DISTRIBUTE-ONCE-Konstrukt den globalen Status des Templates kennen, um die Direktiven korrekt ausführen zu können (siehe Abschnitt 5.6.2).

In der Template-Tabelle wird das APS der Deklarationsumgebung gespeichert, um so bei der DISTRIBUTE-Direktive feststellen zu können, ob das APS der Template-Deklarationsumgebung mit dem derzeit aktiven APS übereinstimmt (vgl. Abschnitt 4.2.3). Wenn die APS unterschiedlich sind, dann wird eine Laufzeitfehlermeldung ausgegeben.

Wenn sich ein Template im Status *template_defined* befindet, dann ist es auf ein Prozessorfeld verteilt worden. Daher müssen dann auch die Verteilungen der Dimensionen und Prozessorfeldinformationen abgespeichert werden. Die Speicherung der Verteilungen geschieht dimensionsweise, d.h. in jeder Dimension wird die Art der Verteilung und die jeweils geeigneten Parameter festgehalten. Zusätzlich wird das Prozessorfeld, auf das ein Template verteilt worden ist, kopiert und in dem Template gespeichert, was durch das Feld Prozessorfeldinformationen in Abbildung 5.1 angedeutet ist.

Schließlich werden noch verschiedene Verwaltungsinformationen in dem Template gespeichert, die an dieser Stelle nicht weiter erläutert werden, weil sie für das Verständnis der Datenstruktur nicht wichtig sind.

5.1.2 Prozessorfeldtabelle

Die Informationen, die die Prozessorfeldtabelle beinhaltet, können aus Abbildung 5.3 entnommen werden. In dieser Tabelle sind N Prozessorfelder abgespeichert. Zu jedem Prozessorfeld wird der Name des Prozessorfeldes, der Name des Unterprogramms, in dem es deklariert worden ist, der Rang, die Ausdehnung in jeder Dimension, das APS der Deklarationsumgebung sowie der Anfang und das Ende im APS gespeichert. Die drei das APS betreffenden Angaben müssen gespeichert werden, um so beispielsweise das Prozessorfeldelement berechnen zu können, wenn nur die Nummer des Prozessors im APS gegeben ist und umgekehrt.

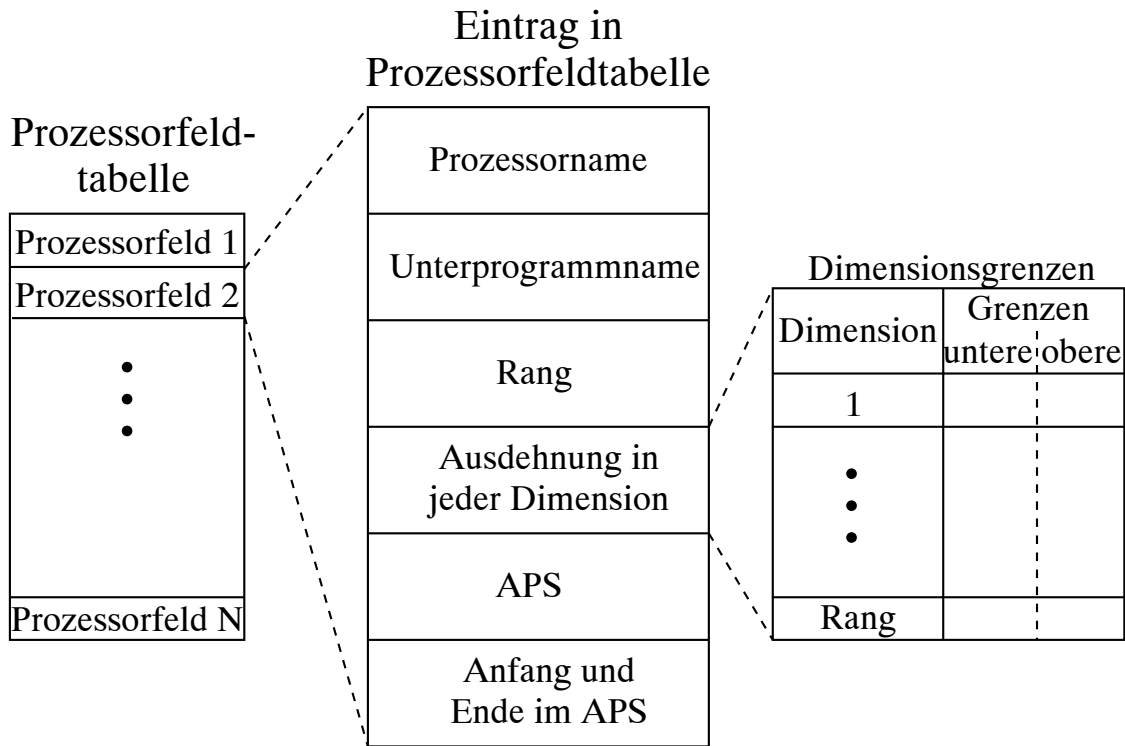


Abbildung 5.3: Inhalt der Prozessorfeldtabelle

5.2 PROCESSORS-Direktive

Mit der PROCESSORS-Direktive können Prozessorfelder deklariert werden. Der Prozessorname, der Unterprogrammname der Deklarationsumgebung und der Rang des Prozessorfeldes steht zur Übersetzungszeit fest und ändert sich zur Laufzeit nicht. Deshalb werden diese Informationen vor Programmausführung in die Prozessorfeldtabelle eingetragen. Das APS wird immer zur Laufzeit in die Prozessorfeldtabelle eingetragen. Die Ausdehnung in jeder Dimension sowie Anfang und Ende des Prozessorfeldes im APS kann zur Übersetzungszeit bestimmt werden, falls die Angaben konstant sind. Ansonsten müssen sie während der Direktivenausführung eingetragen werden. In der PROCESSORS-Direktive wurde die Strategie verfolgt, alle Angaben, die konstant sind, in der Initialisierungsphase einmal in die Tabelle einzulesen. Die nicht konstanten Werte werden in dem Unterprogramm SVM_PROC_FILL der Laufzeitbibliothek während der Direktivenausführung in die Tabelle eingetragen. Dies kann nicht im Deklarationsteil stattfinden, da Wertzuweisungen und ein Unterprogrammaufruf nötig sind. Deshalb werden diese vor dem nächsten ausführbaren Befehl ausgeführt. Im einzelnen sieht dies wie folgt aus:

- **Ausdehnung in jeder Dimension:**

Ein Prozessorfeld P mit Rang n , formal dargestellt durch $P(u_1 : o_1, u_2 : o_2, \dots, u_n : o_n)$, wird mit der PROCESSORS-Direktive deklariert. Sei s mit $0 \leq s \leq n$ die Anzahl der Dimensionen, in der mindestens die Unter- oder

Obergrenze nicht konstant ist, und $l_1, \dots, l_s$ spezifiziere diese Dimensionen. Dann werden die nicht konstanten Dimensionen in der Variablen SVM_FIELD und die Anzahl s der nicht konstanten Dimensionen dem Unterprogramm SVM_PROC_FILL übergeben. Die Abspeicherung der Dimension in dieser Variablen sieht wie folgt aus:

```
C *** Deklaration der Variablen SVM_FIELD
      INTEGER SVM_FIELD(7,3)

      ....

C *** Wertzuweisungen
      SVM_FIELD(1,1) = l1
      SVM_FIELD(1,2) = ul1
      SVM_FIELD(1,3) = ol1

      ....

      SVM_FIELD(s,1) = ls
      SVM_FIELD(s,2) = uls
      SVM_FIELD(s,3) = ols
```

• **Berechnung des Anfangs und Endes des Prozessorfeldes im APS:**

Falls mehrere Prozessorfelder in einer Programmeinheit deklariert werden, dann werden sie in der Reihenfolge ihres Auftretens auf das APS abgebildet. Es werden insgesamt n Prozessorfelddeklarationen in einer Programmeinheit durchgeführt. Diese Prozessorfelder seien o.B.d.A. von $P_1, \dots, P_n$ durchnummeriert. Ferner geben die Funktionen $o_l(j)$ und $u_l(j)$ mit $1 \leq j \leq n$ die obere bzw. untere Grenze der Prozessorfeldes P_j in der l -ten Dimension zurück. Sei i mit $1 \leq i \leq n$ die erste Prozessorfelddeklaration, in der in einer Dimension eine nicht konstante Grenze verwendet wird und die vorherigen Prozessorfelder seien auf die abstrakten Prozessoren des APS von 0 bis $MAPPED_APS_PROCS$ abgebildet worden. Ferner sei die Funktion $NUM_PROCS(j)$ mit $1 \leq j \leq n$ für jedes Prozessorfeld definiert durch

$$NUM_PROCS(j) := \prod_{l=1}^{Rang \text{ von } P_j} (o_l(j) - u_l(j) + 1).$$

Diese Funktion gibt die Anzahl der abstrakten Prozessoren an, die das Prozessorfeld P_j benötigt.

Dann wird P_i auf die APS-Prozessoren von $MAPPED_APS_PROCS + 1$ bis $MAPPED_APS_PROCS + NUM_PROCS(i)$ abgebildet. Die restlichen Prozessorfelder k mit $i < k \leq n$ werden dann auf die APS-Prozessoren von

$$MAPPED_APS_PROCS + 1 + \sum_{l=i}^{k-1} NUM_PROCS(l)$$

bis

$$MAPPED_APS_PROCS + 1 + \sum_{l=i}^k NUM_PROCS(l)$$

abgebildet. Diese Berechnungen werden im Fortran-Programm über die Variablen SVM_FIRST_PROC_IN_APS und SVM_LAST_PROC_IN_APS durchgeführt. Haben diese den Wert -1, so bedeutet dies, daß die Berechnungen schon zur Übersetzungszeit durchgeführt werden konnten, da die Prozessorfelder, die zuvor schon auf das APS abgebildet wurden, nur konstante Grenzen in allen Dimensionen haben.

Die Variablen, die in der oben beschriebenen Weise belegt werden, und der Prozessor-identifikator werden dem Unterprogramm SVM_PROC_FILL übergeben, das dann diese Werte in die Prozessorfeldtabelle an die entsprechende Stelle einträgt. Ein Beispiel soll die oben erwähnten Mechanismen im Fortran-Programm erläutern. Die Direktive im Unterprogramm

```
C *** Unterprogramm in Fortran, dem die Integer-Variable N uebergeben wird
      SUBROUTINE EXAMPLE_PROCESSORS1(N)
      INTEGER N

C *** Prozessorfelddeklaration von 3 Prozessorfeldern P1, P2, P3
CSVM$ PROCESSORS:: P1(-5:5,10), P2(1:N, 1:2*N), P3(4,10)

      ....

      END
```

wird umgesetzt zu

```
C *** Unterprogramm in Fortran, dem die Integer-Variable N uebergeben wird
      SUBROUTINE EXAMPLE_PROCESSORS1(N)
      INTEGER N

C *** Deklaration der neu benutzten Variablen
      INTEGER SVM_FIELD(7,3)
      INTEGER SVM_LAST_PROC_IN_APS
      INTEGER SVM_FIRST_PROC_IN_APS

      ....

C *** generierter Code, der vor dem ersten ausfuehrbaren Statement in dem
C *** Unterprogramm ausgefuehrt wird
      SVM_FIRST_PROC_IN_APS = -1
      SVM_LAST_PROC_IN_APS = -1

C *** In die Prozessorfeldtabelle wird bei P1 nur das APS abgespeichert
      CALL SVM_PROC_FILL(1,SVM_FIRST_PROC_IN_APS,SVM_LAST_PROC_IN_APS,0,
+                        SVM_FIELD)
```


abgebildet, umgewandelt:

<pre> PROGRAM MULTIPLIKATION CSVM\$ PROCESSORS::P(1,10) CALL MULT(P) END </pre>	<pre> PROGRAM MULTIPLIKATION CSVM\$ PROCESSORS::P(1,10) INTEGER P C *** weise der Variablen P den Prozessor- C *** identifikator, in diesem Fall 5, zu P=5 CALL MULT(P) END </pre>
<pre> SUBROUTINE MULT(Q) CSVM\$ PROCESSORS::Q(1,10) END </pre>	<pre> SUBROUTINE MULT(Q) CSVM\$ PROCESSORS::Q(1,10) INTEGER Q, SVM_FIELD(7,3) C *** ueberpruefe, ob aktuelles und C *** formales Argument bzgl. Rang C *** und Ausdehnung in jeder Dimension C *** uebereinstimmen CALL SVM_DUMMY_PROC(2,Q,0, SVM_FIELD) END </pre>

In der aufrufenden Programmeinheit wird der Prozessorfeldname als *Integer*-Variable deklariert. Dieser Variablen wird vor der ersten ausführbaren Anweisung des aufrufenden Programms der Prozessorfeldidentifikator zugewiesen.

In der Programmeinheit, die als formalen Parameter ein Prozessorfeld enthält, wird eine *Integer*-Variable deklariert. Dadurch ist sichergestellt, daß der aktuelle und formale Parameter denselben Typ besitzen. Wenn das Prozessorfeld in der Programmeinheit z.B. bei einem *Barrier* verwendet wird, dann muß der Wert der Variablen nur als Index in der Prozessorfeldtabelle interpretiert werden, um so auf die Daten des aktuellen Prozessorfeldes zugreifen zu können.

Weiterhin wird in dem Unterprogramm SVM_DUMMY_PROC der Laufzeitbibliothek überprüft, ob das aktuelle Prozessorfeld mit der Deklaration des formalen Prozessorfeldes übereinstimmt. Die Werte für das formale Prozessorfeld werden mit denselben Mechanismen, wie bei der Prozessorfelddeklaration bereits erläutert, dem Unterprogramm SVM_DUMMY_PROC übergeben.

5.3 TEMPLATE- und CREATE-Direktive

Mit der TEMPLATE-Direktive werden Templates deklariert. Der Template-Name, der Unterprogrammname der Deklarationsumgebung und der Rang des Templates ändern sich zur Laufzeit nicht. Diese werden deshalb in der Initialisierungsphase des Programms in die Template-Tabelle eingetragen. Das APS der Deklarationsumgebung wird immer zur Laufzeit in die Template-Tabelle eingetragen.

Die Ausdehnung in den Template-Dimensionen hängt bei der TEMPLATE- und CREATE-Direktive häufig vom Ausführungskontext ab. Falls die Dimensionsgrenzen konstant sind, so werden sie vor dem Start des Programms in die Tabelle eingelesen, ansonsten wird derselbe Mechanismus wie beim Prozessorfeld in Abschnitt 5.2 unter dem Punkt *Ausdehnung in jeder Dimension* verwendet. Bei der TEMPLATE-Direktive wird das Unterprogramm SVM_TEMPL_FILL der Laufzeitbibliothek vor dem ersten ausführbaren Befehl in der Programmeinheit aufgerufen, während dieses Unterprogramm bei der CREATE-Direktive unmittelbar hinter der Direktive aufgerufen wird. In einem Feld werden dann die nicht konstanten Dimensionen dem Unterprogramm übergeben.

Nachdem die CREATE-Direktive ausgeführt worden ist, befindet sich das Template in dem Zustand *template_declared* (vgl. Abbildung 5.2). Ist bei der Template-Direktive der Rang und die Ausdehnung in jeder Dimension definiert worden, so befindet sich das Template nach Ausführung der Direktive ebenfalls im Zustand *template_declared*. Ist bei der Template-Direktive hingegen nur der Rang eines Templates definiert worden, so hat das Template den Zustand *template_empty*.

Mit Hilfe eines Beispiels soll die Umwandlung des Fortran-Programms bei der Umsetzung der TEMPLATE- und CREATE-Direktive verdeutlicht werden:

```

PROGRAM EXAMPLE_TEMPLATE1
  INTEGER N
C *** Deklaration von Template T1 und T2
CSVM$ TEMPLATE::T1(1000)
CSVM$ TEMPLATE::T2(:, :)

C *** N haengt von Berechnungen im Programm ab
      N = ....

C *** Template T2 wird kreiert
CSVM$ CREATE::T2(100,N)

      ....
END

```

Dies wird transformiert zu:

```

PROGRAM EXAMPLE_TEMPLATE1
  INTEGER N
C *** Deklaration des neu benutzten Feldes

```

```

      INTEGER SVM_FIELD(7,3)

C *** vor dem ersten ausfuehrbaren Befehl der Programmeinheit die zwei
C ***      Funktionsaufrufe

C *** Speichere in Template-Tabelle bei T1 das APS der Deklarationsumgebung
C *** und setze Zustand auf template_declared
      CALL SVM_TEMPL_FILL(1,0,SVM_FIELD,0)

C *** Speichere in Template-Tabelle bei T2 das APS der Deklarationsumgebung
C *** und setze Zustand auf template_empty
      CALL SVM_STORE_APS_IN_TEMPL(2)


C *** N haengt von Berechnungen im Programm ab
      N = ....

C *** Template T2 wird kreiert
CSVM$ CREATE::T2(100,N)
      SVM_FIELD(1,1) = 2
      SVM_FIELD(1,2) = 1
      SVM_FIELD(1,3) = N
C *** Unterprogrammaufruf, um Dimensionsgrenzen in Template-Tabelle einzutragen
C *** und Zustand auf template_declared zu setzen
      CALL SVM_TEMPL_FILL(2,1,SVM_FIELD,1)

      ....
      END

```

Zu diesem Beispiel sei noch bemerkt, daß durch die TEMPLATE-Direktive T2(:, :) lediglich der Rang deklariert wird. Daher wird T2 auf den Zustand *template_empty* gesetzt. Erst bei der CREATE-Direktive ändert sich der Zustand auf *template_declared*. Da bei T1 in der TEMPLATE-Direktive auch die Ausdehnung in jeder Dimension T1(1000) definiert wurde, wird T1 in der TEMPLATE-Direktive auf den Zustand *template_declared* gesetzt.

Ein Template kann durch Parameterübergabe an ein Unterprogramm oder eine Funktion weitergegeben werden. Der Mechanismus, der bei der Umsetzung angewendet wird, ist derselbe wie bei der Prozessorfeldparameterübergabe, d.h. Einführung einer *Integer*-Variablen, die den Template-Identifikator enthält. In dem aufgerufenen Unterprogramm wird eine Routine aufgerufen, die überprüft, ob das aktuelle Template-Argument und der formale Template-Parameter, der in der Programmeinheit deklariert ist, übereinstimmt.

5.4 DISTRIBUTE- und REDISTRIBUTE-Direktive

Mit der DISTRIBUTE- bzw. REDISTRIBUTE-Direktive können Templates auf Prozessorfelder verteilt werden. Eine Vorbedingung für die Ausführung dieser Direktiven ist, daß sich das Template im Zustand *template_declared* befindet. Nach Ausführung der DISTRIBUTE-Direktive wird der neue Zustand *template_defined* in das globale Status-Flag des Templates eingetragen. Weiterhin ändert jeder Prozessor, dem mindestens ein Template-Element zugewiesen wird, in seiner lokalen Template-Tabelle den Status des Templates in *template_defined* um. Die anderen Prozessoren lassen den Status des Templates in der Template-Tabelle unverändert. Weiterhin wird das Prozessorfeld, auf das ein Template verteilt wird, in die Template-Tabelle hineinkopiert. Der Grund hierfür liegt darin, daß die Informationen, die sich im Prozessorfeld befinden, beispielsweise bei der *linked distribution* [50] benötigt werden. Da der Gültigkeitsbereich eines Templates und Prozessorfeldes aber unterschiedlich sein kann, muß das Prozessorfeld kopiert werden.

```

        PROGRAM DISTR_TEMPL_IN_SUBR
C *** Template-Deklaration
CSVM$ TEMPLATE::T1(100,30000)

        ....

C *** T1 ist nicht auf ein Prozessorfeld verteilt
        CALL DISTR_TEMPL(T1)
C *** Prozessorfeld P aus Unterprogramm B in dieser Umgebung nicht sichtbar,
C *** dennoch ist T1 nach Unterprogrammaufruf von B auf Prozessoren verteilt

        ....

        END

        SUBROUTINE DISTR_TEMPL(T)
C *** Prozessorfelddeklaration
CSVM$ PROCESSORS::P(5,5)

C *** Deklaration des formalen Template-Argumentes
CSVM$ TEMPLATE::T(100,30000)

C *** Verteilung des Templates T wird an das aktuelle Argument der aufrufenden
C *** Stelle uebergeben
CSVM$ DISTRIBUTE (BLOCK, CYCLIC(2)) ONTO P :: T

        ....

        END

```

In Beispielprogramm DISTR_TEMPL_IN_SUBR wird das Template T1 deklariert, das, bevor es an das Unterprogramm DISTR_TEMPL übergeben wird, nicht auf

ein Prozessorfeld verteilt worden ist. In dem Unterprogramm wird es auf ein dort deklariertes Prozessorfeld verteilt. Nach Beendigung des Unterprogramms existiert das Prozessorfeld in dem aufrufenden Programm DISTR_TEMPLIN.SUBR im Gegensatz zum Template nicht weiter.

Nachfolgend wird für die BLOCK- und die CYCLIC-Verteilung ausführlich beschrieben, welche Parameter im Template abgespeichert werden. Im weiteren bezeichne N die Anzahl der Template-Elemente in einer Dimension, die verteilt werden sollen, und P die Anzahl der Prozessoren in einer Prozessorfelddimension, auf die die Template-Dimension verteilt werden soll. Diese Prozessoren seien o.B.d.A. von 1 bis P durchnummeriert, und p mit $1 \leq p \leq P$ bezeichne den Prozessor, der einen Teil des Templates erhalten soll.

Wenn im Template eine Dimension d mit der HPF-BLOCK-Verteilung auf eine Prozessorfelddimension verteilt worden ist, dann speichert sich der Prozessor p in seiner lokalen Template-Tabelle in der Dimension d die Verteilungsart sowie die Unter- und Obergrenze des Segmentes, das ihm zugewiesen wird:

$$\begin{aligned} \text{Untergrenze} &= \left\lceil \frac{N}{P} \right\rceil \cdot (p - 1) + 1 \\ \text{Obergrenze} &= \min(\text{Untergrenze} + \left\lceil \frac{N}{P} \right\rceil - 1, N) \end{aligned}$$

Um feststellen zu können, ob ein Template-Element e mit $1 \leq e \leq N$ in der Dimension d auf dem Prozessor p abgespeichert wurde, muß nur überprüft werden, ob das Element e im Intervall $[\text{Untergrenze}, \text{Obergrenze}]$ enthalten ist¹.

Bei der CYCLIC-Verteilung [7, 8] der Dimension d eines Templates auf eine Prozessorfelddimension definiere der Parameter l die Segmentgröße bei der zyklischen Verteilung (siehe Abschnitt 3.2.1). Die Anzahl der Segmente, die Prozessor p erhält, ist gegeben durch

$$\text{Anz_Seg}(p) = \left\lceil \frac{N - (p - 1) \cdot l}{P \cdot l} \right\rceil.$$

Die untere Grenze des Segmentes i , $UG(p,i)$ mit $1 \leq i \leq \text{Anz_Seg}(p)$, ergibt sich aus

$$UG(p,i) = \underbrace{(p - 1) \cdot l + 1}_A + (i - 1) \cdot \underbrace{P \cdot l}_B.$$

Die obere Grenze des Segmentes i , $OG(p,i)$ mit $1 \leq i \leq \text{Anz_Seg}$, ergibt sich aus

¹Dies wird beispielsweise bei der *linked distribution* benötigt.

$$OG(p, i) = \min(UG(p, i) + l - 1, N).$$

Der Prozessor p speichert in der Dimension d des Templates, das in der lokalen Template-Tabelle abgelegt ist, die Verteilungsart, den Parameter l , $Anz_Seg(p)$ und die zwei Zahlen A und B ab, die in dem Berechnungsausdruck der Untergrenze gekennzeichnet sind. Mit diesen Informationen ist es möglich, die Template-Elemente aufzulisten, die dem Prozessor p zugeordnet wurden.

Um auch hier feststellen zu können, ob ein Template-Element e mit $1 \leq e \leq N$ in der Dimension d auf dem Prozessor p abgespeichert wurde, muß folgender Vergleich durchgeführt werden:

$$p = MODULO\left(\left\lfloor \frac{e-1}{l} \right\rfloor, P\right) + 1$$

Falls dies nicht gilt, dann ist das Element e nicht auf dem Prozessor p abgespeichert, ansonsten befindet sich e auf p .

Falls im Template eine Dimension sequenzialisiert wird, dann werden in dieser Dimension die Verteilungsart und die Template-Grenzen abgespeichert. Bei der GENERAL-BLOCK-Verteilung werden, wie bei der BLOCK-Verteilung, die BLOCK-Grenzen abgespeichert, die durch das Feld bestimmt werden, das dem Konstrukt übergeben wird.²

Abschließend wird die Umsetzung der DISTRIBUTE-Direktive zuerst formal und dann an einem Beispiel erläutert. Bei dieser Direktive wird, ähnlich der Prozessor- bzw. Template-Tabelle, eine **Distribute-Tabelle** im lokalen Speicher eines jeden Prozessors der *initial processor set* angelegt. Jede Direktive erhält zur Übersetzungszeit einen Identifikator, der auch als Index interpretiert werden kann, der die Position in der Distribute-Tabelle angibt, wo sich die Informationen zu dieser Direktive befinden. In der Tabelle werden die Informationen eingetragen, die sich während der Programmausführung nicht ändern:

- **Verteilungsart des Templates pro Dimension:**

In die Tabelle wird die Art der Verteilung³, ob eine Dimension sequenzialisiert oder mit der BLOCK-, CYCLIC- oder GENERAL-BLOCK-Strategie verteilt werden soll, während der Initialisierungsphase des Programms eingetragen. Bei den zwei letztgenannten Strategien muß zusätzlich noch der Parameter, der die Segmentgröße bei der CYCLIC-Strategie bestimmt, bzw. ein Zeiger auf das Feld, das die Segmentgrößen bei der GENERAL-BLOCK-Strategie angibt, übergeben werden, um die Berechnungen für die Verteilungen ausführen zu können. Diese zwei Werte werden nicht in die Distribute-Tabelle bei Programmstart eingetragen, da sie in der Regel erst bei der Direktivenausführung

² vgl. Abbildung 4.2.3

³ Im Syntaxdiagramm ist dies in der DISTRIBUTE-Direktive die *dist-format-list*.

bestimmt werden können und nicht schon in der Initialisierungsphase des Programms. Diese Parameter werden dem Unterprogramm SVM_DISTRIBUTE der Laufzeitbibliothek, das die Verteilung durchführt, über das Fortran-Feld SVM_FIELD1 übergeben.

Seien die n Verteilungsarten in einer DISTRIBUTE-Direktive gegeben durch $strat(i)$ mit $1 \leq i \leq n$, und l_j mit $1 \leq j \leq s$ spezifiziere die s Dimensionen mit $0 \leq s \leq n$, in denen $strat(l_j)$ eine CYCLIC- bzw. GENERAL-BLOCK-Strategie zurückliefert. Weiterhin gebe die Funktion $GET_PARAMETER(k)$ mit $k \in \{l_1, \dots, l_s\}$ für die k -te Dimension den Parameter bei einer CYCLIC-Strategie bzw. einen Zeiger auf das Feld bei einer GENERAL-BLOCK-Verteilung zurück. Dann werden diese Werte der Variablen in dem Feld SVM_FIELD1 wie folgt abgespeichert:

```
C *** Deklaration der Variablen SVM_FIELD1
      INTEGER SVM_FIELD1(7)

      ....

C *** Wertzuweisungen
      SVM_FIELD1(l1) = GET_PARAMETER(l1)
      SVM_FIELD1(l2) = GET_PARAMETER(l2)

      ....

      SVM_FIELD1(ls) = GET_PARAMETER(ls)
```

- **Spezifikation des Prozessorfeldes pro Dimension:**

Die drei Arten, ein Prozessorfeld zu spezifizieren, im Syntaxdiagramm der DISTRIBUTE-Direktive als *dist-spec-list* bezeichnet, sind die Replikation, der *Integer*-Ausdruck und die Sektion. Diese Arten werden in der Distribute-Tabelle gespeichert. Bei dem *Integer*-Ausdruck und der Sektion muß der Ausdruck bzw. die Sektionsgrenzen dem Unterprogramm SVM_DISTRIBUTE, das die Verteilung durchführt, übergeben werden, da es in der Regel von Parametern abhängt, die während der Direktivenausführung erst bestimmt werden können. Dies geschieht über die Variable SVM_FIELD2.

Seien die m Prozessorspezifikationen in einer DISTRIBUTE-Direktive gegeben durch $spec(i)$ mit $1 \leq i \leq m$, und l_j mit $1 \leq j \leq s$ gebe die s Dimensionen mit $0 \leq s \leq m$ an, in denen $spec(l_j)$ einen *Integer*-Ausdruck bzw. eine Sektion spezifiziere. Die Funktionen $GET_LOWER(k)$ bzw. $GET_UPPER(k)$ mit $k \in \{l_1, \dots, l_s\}$ liefern für die k -te Dimension die Unter- bzw. Obergrenze einer Sektion oder bei einem *Integer*-Ausdruck den Ausdruck in beiden Funktionen, was die Implementierung vereinfacht, zurück. Dann werden der Variablen SVM_FIELD2 die Werte folgendermaßen zugewiesen:

```
C *** Deklaration der Variablen SVM_FIELD2
      INTEGER SVM_FIELD2(7,3)
```

```

....

C *** Wertzuweisungen
SVM_FIELD2(l1,1) = l1
SVM_FIELD2(l1,2) = GET_LOWER(l1)
SVM_FIELD2(l1,3) = GET_UPPER(l1)

....

SVM_FIELD2(ls,1) = ls
SVM_FIELD2(ls,2) = GET_LOWER(ls)
SVM_FIELD2(ls,3) = GET_UPPER(ls)

```

An einem komplexeren Beispiel wird gezeigt, wie die oben erwähnten Mechanismen in die Umsetzung integriert werden. In dem Unterprogramm, dem ein Feld FIELD übergeben wird, in dem sich die Segmentgrößen befinden, die jedem Prozessor in einer Dimension mit der GENERAL-BLOCK-Verteilung zugewiesen werden sollen,

```

SUBROUTINE EXAMPLE_DISTRIBUTE1(FIELD)

C *** Feld, das Segmentgroessen fuer GENERAL-BLOCK-Verteilung beinhaltet
INTEGER FIELD(4)

C *** Deklaration eines Prozessorfeldes P mit 5 Dimensionen
CSVM$ PROCESSORS::P(4,2,5,5,3)
C *** Deklaration eines Template mit 4 Dimensionen
CSVM$ TEMPLATE::T(100,200,300,400)

C *** Das Template T wird auf das Prozessorfeld P verteilt
CSVM$ DISTRIBUTE (BLOCK, CYCLIC(3), GENERAL_BLOCK(FIELD), *)
CSVM$+          ONTO P(1:4,*,2:4,1:5, 2)::T

....
END

```

wird die DISTRIBUTE-Direktive expandiert zu

```

SUBROUTINE EXAMPLE_DISTRIBUTE1(FIELD)

C *** Feld, das Segmentgroessen fuer GENERAL-BLOCK-Verteilung beinhaltet
INTEGER FIELD(4)
C *** Deklaration der neu verwendeten Variablen
INTEGER SVM_FIELD1(7), SVM_FIELD2(7,3)

CSVM$ PROCESSORS::P(4,2,5,5,3)
CSVM$ TEMPLATE::T(100,200,300,400)

C *** Code, der fuer die DISTRIBUTE-Direktive erzeugt wird und vor das erste
C *** ausfuehrbare Statement des Unterprogramms gesetzt wird

```

```

C *** Parameter der CYCLIC-Verteilung, der Segmentgroesse bestimmt, wird
C *** abgespeichert
    SVM_FIELD1(2) = 3
C *** Zeiger auf das Feld wird abgespeichert
    SVM_FIELD1(3) = %LOC(FIELD)

C *** Abspeichern der Sektionen, die im Prozessorfeld in der 1-ten, 3-ten und
C *** 4-ten Dimension spezifiziert sind
    SVM_FIELD2(1,1) = 1
    SVM_FIELD2(1,2) = 1
    SVM_FIELD2(1,3) = 4
    SVM_FIELD2(3,1) = 3
    SVM_FIELD2(3,2) = 2
    SVM_FIELD2(3,3) = 4
    SVM_FIELD2(4,1) = 4
    SVM_FIELD2(4,2) = 1
    SVM_FIELD2(4,3) = 5
C *** Abspeichern des Integer-Ausdrucks, der im Prozessorfeld in der 5-ten
C *** Dimension spezifiziert ist
    SVM_FIELD2(5,1) = 5
    SVM_FIELD2(5,2) = 2
    SVM_FIELD2(5,3) = 2

C *** Unterprogrammaufruf, der die Verteilung in T abspeichert
    CALL SVM_DISTRIBUTE(1,1,1,SVM_FIELD1,SVM_FIELD2)

    ....
    END

```

An dieser Stelle sei noch bemerkt, daß dem Unterprogramm SVM_DISTRIBUTE der Identifikator der DISTRIBUTE-Direktive, des Prozessorfeldes und des Templates übergeben wird, zusätzlich zu den Variablen SVM_FIELD1 und SVM_FIELD2. Dadurch kann in dem Unterprogramm auf die Distribute-, Prozessor- und Template-Tabelle zugegriffen werden, um die notwendigen Informationen daraus zu entnehmen. Weiterhin wird die Zuordnung der Template-Dimensionen zu den Prozessorfelddimensionen nach dem in Abschnitt 4.2.3 beschriebenen Schema vorgenommen, und die Template-Elemente werden nach dieser Zuordnung auf die Prozessoren verteilt.

5.5 UNDEF- und DESTROY-Direktive

Die UNDEF- und DESTROY-Direktive werden an dieser Stelle gemeinsam behandelt, da sie hauptsächlich eine Zustandsänderung des Templates, in Abbildung 5.2 dargestellt, bewirken.

Die UNDEF-Direktive bewirkt, daß eine Verteilung, die in einem Template gespeichert wurde, verworfen wird. Daher wird das Template von dem Zustand *templa-*

te_defined in den Zustand *template_declared* überführt.

Die DESTROY-Direktive zerstört ein Template, das zuvor mit der CREATE-Direktive angelegt worden ist. Nach Ausführung dieser Direktive hat das Template den Zustand *template_empty*. Anschließend kann das Template mit der CREATE-Direktive wieder kreiert werden. Beiden Direktiven ist gemein, daß sie nur durch einen einzigen Unterprogrammaufruf in Fortran realisiert werden.

5.6 PDO-Direktive

Mit der PDO-Direktive wird spezifiziert, daß Schleifeniterationen parallel auf mehreren Prozessoren ausgeführt werden können. In Verbindung mit Templates ist dies beim vordefinierten und dynamischen Schleifen-Scheduling möglich (vgl. Abschnitt 4.2 bzw. 4.3). Das vordefinierte Scheduling kann in der PDO-Direktive in der *Strategy*-Option mit dem ON-HOME-Konstrukt angesteuert werden und das dynamische Scheduling mit dem DISTRIBUTE-ONCE-Konstrukt. Die Umsetzung dieser zwei Konstrukte wird in den nachfolgenden Unterabschnitten behandelt.

5.6.1 ON-HOME-Scheduling

In der *Strategy*-Option der Direktive kann angegeben werden, daß Schleifeniterationen mit dem ON-HOME-Konstrukt in der Weise verteilt werden sollen, wie ein Template verteilt worden ist. Bei perfekt geschachtelten Schleifen wird jede zugehörige Schleife so verteilt, wie eine ihr zugeordnete Template-Dimension auf das Prozessorfeld verteilt worden ist⁴. Im folgenden wird nun beschrieben, wie eine parallele Schleife, die über ein Template verteilt werden soll, umgewandelt wird.

Die ursprüngliche Schleife, die den ganzen Iterationsraum bearbeitet hat, wird ersetzt durch zwei Schleifen, eine WHILE- und eine DO-Schleife. Die neue Schleifenstruktur sieht dann in einem Pseudo-Code wie folgt aus:

```
C *** lese die Anzahl der Segmente, die einem Prozessor in einer
C *** Template-Dimension zugewiesen wurden, aus
      Anz_Seg = I_SVM_GET_NUM_BLOCKS(Template-ID, Template-Dimension)
      Segment = 0

C *** durchlaufe in einer While-Schleife diese Segmente
      WHILE (Segment <= Anz_Seg)

          Segment = Segment + 1
C ***      Berechne zu jedem Segment die Grenzen von und bis
          SVM_GET_BOUNDS(Template-ID, Segment, von, bis)

          DO I=von, bis
```

⁴Wie diese Zuordnung gesteuert wird, wurde bereits in Abschnitt 4.2 erläutert.

C **urspruenglicher Schleifenkoerper**

 ENDDO
 ENDWHILE

In der inneren DO-Schleife werden nur die Iterationen ausgeführt, deren entsprechendes Template-Element auf dem Prozessor p abgespeichert ist. Um dies zu erreichen, liest man zuerst die Anzahl der Segmente aus, die bei der Verteilung einer Template-Dimension dem Prozessor p zugewiesen wurden. Dies geschieht in der Funktion `L_SVM_GET_NUM_BLOCKS`. In der WHILE-Schleife durchläuft man dann alle Segmente. Zu jedem Segment berechnet man mit der Funktion `SVM_GET_BOUNDS` die Unter- und Obergrenzen. Dieser Bereich beinhaltet zu diesem Segment alle Template-Elemente, die dem Prozessor p zugewiesen wurden. Diese Grenzen durchläuft man nun in der DO-Schleife und führt den ursprünglichen Schleifenkörper aus.

Die hier benötigten Werte werden alle aus der lokal gespeicherten Template-Tabelle gelesen. Die entsprechenden Werte wurden, wie bereits in Abschnitt 5.4 diskutiert, in die Template-Tabelle für die verschiedenen Verteilungsarten abgespeichert. Falls jedoch die Schrittweite der Schleife ungleich 1 ist, dann muß jedes Segment, das in der inneren Schleife bearbeitet wird, zuerst angepaßt werden, damit dieselben Iterationen wie im sequentiellen Fall bearbeitet werden. Ohne Beschränkung der Allgemeinheit befinden sich die Segmentgrenzen innerhalb der Schleifengrenzen, ansonsten werden die Segmentgrenzen durch die entsprechenden Schleifengrenzen überschrieben. Zwei Fälle werden unterschieden (SA bezeichne den Schleifenanfang, SW die Schrittweite, BA_{alt} und BA_{neu} den alten bzw. neuen Segmentanfang und BE das Segmentende):

1. Schrittweite > 1 :

Falls $(BA_{alt} - SA) \bmod SW$ ungleich 0 ist, dann wird der neue Segmentanfang berechnet aus

$$BA_{neu} = BA_{alt} + SW - (BA_{alt} - SA) \bmod SW,$$

ansonsten ist der alte Segmentanfang gleich dem neuen, da der Segmentanfang auf einer Schleifeniteration liegt, die bei der Schleife ausgeführt werden muß.

2. Schrittweite < 0 :

Zuerst wird BA_{alt} mit BE vertauscht, da das Segment von Ende bis Anfang, also in umgekehrter Reihenfolge, wegen der negativen Schrittweite, durchlaufen wird. Falls $(SA - BA_{alt}) \bmod SW$ gleich 0 ist, dann ist $BA_{alt} = BA_{neu}$, ansonsten wird der neue Segmentanfang berechnet aus

$$BA_{neu} = SA - (|SA - BA_{alt}| \operatorname{div} |SW| + 1) \cdot |SW|.$$

Die im Pseudo-Code vorgestellte allgemeine Schleifenstruktur wurde gewählt, um so alle Fälle, die bei der Verteilung einer Template-Dimension auftreten können, mit einer einzigen Schleifenstruktur bearbeiten zu können. Dies ist außerdem notwendig, da zur Übersetzungszeit meist unbekannt ist, wie eine Template-Dimension verteilt ist. Wird ein Template durch das *Interface* an eine Funktion weitergegeben, so hängt die Verteilung von dem Template der aufrufenden Stelle ab und kann von Aufruf zu Aufruf unterschiedlich sein.

An einem Beispiel wird die Schleifentransformation verdeutlicht, und es wird die Funktionsweise der einzelnen Funktionen detaillierter erklärt. Die zu verteilende Schleife sieht wie folgt aus:

```
CSVM$ PDO(LOOPS(I,J), STRATEGY(ON_HOME(T(I,J))))
      DO I = 1, 10
        DO J = 1, 10

C          ....

          ENDDO
        ENDDO
```

Diese perfekt geschachtelten Schleifen werden transformiert zu:

```
C *** Variablendeklarationen
      INTEGER SVM_BLOCK_NUM, SVM_CURRENT_BLOCK, SVM_CONT_ITER,
              SVM_FROM, SVM_TO, SVM_BLOCK_NUM_1, SVM_CURRENT_BLOCK_1,
              SVM_CONT_ITER_1, SVM_FROM_1, SVM_TO_1

C *** Ueberpruefe in I_SVM_EXE_LOOP, ob Prozessor einige Iterationen der
C *** Schleife ausfuehren muss und setze APS um
      IF (I_SVM_EXE_LOOP(1, 0, SVM_FIELD1, 0, SVM_FIELD2) .EQ. 1) THEN

C ***   Lese fuer die I-Schleife die Anzahl der Segmente ein
      SVM_BLOCK_NUM = I_SVM_GET_NUM_BLOCKS(1, 1)

C ***   Initialisierung der in der WHILE-Schleife benoetigten Laufvariablen
      SVM_CURRENT_BLOCK = 0

C ***   Anfang der in Fortran simulierten WHILE-Schleife
      1      CONTINUE

C ***       Inkrementiere Laufvariable in WHILE-Schleife
      SVM_CURRENT_BLOCK = SVM_CURRENT_BLOCK+1

C ***       Lese die Grenzen fuer das Segment SVM_CURRENT_BLOCK ein; falls kein
C ***       Segment mehr vorhanden, dann setze SVM_CONT_ITER auf 0 und beende
C ***       WHILE-Schleife
      CALL SVM_GET_BOUNDS(1,1,SVM_BLOCK_NUM,SVM_CURRENT_BLOCK,
+                          1,10,1,SVM_FROM,SVM_TO,SVM_CONT_ITER)

C ***       Falls kein Segment mehr vorhanden, dann beende Schleife und springe
```

```

C ***      hinaus
            IF (SVM_CONT_ITER .EQ. 0) GOTO 4

C ***      Bearbeite Iterationen
            DO I = SVM_FROM, SVM_TO

C ***      Beginn der inneren J-Schleife
            SVM_BLOCK_NUM_1 = I_SVM_GET_NUM_BLOCKS(1, 2)
            SVM_CURRENT_BLOCK_1 = 0

2          CONTINUE
            SVM_CURRENT_BLOCK_1 = SVM_CURRENT_BLOCK_1+1
            CALL SVM_GET_BOUNDS(1,2,SVM_BLOCK_NUM_1,
+                               SVM_CURRENT_BLOCK_1,1,10,1,
+                               SVM_FROM_1,SVM_TO_1,
+                               SVM_CONT_ITER_1)
            IF (SVM_CONT_ITER_1 .EQ. 0) GOTO 3
            DO J = SVM_FROM_1, SVM_TO_1

C ***      urspruenglicher Schleifenkoerper
C          ....

            ENDDO
            GOTO 2

3          CONTINUE
            ENDDO
            GOTO 1

4          CONTINUE

C ***      Setze APS zurueck
            CALL SVM_PSET_RESET_APS()
ENDIF

```

Am Anfang eines parallelen Schleifennestes wird mit der IF-Abfrage und der Funktion `I_SVM_EXE_LOOP` festgestellt, ob ein Prozessor einen Teil des Templates abgespeichert hat oder nicht. Falls nicht, dann überspringt er diese Schleife. Wenn dem Prozessor aber ein Teil des Templates zugewiesen wurde, dann führt er den THEN-Zweig der IF-Abfrage durch. Im Unterprogramm `I_SVM_EXE_LOOP` wird zusätzlich das APS umgesetzt, d.h. es befinden sich nur noch die Prozessoren im APS, die dieselben Schleifeniterationen kooperativ ausführen.

Danach wird die erste Schleife, die I-Schleife, transformiert. Zuerst wird die Anzahl der Segmente mit der Funktion `I_SVM_GET_NUM_BLOCKS` eingelesen und die Laufvariable der WHILE-Schleife initialisiert. Da in Standard-Fortran keine WHILE-Schleifen existieren, werden diese durch IF- und GOTO-Befehle realisiert. In der Funktion `SVM_GET_BOUNDS` wird überprüft, ob noch ein Segment des Templates dem Prozessor, der diese Funktion aufgerufen hat, zugewiesen wurde. Wenn dies nicht der Fall ist, dann wird das Flag `SVM_CONT_ITER` auf 0 gesetzt

und mit einem GOTO aus der WHILE-Schleife gesprungen, ansonsten werden die Grenzen dieses Segmentes ausgelesen, der Schrittweite der ursprünglichen Schleife entsprechend angepaßt und in der DO-Schleife mit der Laufvariablen I, die der ursprünglichen Schleifenvariablen entspricht, ausgeführt. Die zweite Schleife, die J-Schleife, wird genauso transformiert, außer daß sich die Namen der Hilfsvariablen unterscheiden.

Am Ende der Schleife wird dann das APS wieder auf das APS vor Schleifenausführung zurückgesetzt.

5.6.2 Scheduling über DISTRIBUTE-ONCE

Eine andere Möglichkeit, Schleifeniterationen auf Prozessoren zu verteilen, ist, in der STRATEGY-Option der PDO-Direktive die DISTRIBUTE-ONCE-Strategie zu wählen. Diese verteilt die Iterationen genau so wie das ON-HOME-Konstrukt, wenn das Template bereits auf ein Prozessorfeld verteilt wurde, also der globale Template-Zustand *template_defined* ist. Ist der Zustand hingegen *template_declared*, dann ist das Template noch nicht auf die Prozessoren verteilt worden. Daher wird es bei der ersten Schleifenausführung auf die Prozessoren gemäß den Verteilungsstrategien verteilt, die in dem Konstrukt angegeben sind (vgl. Abschnitt 4.3). Die Verteilungsarten, mit denen eine Template-Dimension auf eine Prozessorfelddimension verteilt werden kann, sind:

1. Sequentialisierung
2. BLOCK-Verteilung
3. CYCLIC-Verteilung
4. dynamische Arbeitsverteilungsstrategien

Die Transformation der parallelen Schleife mit Hilfe dieses Konstruktes sieht ähnlich aus wie die des ON-HOME-Scheduling. Eine Änderung besteht darin, den Unterprogrammaufruf `LSVM_EXE_LOOP` der Laufzeitbibliothek durch den Unterprogrammaufruf `LSVM_EXE_DISTR_ONCE_LOOP` zu ersetzen. Außerdem finden vor der IF-Abfrage Wertzuweisungen statt. Diese haben denselben Charakter wie in Abschnitt 5.4 bei der DISTRIBUTE-Direktive, da sie die Verteilungsarten verschlüsseln⁵. In dem Unterprogramm `LSVM_EXE_DISTR_ONCE_LOOP` wird dann auch die Verteilung des Templates durchgeführt, so es denn nicht bereits auf ein Prozessorfeld verteilt worden ist. Ansonsten wird das Unterprogramm `LSVM_EXE_LOOP` intern aufgerufen und das ON-HOME-Scheduling durchgeführt, wie im vorherigen Abschnitt bereits beschrieben wurde.

An einem Beispiel wird gezeigt, wie diese Umsetzung funktioniert. Das Programm

⁵Die dynamischen Arbeitsverteilungsstrategien werden ähnlich verschlüsselt wie eine CYCLIC-Verteilung.

```

        PROGRAM EXAMPLE_DISTRIBUTE_ONCE
        INTEGER I,J
CSVM$ PRIVATE::I,J
CSVM$ PROCESSORS::P(10,5)
CSVM$ TEMPLATE::T(10000,500)

        ...

CSVM$ PDO(LOOPS(I,J),
CSVM$+   STRATEGY(DISTRIBUTE_ONCE(T(I,J), (BLOCK,BLOCK))),
CSVM$+   PROCESSORS(P(1:10, 1:5))
CSVM$+ )
        DO I = 1, 10000
            DO J = 1, 500
                ....
            ENDDO
        ENDDO

        ...
    END

```

wird transformiert zu

```

        PROGRAM EXAMPLE_DISTRIBUTE_ONCE
        INTEGER I,J

C *** Deklaration der unten benoetigten Integer-Variablen
        INTEGER SVM_FIELD1(7), SVM_FIELD2(7,3), SVM_BLOCK_NUM, SVM_CURRENT_BLOCK,
                SVM_CONT_ITER, SVM_FROM, SVM_TO, SVM_BLOCK_NUM_1,
                SVM_CURRENT_BLOCK_1, SVM_CONT_ITER_1, SVM_FROM_1, SVM_TO_1

CSVM$ PRIVATE::I,J
CSVM$ PROCESSORS::P(10,5)
CSVM$ TEMPLATE::T(10000,500)

        ...

C *** generierte Wertzuweisungen, die fuer das DISTRIBUTE noetig sind
        SVM_FIELD2(1,1) = 3
        SVM_FIELD2(1,2) = 1
        SVM_FIELD2(1,3) = 10
        SVM_FIELD2(2,1) = 3
        SVM_FIELD2(2,2) = 1
        SVM_FIELD2(2,3) = 5

C *** Falls das Template noch nicht verteilt worden ist, dann verteile es in dem
C *** Unterprogramm und fahre dann fort wie beim Unterprogramm I_SVM_EXE_LOOP
        IF (I_SVM_EXE_DISTR_ONCE_LOOP(1, 0, SVM_FIELD1, 2, SVM_FIELD2, 3) .EQ. 1) THEN

C ***     Lese fuer die I-Schleife die Anzahl der Segmente ein
        SVM_BLOCK_NUM = I_SVM_GET_NUM_BLOCKS(1, 1)

```

```

C ***      Initialisierung der in der WHILE-Schleife benoetigten Laufvariablen
           SVM_CURRENT_BLOCK = 0

C ***      Anfang der in Fortran simulierten WHILE-Schleife
1          CONTINUE

C ***      Inkrementiere Laufvariable in WHILE-Schleife
           SVM_CURRENT_BLOCK = SVM_CURRENT_BLOCK+1

C ***      Lese die Grenzen fuer das Segment SVM_CURRENT_BLOCK ein; falls kein
C ***      Segment mehr vorhanden, dann setze SVM_CONT_ITER auf 0 und beende
C ***      WHILE-Schleife
           CALL SVM_GET_BOUNDS(1,1,SVM_BLOCK_NUM,SVM_CURRENT_BLOCK,1,10000,1,
+               SVM_FROM,SVM_TO,SVM_CONT_ITER)
C ***      Falls kein Segment mehr vorhanden, dann beende Schleife und springe
C ***      hinaus
           IF (SVM_CONT_ITER .EQ. 0) GOTO 4

C ***      Bearbeite Iterationen
           DO I=SVM_FROM,SVM_TO,1

C ***      Beginn der inneren J-Schleife
           SVM_BLOCK_NUM_1 = I_SVM_GET_NUM_BLOCKS(1, 2)
           SVM_CURRENT_BLOCK_1 = 0
2          CONTINUE
           SVM_CURRENT_BLOCK_1 = SVM_CURRENT_BLOCK_1+1
           CALL SVM_GET_BOUNDS(1,2,SVM_BLOCK_NUM_1,SVM_CURRENT_BLOCK_1
+               1,500,1,SVM_FROM_1,SVM_TO_1,SVM_CONT_ITER_1)
           IF (SVM_CONT_ITER_1 .EQ. 0) GOTO 3
           DO J=SVM_FROM_1,SVM_TO_1,1

C ***      urspruenglicher Schleifenkoerper
C          ....

           ENDDO
           GOTO 2
3          CONTINUE
           ENDDO
           GOTO 1
4          CONTINUE
           CALL SVM_PSET_RESET_APS()
           ENDIF

           ...
           END

```

Vor der IF-Abfrage werden, wie oben beschrieben, die Wertzuweisungen ausgeführt, die für die Ausführung des Distributions nötig sind. Danach wird in der IF-Abfrage das Unterprogramm `I_SVM_EXE_DISTR_ONCE_LOOP` aufgerufen. Die übrigen Transformationen wurden bereits im vorherigen Abschnitt beschrieben.

Kapitel 6

Implementierung

Die Umsetzung der SVM-Fortran Direktiven erfolgt in zwei Stufen. In der ersten Stufe liest der SVM-Präprozessor das Programm mit den Direktiven ein und wandelt das Fortran-Programm um, wie in Kapitel 5 beschrieben. Dieser Präprozessor ist in der Programmiersprache C [37] geschrieben und hat insgesamt mehr als 17500 Zeilen Code.

In einer zweiten Stufe werden die Funktionen, die in dem modifizierten Fortran-Programm aufgerufen werden, in einer Laufzeitbibliothek zur Verfügung gestellt. Diese Bibliothek ist ebenfalls in C geschrieben und umfaßt etwa 14000 Zeilen Code.

Im folgenden wird prinzipiell beschrieben, wie die Umsetzung der Scheduling-Direktiven in den Präprozessor integriert worden ist. Danach werden einige Besonderheiten der Implementierung erläutert.

6.1 Integration in einen Präprozessor für SVM-Fortran

Zunächst wird die Vorgehensweise beschrieben, um aus einem sequentiellen Fortran-Programm ein ausführbares Programm für einen Parallelrechner zu erzeugen. Danach wird dann die Arbeitsweise des Präprozessors vorgestellt.

In Abbildung 6.1 ist die Vorgehensweise bei der Parallelisierung eines Programms und dessen Ausführung beschrieben. Zunächst modifiziert der Benutzer das Fortran-Programm durch Hinzufügen von Direktiven. Dabei sind mannigfaltige Entscheidungen zu treffen, wie beispielsweise eine Schleife parallelisiert werden soll, ob die Verteilung in einem Template abgespeichert werden soll oder ob Programmteile parallel ausgeführt werden können. Danach wird das Programm mit den Annotationen dem Präprozessor übergeben.

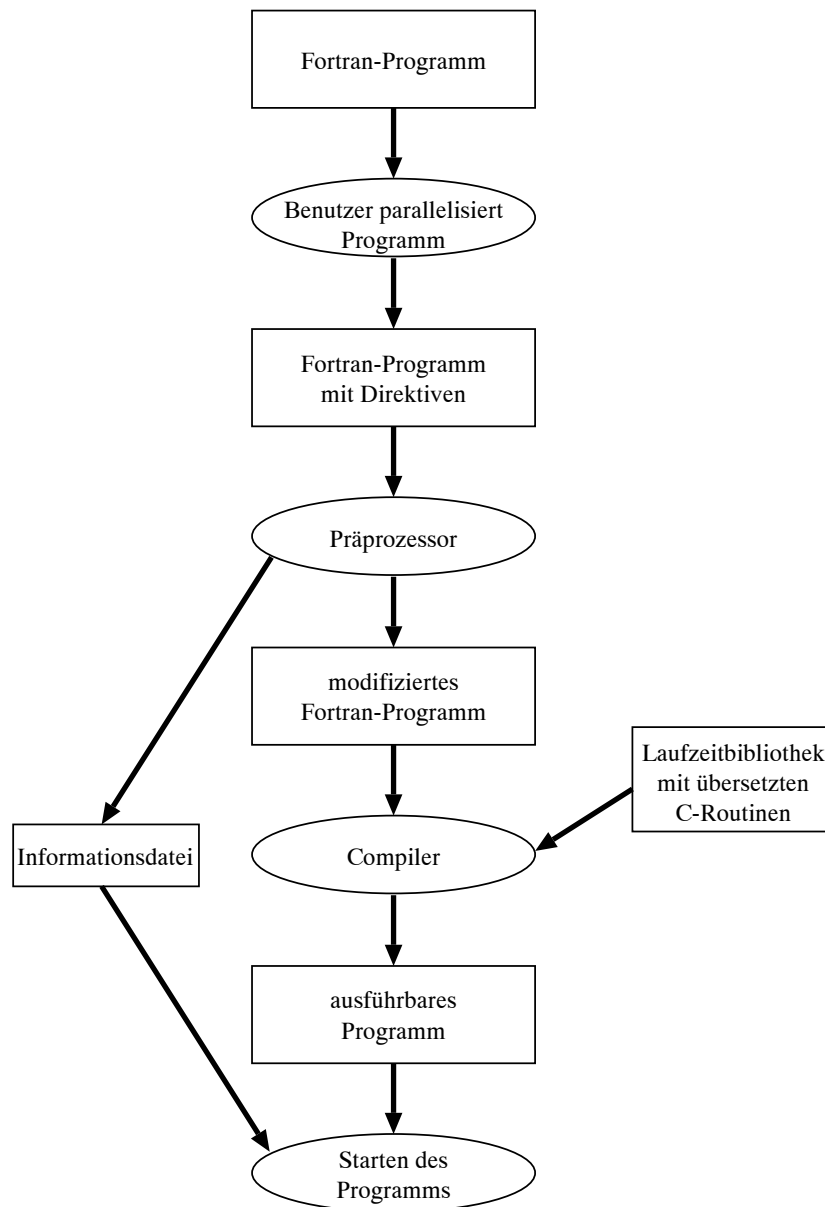


Abbildung 6.1: Vorgehensweise bei der Parallelisierung eines Programms und dessen Ausführung

Der Präprozessor wandelt die Direktiven in Fortran-Code um, der Aufrufe von Routinen der Laufzeitbibliothek enthält. Er liefert entsprechende Fehlermeldungen, falls Syntaxfehler oder Inkonsistenzen beim Gebrauch der Direktiven wie z.B. Verwendung eines Templates, das nicht deklariert worden ist, vorliegen.

Abbildung 6.2 gibt eine grobe Übersicht über die Vorgehensweise im Präprozessor. Im Detail ist dies:

- Zunächst wird das ganze Programm eingelesen und in eine interne Repräsentation, einem abstrakten Syntaxbaum mit Symboltabelle, überführt. Dabei wird

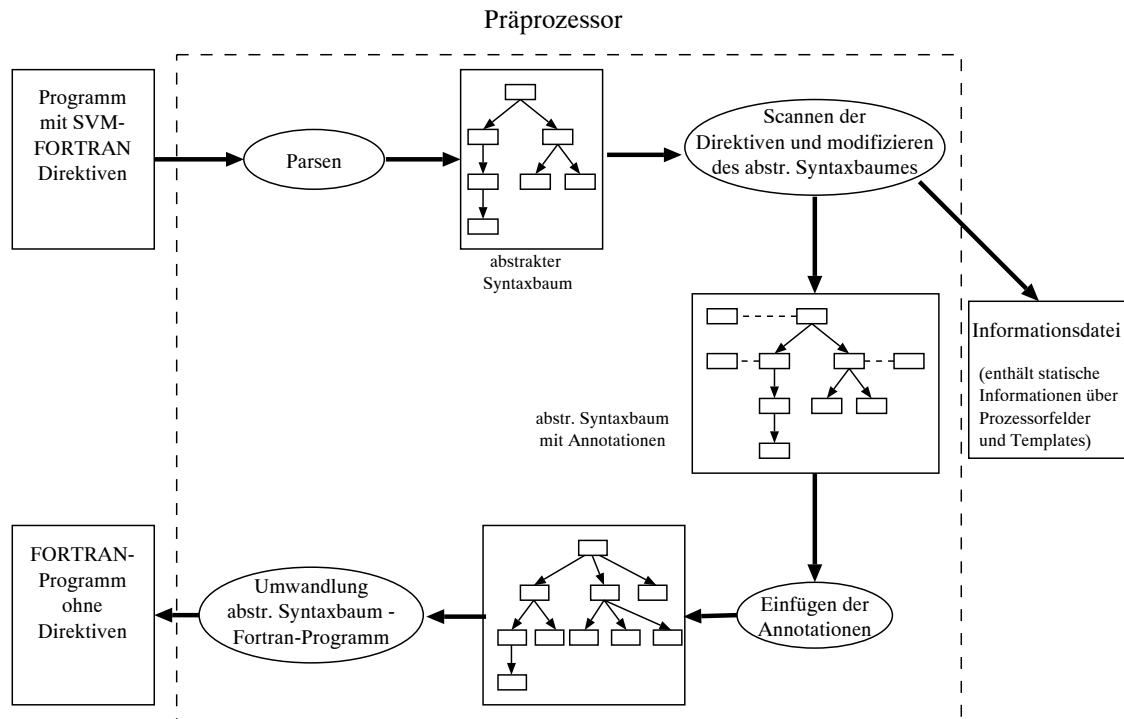


Abbildung 6.2: Arbeitsweise des Präprozessors

das Tool PAFF, ausführlich in [9] beschrieben, verwendet. PAFF liefert zahlreiche Funktionen, mit denen man den abstrakten Syntaxbaum manipulieren kann.

- Der abstrakte Syntaxbaum wird ganz durchlaufen. Wenn der Präprozessor dabei auf eine SVM-Fortran-Direktive stößt, dann werden folgende Schritte ausgeführt:
 1. Die Direktive wird normalisiert, d.h. es werden alle Leerzeichen aus der Direktive entfernt.
 2. Es wird das erste Schlüsselwort der Direktive eingelesen und die entsprechende Funktion, die diese Direktive behandelt, aufgerufen.
 3. In der Funktion wird die Umsetzung der Direktive durchgeführt. Informationen, die in Abschnitt 6.2.1 beschrieben werden, werden in eine Informationsdatei hineingeschrieben. Wenn jedoch Änderungen im Fortran-Code wie z.B. Hinzufügen von Wertzuweisungen und Funktionsaufrufen durchgeführt werden müssen, dann werden diese neuen Programmzeilen mit Prioritäten an einen Knoten des abstrakten Syntaxbaumes angehängt, ohne ihn zu verändern. Diese Prioritäten geben an, ob die neue Programmzeile vor oder hinter den Knoten eingefügt werden soll, oder falls mehrere Programmzeilen bei einem Knoten eingefügt werden sollen, wie dann die Einfügereihenfolge zwischen diesen Programmzeilen ist. Dieser abstrakte Syntaxbaum mit Annotationen ist nötig, da zu unterschiedli-

chen Zeitpunkten neue Programmzeilen an einen Knoten angehängt werden können, aber zwischen diesen eine bestimmte Reihenfolge in jedem Fall eingehalten werden muß, unabhängig vom Einfügezeitpunkt.

- Die Annotationen werden der Priorität entsprechend vor oder hinter den Knoten in den abstrakten Syntaxbaum eingefügt.
- Der abstrakte Syntaxbaum wird in ein Fortran-Programm umgewandelt.

Nachdem der Präprozessor das neue Fortran-Programm erzeugt hat, wird es von einem Fortran-Compiler, der die Laufzeitbibliothek mit einbindet, übersetzt. Dabei entsteht ein ausführbares Programm, das der Benutzer dann auf einem SVM-Rechner laufen lassen kann. Dieses Programm liest am Anfang die Informationsdatei ein und verarbeitet die dort enthaltenen Daten, bevor es das transformierte ursprüngliche Programm ausführt.

6.2 Beschreibung der Implementierung

Die Scheduling-Direktiven werden im Präprozessor in der Datei *workshar.c* umgesetzt, die knapp 5000 Zeilen C-Code umfaßt. Dort wird die Umwandlung des abstrakten Syntaxbaumes in einen abstrakten Syntaxbaum mit Annotationen für die in Kapitel 5 beschriebenen Direktiven durchgeführt. Auf eine detailliertere Darstellung der Funktionsweise der einzelnen Funktionen in dieser Datei wird an dieser Stelle verzichtet.

Weiterhin wird auch die Informationsdatei *svm_workshar.dat* beim Aufruf des Präprozessors erzeugt. Diese Datei beinhaltet Informationen zu Prozessorfeldern, Templates und Verteilungsarten von Templates aus der DISTRIBUTE-Direktive. Diese Datei wird im Initialisierungsteil des parallelen Programms eingelesen, und die Werte werden in der entsprechenden Prozessor-, Template- oder Distribute-Tabelle abgespeichert. Die Datei dient also dazu, Informationen aus der Übersetzungsphase in die Ausführungsphase des Programms zu übertragen. Das Format dieser Datei wird im folgenden Abschnitt beschrieben.

In der Datei *r_workshar.c* befindet sich der Teil der Laufzeitbibliothek, der die Funktionen beinhaltet, die bei der Umsetzung der Scheduling-Direktiven aufgerufen werden. Diese Datei umfaßt ca. 4700 Zeilen. Wie diese Funktionen prinzipiell arbeiten, wurde in Kapitel 5 vorgestellt. Der Speicherplatzbedarf der Prozessorfeld-, Template- und Distribute-Tabelle wird in einem Abschnitt dargestellt.

6.2.1 Format der Datei *svm_workshar.dat*

Zunächst wird motiviert, warum die Datei *svm_workshar.dat* eingeführt worden ist. Anschließend wird beschrieben, welche Informationen diese Datei enthält.

Die Datei *svm_workshar.dat* wurde aus Optimierungsgründen eingeführt. Der Vorteil dieser Datei besteht darin, daß die konstanten Werte, die sich während der Programmausführung nicht ändern, bei einer Prozessorfeld- oder Template-Deklaration bzw. einer DISTRIBUTE-Direktive nur einmal während der Initialisierung des Programms in die Tabelle eingelesen werden. Falls eine der Direktiven häufiger aufgerufen wird, dann werden diese Werte bei jedem Aufruf des Unterprogramms, das die Direktive umsetzt, zur Laufzeit nicht mehr eingelesen, was einen Zeitgewinn bedeutet. Die Werte befinden sich schon in der Tabelle und müssen dort nur noch ausgelesen werden. Die fehlenden Parameter werden, wie in Kapitel 5 erwähnt, den Unterprogrammen über Variablen übergeben.

Die Datei *svm_workshar.dat* beinhaltet Informationen zu:

1. Distribute-Spezifikation:

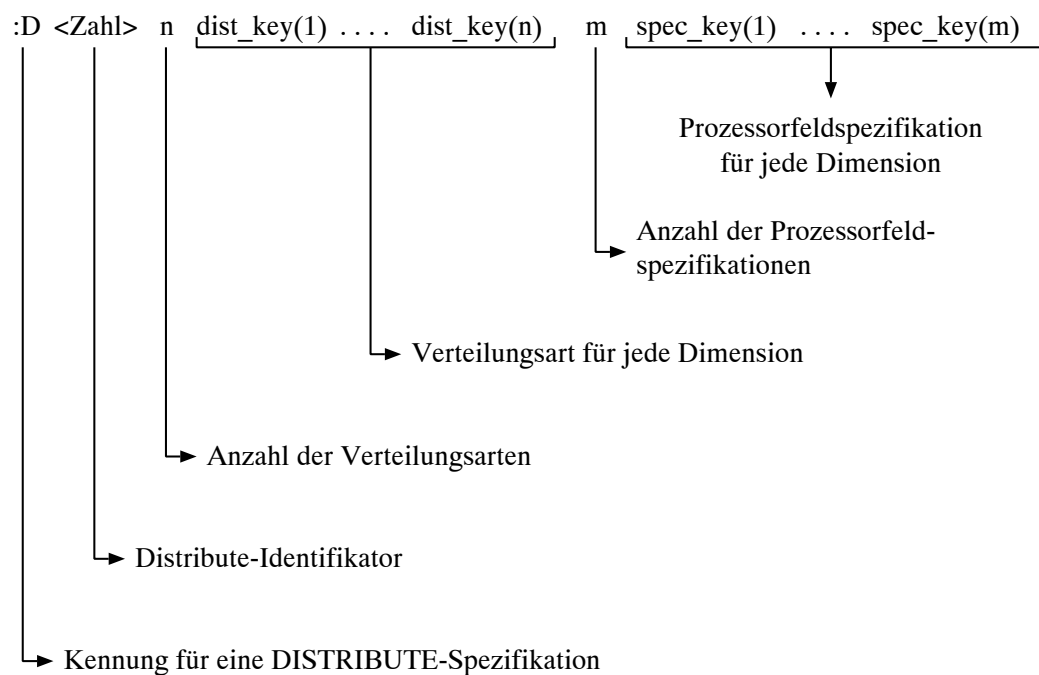


Abbildung 6.3: DISTRIBUTE-Spezifikation in der Datei *svm_workshar.dat*

Bei einer DISTRIBUTE-Direktive werden die Verteilungsart und die Spezifikation des Prozessorfeldes, die zusammen in Abschnitt 5.4 erklärt wurden, in der Datei abgespeichert. In einer Direktive seien n Verteilungsarten und m Spezifikationen für Prozessorfelder gegeben. Die Funktionen $\text{dist_key}(i)$ mit $1 \leq i \leq n$ und $\text{spec_key}(j)$ mit $1 \leq j \leq m$ liefern den Schlüssel für die Verteilungsarten bzw. die Prozessorfeldspezifikationen in der entsprechenden Dimension. In der Datei wird dann für jede DISTRIBUTE-Direktive eine Zeile mit dem Format aus Abbildung 6.3 generiert.

2. Prozessorfelder:

Zu jedem Prozessorfeld wird in der Datei eine Prozessorspezifikation gespeichert, die den Prozessoridentifikator, den Prozessorfeldnamen, den Unterprogrammnamen der Deklarationsumgebung des Prozessorfeldes, den Rang, den Anfangs- und Endprozessor im APS, auf die das Prozessorfeld abgebildet wird, die Anzahl konstanter Dimensionen sowie die Unter- und Obergrenze für die konstanten Dimensionen enthält.

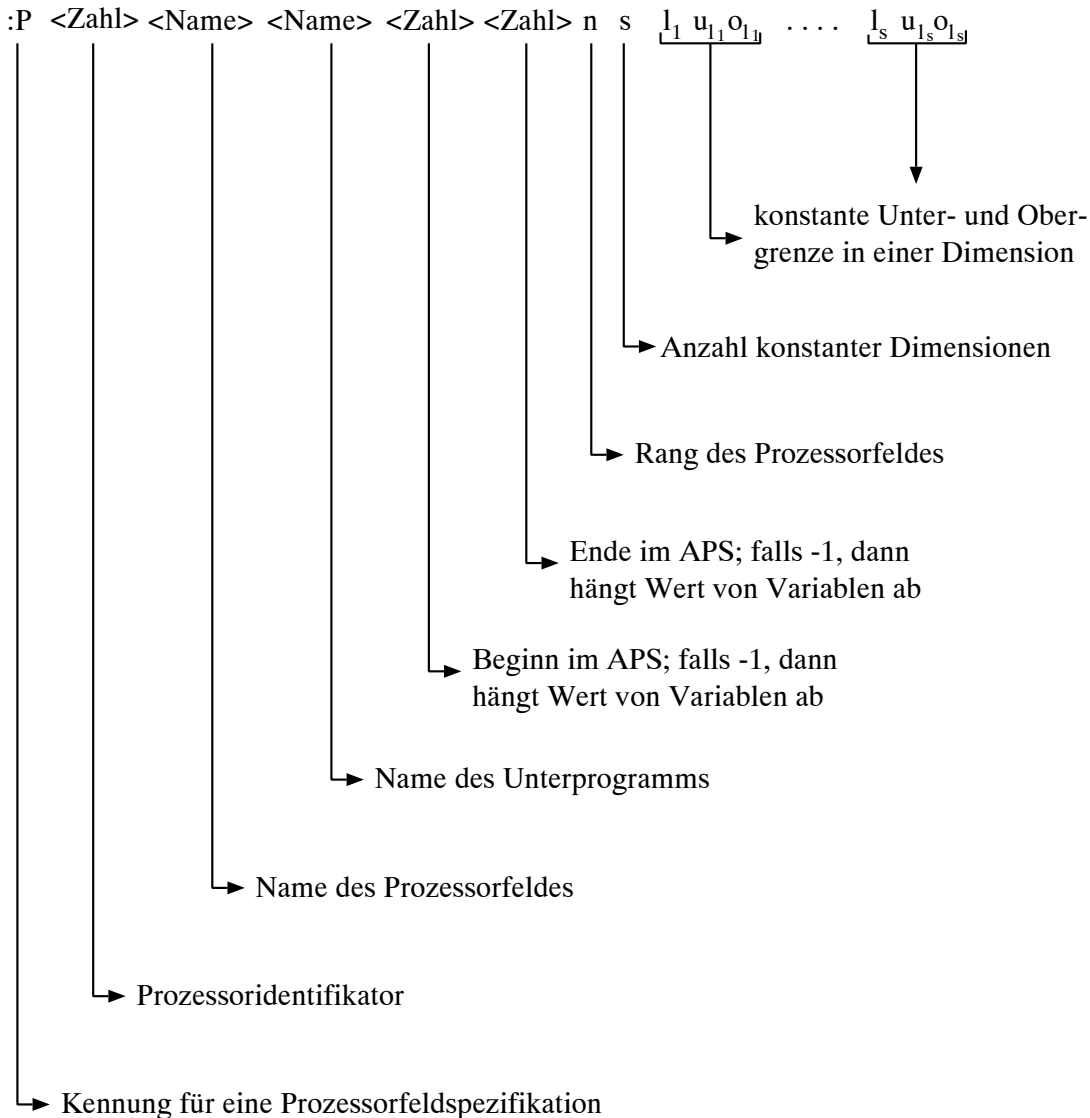


Abbildung 6.4: Prozessorspezifikation in der Datei *svm_workshar.dat*

Ein Prozessorfeld mit Rang n , dessen Ausdehnung in den Dimensionen formal dargestellt sei durch $(u_1 : o_1, u_2 : o_2, \dots, u_n : o_n)$, wird mit der PROCESSORS-Direktive deklariert. Sei s mit $0 \leq s \leq n$ die Anzahl der Dimensionen, in der sowohl die Unter- als auch Obergrenze konstant ist, und $l_1, \dots, l_s$ spezifiziere diese Dimensionen, dann werden diese konstanten Dimensionen und die

oben erwähnten Informationen in der Datei in einer Zeile abgelegt. Das Format kann aus Abbildung 6.4 entnommen werden. Diese Werte werden in der Initialisierungsphase des Programms in die lokale Prozessortabelle eines jeden Prozessors eingetragen.

3. Templates:

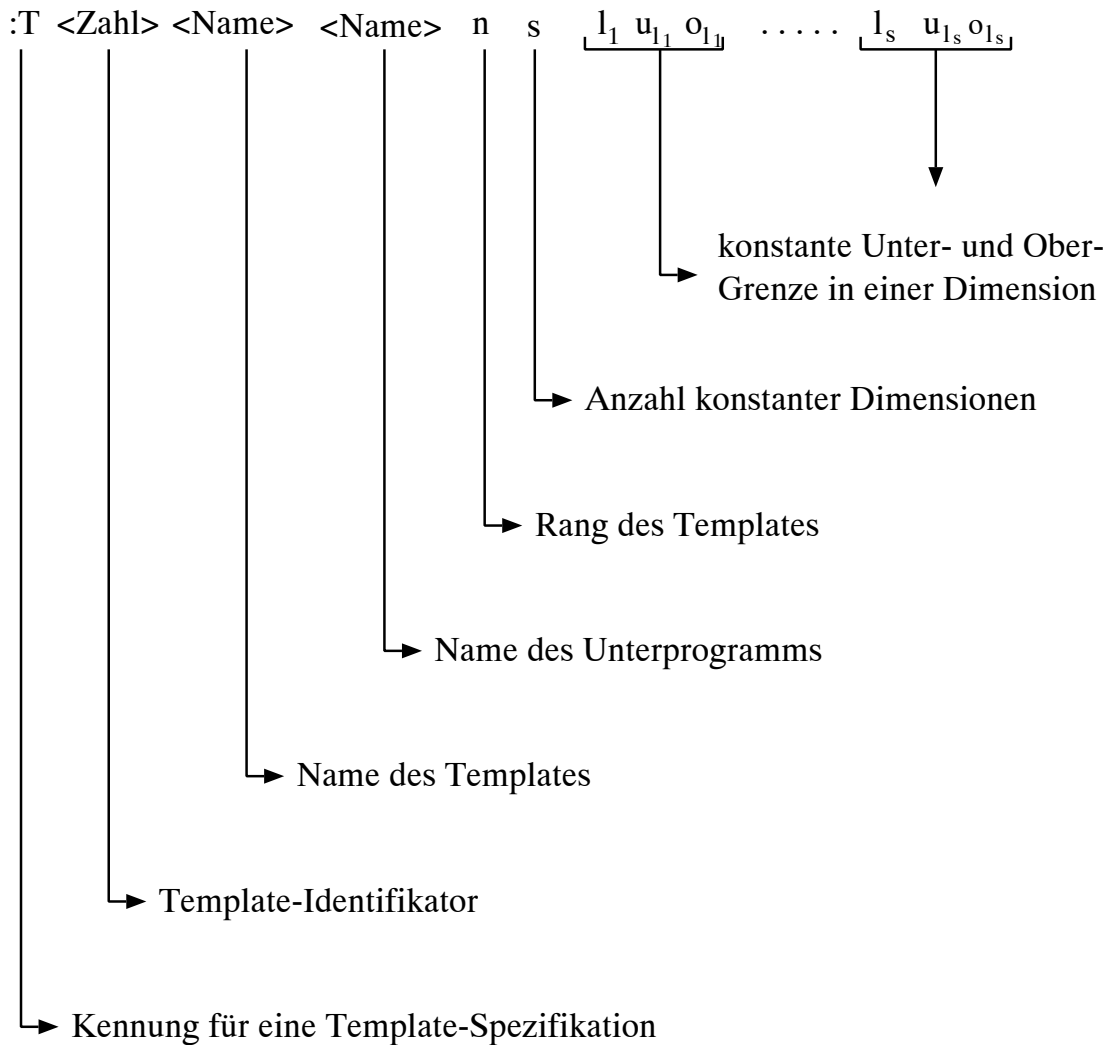


Abbildung 6.5: Template-Spezifikation in der Datei *svm_workshar.dat*

Bei den Templates sieht ein Eintrag ähnlich aus wie bei den Prozessorfeldern, es wird in der Datei der Template-Identifikator, der Template-Name, der Name des Unterprogramms, in der die Template-Deklaration ausgeführt wurde, der Rang des Templates, die Anzahl der konstanten Template-Dimensionen sowie die Unter- und Obergrenze für diese Dimensionen abgespeichert. Die konstanten Dimensionen seien genau so dargestellt wie bereits für die Prozessorfelder

erläutert. In der Datei wird die in Abbildung 6.5 abgebildete Zeile für jede Template-Deklaration generiert.

6.2.2 Speicherplatzbedarf der internen Tabellen

In Tabelle 6.1 wird der Speicherplatz für ein Prozessorfeld, ein Template und eine DISTRIBUTE-Direktive angegeben. Der Bedarf bei einem Eintrag in der Prozessorfeldtabelle und in der Distribute-Tabelle ist konstant, bei einer Template-Tabelle variiert dieser Wert hingegen. Der Grund hierfür ist die Speicherung der Verteilung eines Templates auf ein Prozessorfeld. Bei der *irregular distribution* werden dort dynamische Strukturen angelegt, die die Verteilung festhalten. Der Speicherplatzbedarf dafür kann in [50] nachgeschlagen werden. Bei den in dieser Diplomarbeit beschriebenen Verteilungsstrategien ist der Speicherplatzbedarf konstant und beträgt pro Eintrag in der Template-Tabelle 696 Byte.

Art der Tabelle	Speicherplatzbedarf für einen Eintrag
Prozessortabelle	168 Bytes
Template-Tabelle	696 Bytes
Distribute-Tabelle	64 Bytes

Tabelle 6.1: Speicherplatzbedarf eines Eintrags in der Prozessorfeld-, Template- und Distribute-Tabelle

6.3 Ergebnisse und Messungen anhand einiger Testbeispiele

Am Anfang wird der Parallelrechner, auf dem diese Messungen durchgeführt wurden, und das Betriebssystem beschrieben. Danach werden die Beispiele und Meßergebnisse kurz vorgestellt.

6.3.1 Der Parallelrechner Intel Paragon XP/S 10 mit ASVM

Die Intel Paragon XP/S ist ein massiv paralleler Rechner der Firma Intel SSD. Ausführliche Informationen zum Rechner können aus den Handbüchern des Herstellers oder aus [10] beispielsweise entnommen werden. Die wesentlichen Daten sind in Tabelle 6.2 festgehalten.

Das SVM-System für die Intel Paragon ist das ASVM (*Advanced Shared Virtual Memory*) der Firma Intel SSD, das aus einer Diplomarbeit an der Universität München

	Intel Paragon XP/S 10
Anzahl Rechenknoten	140
Prozessortyp	Intel i860XP
Verbindungsstruktur	2-dim. Gitter
Leistung pro Prozessor	75 MFLOPS (8 Byte Genauigkeit)
Speicher pro Prozessor	32 MByte
Taktzeit	20 ns
Peak Performance des Gesamtsystems	10,5 GFLOPS
Betriebssystem	OSF/1 AD
Version	1.2

Tabelle 6.2: Daten des verwendeten massiv-parallelen Rechners

hervorgegangen ist. Das ASVM basiert auf der strikten Kohärenzstrategie. Die Seitengröße ist fest vorgegeben und beträgt 8KB. Ausführliche Informationen zu ASVM kann man aus [59] entnehmen.

Die Übersetzung der Laufzeitbibliothek wurde mit dem Intel-Compiler `icc`, Version 4.5, und der Optimierungsstufe `-O4 -Knoieee` und die des modifizierten Fortran-Programms mit dem Intel-Compiler `if77`, Version 4.5, und der Optimierungsstufe `-O4 -Knoieee` durchgeführt.

6.3.2 Zeitaufwand der einzelnen Direktiven

In diesem Abschnitt wird der *Overhead*, der durch die Direktiven entsteht, für jede einzelne Direktive aufgezeigt.

Direktive	Ausführungszeit in μs
PROCESSORS	13 - 31
UNDEF	12,8
DESTROY	10

Zu den in dieser Tabelle abgebildeten Direktiven sei anzumerken, daß die Anzahl der Prozessoren, die sich im APS befinden, das bei der Direktivenausführung aktiv ist, die Ausführungszeiten nicht beeinflußt, da die durchgeführten Berechnungen unabhängig von der Anzahl der Prozessoren sind. Bei der PROCESSORS-Direktive hängen die Ausführungszeiten vom Rang des Prozessorfeldes ab, weil in der Direktive von dem Rang abhängig mehr Einträge in der Prozessorfeldtabelle vorgenommen werden müssen. Daher ist dort ein Zeitbereich angegeben, in dem sich die Prozessorfelddeklaration bewegt.

Bei der `TEMPLATE`- und `CREATE`-Direktive hängt der Zeitbedarf, dargestellt in Abbildung 6.6, von der Anzahl der Prozessoren im APS der Template-Deklarationsumgebung ab. Die Zeit setzt sich aus einem konstanten Zeitbedarf, der im wesentlichen beim Anlegen von Speicherplatz im gemeinsamen Speicher für das globale Flag (vgl. Abschnitt 5.1.1) entsteht, und einem von der Prozessoranzahl abhängenden variablen Zeitbedarf zusammen. Dieser wiederum setzt sich aus dem Versenden der Speicherplatzadresse des Flags im gemeinsamen Speicher an alle Prozessoren im APS sowie aus einem *Barrier* zusammen. Wegen der globalen Operationen wächst die Ausführungszeit für die `TEMPLATE`- und `CREATE`-Direktive über die Anzahl der Prozessoren logarithmisch.

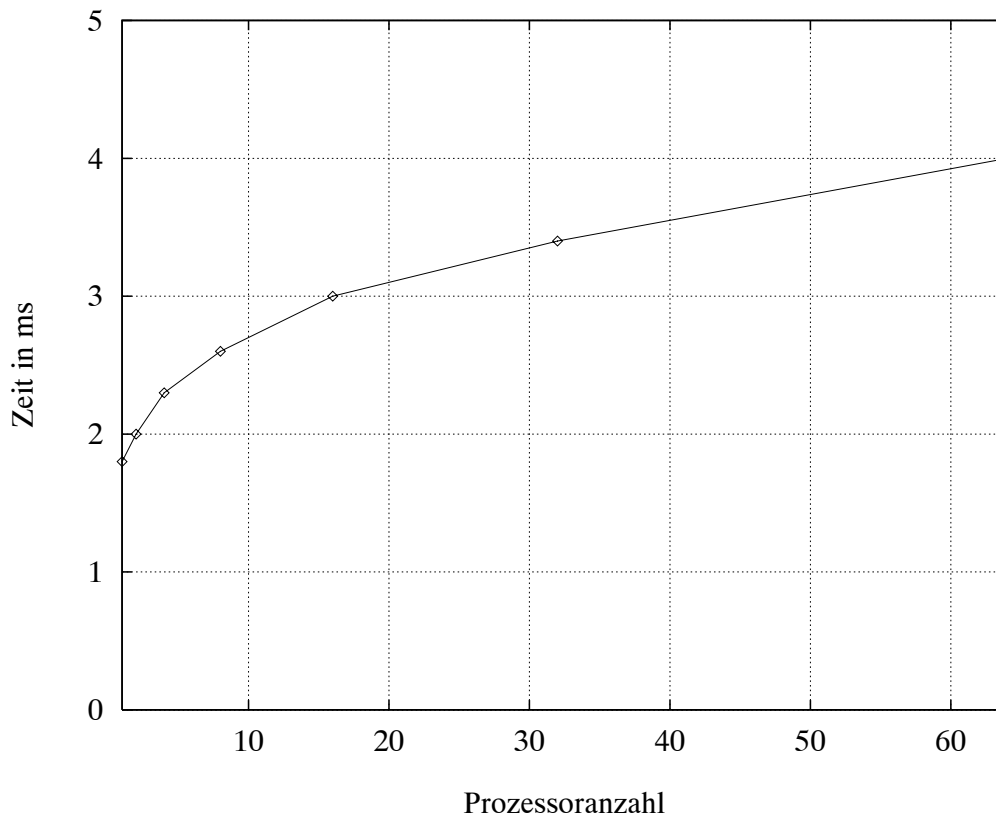


Abbildung 6.6: Zeitbedarf der `TEMPLATE`- und `CREATE`-Direktive

Bei der `DISTRIBUTE`-Direktive setzt sich der Zeitbedarf, der in Abbildung 6.7 dargestellt ist, aus einem konstanten und einem variablen Anteil zusammen, der logarithmisch mit der Anzahl der Prozessoren im APS steigt. Bei der Ausführung werden zwei *Barrier*, einer vor und einer hinter der Direktive, ausgeführt. Diese verursachen den von der Prozessoranzahl abhängenden variablen Zeitanteil.

6.3.3 Vergleich von direktem mit vordefiniertem Scheduling

Der Schwerpunkt dieses Abschnitts liegt in dem Vergleich von direktem mit vordefiniertem Schleifen-Scheduling. Es soll dargestellt werden, wieviel *Overhead* beim

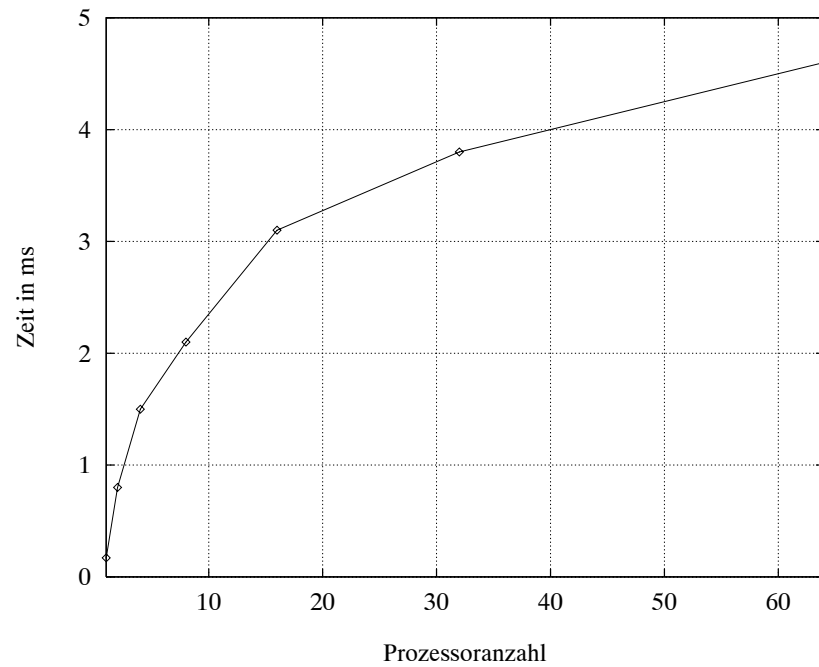


Abbildung 6.7: Zeitbedarf der DISTRIBUTE-Direktive

vordefinierten Scheduling anfällt. Dazu wird zunächst der *Overhead* für das vordefinierte Scheduling einer leeren Schleife dem des direkten Scheduling gegenübergestellt. Anschließend wird anhand zweier bekannter Anwendungen ein Vergleich der beiden Scheduling-Arten durchgeführt.

Zunächst werden die Iterationen einer leeren Schleife mit dem direkten und vordefinierten Scheduling ohne Synchronisationspunkte vor und nach der Schleife auf die Prozessoren verteilt. Der Zeitverbrauch für die Ausführung einer sequentiellen, leeren Schleife wird dabei von der Gesamtzeit subtrahiert, so daß nur der *Overhead* für das Scheduling gemessen wird. In Tabelle 6.3 wird neben dem absoluten Zeitverbrauch der zwei Scheduling-Arten auch die Anzahl der Fließkommaoperationen dargestellt. Dabei wird eine hypothetische, durchschnittliche Rechenleistung von 10 MFlops angesetzt. Dadurch soll ein Gefühl vermittelt werden, wie groß der Aufwand für die Ausführung der Schleife ist.

Typ des Scheduling	Ausführungszeit in μs	Fließkommaoperationen
direktes Scheduling	23.1	231
vordefiniertes Scheduling	2.9	29

Tabelle 6.3: Schleifen-*Overhead*

Aus Tabelle 6.3 kann entnommen werden, daß das vordefinierte Scheduling, auf eine leere Schleife angewendet, knapp acht Mal länger dauert als das direkte Scheduling. Der Grund hierfür ist, daß die ursprüngliche Schleife in eine allgemeine Schleifenstruktur transformiert wird, so daß alle Verteilungsstrategien mit dieser neuen

Schleifenstruktur durchgeführt werden können. Diese Realisierung verursacht drei Funktionsaufrufe sowie die Simulation einer WHILE-Schleife. Beim direkten Scheduling hingegen kann für eine Verteilung optimierter Code generiert werden.

Als erste Anwendung soll hier die **Matrixmultiplikation** vorgestellt werden. Es wurden zwei 512x512 bzw. 1024x1024 Matrizen miteinander multipliziert und das Ergebnis in eine dritte Matrix abgespeichert. Die einzelnen Einträge sind vom Typ REAL*8. Die Speedup-Werte wurden berechnet aus

$$Speedup = \frac{Zeit_{seq. Algorithmus}}{Zeit_{par. Algorithmus}}.$$

Die Zeiten wurden vor und nach dem Funktionsaufruf der Matrixmultiplikation gemessen. Die Matrixmultiplikation wurde als dreifach geschachtelte Schleife in der J-I-K-Form durchgeführt. Der Speedup, der sich dabei ergab, ist in Abbildung 6.8 festgehalten.

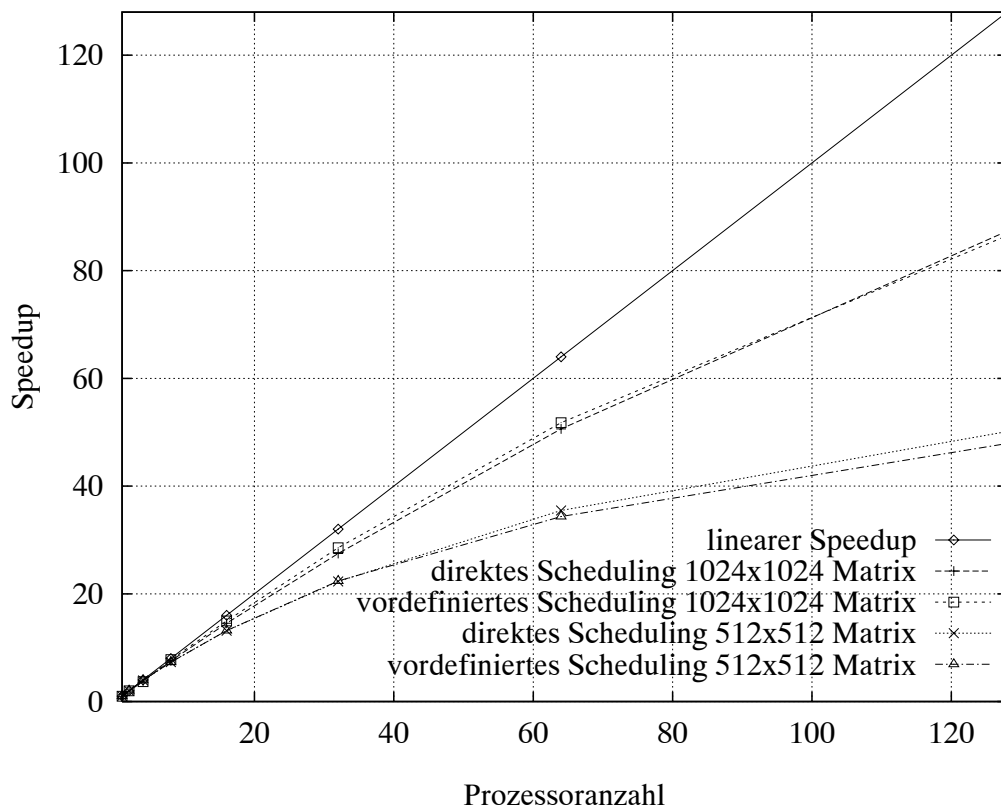


Abbildung 6.8: Speedup-Werte für die Matrixmultiplikation

Die Speedup-Werte sind bei der 512x512 Matrix schlechter als bei der 1024x1024 Matrix, da die Berechnungszeiten wesentlich kürzer sind (beispielsweise 2 Sekunden bei 64 Prozessoren) und der SVM-Fortran-*Overhead* dadurch relativ mehr Zeit verbraucht.

Die Speedup-Werte, die mit dem vordefinierten Scheduling erlangt wurden, sind mit

denen des direkten Scheduling vergleichbar. Der *Overhead*, der den Unterschied bewirkt, liegt in einer Prozessorfeld- und Template-Deklaration, der Ausführung einer DISTRIBUTE-Direktive und dem *Overhead*, der bei der Schleifenausführung, wie in Abschnitt 5.6.1 beschrieben, anfällt.

Als zweite Anwendung wird das Programm **Shallow** [56] vorgestellt, das als *Benchmark-Test* geeignet ist. Dieses Programm wird zur Analyse der Dynamik globaler Ströme in der Atmosphäre verwendet. Die Ausführungszeit des Programms wird im wesentlichen durch die Gittergröße und die Anzahl der Iterationen über das Gitter bestimmt. Im vorliegenden Testbeispiel wurde ein Gitter der Größe 2048x64 gewählt und 50 Iterationen ausgeführt.

Das Programm besteht im wesentlichen aus vier parallelen Schleifen, die während der Initialisierung einmal ausgeführt werden, und 13 weiteren parallelen Schleifen, zwischen denen sich sequentielle Programmteile befinden. In den 13 Schleifen wird auf dem Gitter ein Update der einzelnen Gitterpunkten durchgeführt. Die Speedup-Werte für das Shallow-Programm sind in Tabelle 6.9 festgehalten.

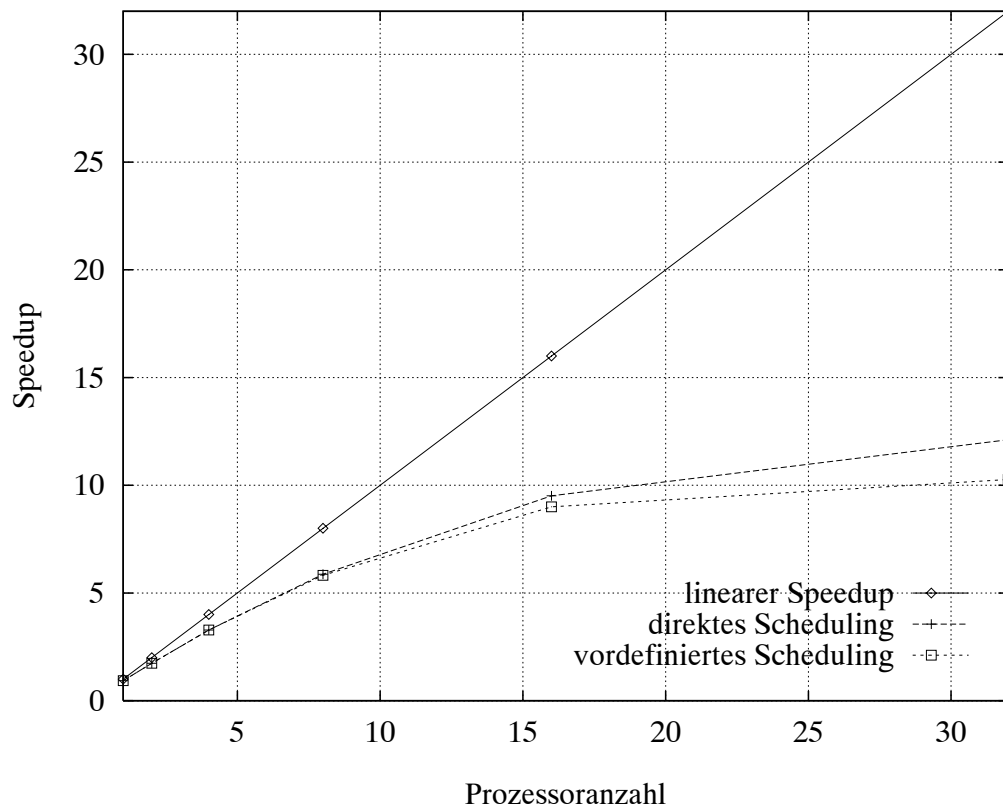


Abbildung 6.9: Speedup-Werte für Shallow

Der Grund für die schlechte Performance liegt in der Kommunikation zwischen den einzelnen Prozessoren begründet, die beispielsweise durch *Barrier*, die vor und hinter jede Schleife ausgeführt werden, hervorgerufen wird. Auch hier ist ersichtlich, daß das vordefinierte Scheduling mit dem direkten Scheduling im wesentlichen vergleichbar ist.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Realisierung und Integration des vordefinierten Scheduling in einen Präprozessor für SVM-Fortran. Für das vordefinierte Scheduling wurde ein Konzept zur Schleifentransformation entwickelt. Diese Transformation ermöglicht es, die Iterationen einer Schleife mit regelmäßigen, unregelmäßigen und dynamischen Verteilungsstrategien auf die Prozessoren zu verteilen. Die Verteilungsstrategie der Iterationen auf die Prozessoren wird dabei in einem Template gespeichert.

Für die Laufzeitadministration der Templates wurde ein Konzept vorgestellt, das auf verteilten Datenstrukturen basiert. Der Vorteil der verteilten Datenstrukturen ist ein schneller Zugriff auf lokal benötigte Daten, da sie auf jedem Prozessor lokal gespeichert sind. Diese Eigenschaft wurde bei der Speicherung der Verteilungsstrategien ausgenutzt.

Für die regelmäßigen Verteilungsstrategien eines Templates wurden Parameter für ein effizientes Scheduling ermittelt, die in den verteilten Datenstrukturen abgespeichert werden und deren Werte von Prozessor zu Prozessor unterschiedlich sind. Dabei wurde für jede Strategie eine Variante gewählt, die möglichst wenig Speicherplatz in Anspruch nimmt und beim vordefinierten Scheduling jedem Prozessor effizient erlaubt, nur die Iterationen herauszufinden, deren entsprechendes Template-Element ihm zugewiesen wurden. Für die unregelmäßigen Verteilungsstrategien wurden Schnittstellen vorbereitet, so daß die relevanten Parameter der unregelmäßigen Verteilungsstrategien im Template abgespeichert werden können.

Mit Hilfe des vordefinierten Scheduling können Schleifen mit derselben Scheduling-Strategie auch über Unterprogrammgrenzen hinweg verteilt werden. Folglich müssen Templates auch an Unterprogramme übergeben werden können. Der dazu notwendige Parameterübergabemechanismus für Templates wurde entwickelt und in das Template-Konzept integriert.

In dieser Arbeit wurden weiterhin Verfahren zur Verwaltung von Prozessorfeldern realisiert. Mit Prozessorfeldern kann der Benutzer Prozessorstrukturen definieren, die die Programmierung erleichtern, da man sie der Problemstruktur anpassen kann.

Die Verwaltung der Prozessorfelder geschieht wie bei den Templates über verteilte Datenstrukturen.

Sowohl bei der Prozessorfeld- als auch bei der Template-Verwaltung ergibt sich das Problem, daß Informationen, die zur Übersetzungszeit vorliegen, zur Laufzeit verfügbar sein müssen. Zur Lösung dieses Problems wurden zwei Verfahren entwickelt. Bei dem ersten Verfahren werden in dem ursprünglichen Programm zur Übersetzungszeit geeignete Transformationen durchgeführt, die die Informationen zur Laufzeit zur Verfügung stellen. Bei dem zweiten Verfahren werden Informationen in eine Datei geschrieben. Diese Datei wird dann vor Programmstart eingelesen. Bei der Prozessorfeld- und Template-Verwaltung wurden beide Verfahren eingesetzt.

Abschließend wurde der *Overhead* der Prozessorfeld- und Template-Verwaltung ermittelt. Es wurde sowohl der Speicherplatzbedarf der Prozessorfelder und Templates bestimmt als auch der Zeitbedarf für alle Direktiven gemessen, die bei der Prozessorfeld- oder Template-Verwaltung verwendet werden. Darüber hinaus wurden die Ursachen für den Zeitbedarf analysiert.

Der durch das vordefinierte Scheduling induzierte Aufwand wurde mit dem des direkten Scheduling verglichen. Untersuchungen ergaben, daß das vordefinierte Scheduling einen größeren Zeitaufwand mit sich bringt. Die Ursache hierfür findet man in den unterschiedlichen Anforderungen. Beim direkten Scheduling wird die Schleife in eine neue Schleifenstruktur transformiert, die für eine einzige Verteilungsart optimiert ist. Beim vordefinierten Scheduling hingegen muß die Schleife so allgemein transformiert werden, daß alle Verteilungsarten mit dieser neuen Struktur realisiert werden können.

Eine Optimierung des vordefinierten Scheduling wäre, ähnlich dem direkten Scheduling, für jede Verteilungsstrategie eine entsprechend optimierte Schleifentransformation zur Verfügung zu stellen. Um diese dann anwenden zu können, wäre zur Übersetzungszeit eine Programmanalyse notwendig, um zu entscheiden, ob die für eine Verteilungsstrategie entsprechend optimierte Schleifentransformation angewendet werden kann oder ob die ineffizientere Transformation, die für alle Verteilungsstrategien verwendet werden kann, durchgeführt werden muß.

Der Benutzer muß die Festlegung der Scheduling-Strategie – für jede parallele Schleife getrennt – manuell durchführen. Wünschenswert wäre hingegen ein Werkzeug, welches in der Lage ist, nach der Analyse einer Schleife und deren Speicherzugriffsmuster eine Schleifen-Scheduling-Strategie vorzuschlagen, die gewährleistet, daß nichtlokale Datenzugriffe minimiert werden. Der Benutzer sollte dazu lediglich spezifizieren müssen, daß eine Schleife parallel ausgeführt werden soll.

Ein langfristiges Ziel wäre es, dieses Werkzeug dahingehend zu erweitern, nicht nur eine Schleife lokal zu analysieren, sondern dies bei einer Programmeinheit oder einem Programm durchzuführen. Schleifen müßten dann nicht nur lokal untersucht werden, sondern es müßte eine für die gesamte Programmeinheit oder das gesamte Programm optimale Abbildung der Parallelität auf den Zielrechner gefunden werden.

Literaturverzeichnis

- [1] R. Allen, K. Kennedy: *Advanced Compilation for Vector and Parallel Computers*, Morgan Kaufmann Publishers Inc., San Mateo, California, 1992
- [2] American National Standards Institute: *Fortran 77*, ANSI X3.9-1978
- [3] American National Standards Institute: *Parallel Extensions for Fortran*, ANSI Technical Committee X3H5, 1993
- [4] F. André, J.-L. Pazat, H. Thomas: *Pandore: A System to Manage Data Distribution*, in Proc. ICS '90, Amsterdam, S. 380-388
- [5] ANSI Technical Committee X3H5: *Parallel Processing Model for High Level Programming Languages*, ANSI X3H5/93-SD2 Revision I, Herausgeber B. Leasure, 1993
- [6] G.H. Barnes, R.M. Brown, M. Kato, D.J. Luck, D.L. Slotnick, R.A. Stokes: *The Illiac IV computer*, IEEE Trans. Comput., Vol. C-17, 1968, S. 746-757
- [7] S. Benkner, P. Brezany, H. Zima: *Compiling High Performance Fortran in the PREPARE Environment*, in Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, 1993, S. 105-116
- [8] S. Benkner, P. Brezany, H. Zima: *Processing Array Statements and Procedure Interfaces in the PREPARE High Performance Fortran Compiler*, Technical Report ACPC/TR94-6, Universität Wien, 1994
- [9] R. Berrendorf: *Der FORTRAN-Parser PAFF als wiederverwendbares Modul für Programmier-Tools*, Jül-Spez-537, Forschungszentrum Jülich (KFA), 1989
- [10] R. Berrendorf, H.C. Burg, U. Detert, R. Esser, M. Gerndt, R. Knecht: *Intel Pargon XP/S - Architecture, Software Environment, and Performance*, Interner Bericht KFA-ZAM-IB-9409, Forschungszentrum Jülich (KFA), 1994
- [11] R. Berrendorf, M. Gerndt, W.E. Nagel, J. Prümmer: *SVM Fortran*, Interner Bericht KFA-ZAM-IB-9322, Forschungszentrum Jülich (KFA), 1993
- [12] R. Berrendorf, M. Gerndt, W.E. Nagel, J. Prümmer: *SVM-Fortran Reference Manual Version 1.1*, Forschungszentrum Jülich (KFA), 1994, Interner Bericht in Vorbereitung

- [13] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, M.-Y. Wu: *Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers*, J. Parallel and Distributed Computing, Vol. 21, 1994, S. 15-26
- [14] G. Cabillic, T. Priol, I. Puaut: *MYOAN: An Implementation of the KOAN Shared Virtual Memory on the Intel Paragon*, Technical Report N° 812, IRISA, 1994
- [15] B. Chapman, P. Mehrotra, H. Zima: *Programming in Vienna Fortran*, Technical Report ACPC/TR 92-3, Universität Wien, 1992
- [16] B. Chapman, P. Mehrotra, H. Zima: *High Performance Fortran Without Templates: An Alternative Model for Distribution and Alignment*, Technical Report ACPC/TR 93-17, Universität Wien, 1993
- [17] B. Chapman, P. Mehrotra, H. Zima: *Extending HPF For Advanced Data Parallel Applications*, Technical Report ACPC/TR94-7, Universität Wien, 1994
- [18] P. Crooks, R.H. Perrott: *An Automatic Data Distribution Generator for Distributed Memory MIMD Machines*, in Proc. Fourth Workshop on Compilers for Parallel Computers, Delft, 1993, S. 33-44
- [19] F. Darema-Rogers, V.A. Norton, G.F. Pfister: *Using a Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications*, IBM Research Report RC 11552, IBM T.J. Watson Research Center, Yorktown Heights, New York, 1985
- [20] R. Das, J. Saltz: *A Manual for PARTI Runtime Primitives Revision 2*, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1992
- [21] M. Dubois, C. Scheurich, F. Briggs: *Memory Access Buffering in Multiprocessors*, in Proc. 13. Intl. Symposium on Computer Architecture, IEEE, 1986, S. 434-442
- [22] M. Dubois, C. Scheurich, F.A. Briggs: *Synchronization, Coherence, and Event Ordering in Multiprocessors*, IEEE Computer, 1988, S. 9 - 21
- [23] M.J. Flynn: *Very High-Speed Computing Systems*, in Proc. IEEE, Vol. 54, No. 12, 1966, S. 1901-1909
- [24] I.T. Foster, K.M. Chandy: *Fortran M: A Language for Modular Parallel Programming*, Technical Report MCS-P327-0992 Revision 1. Mathematics and Computer Science Division, Argonne National Laboratory, 1993
- [25] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, M.-Y. Wu: *Fortran D Language Specification*, Rice COMP TR90-141, Rice University, Houston Texas, 1992

- [26] K. Gharachorloo, A. Gupta, J. Hennessy: *Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors*, in Proc. ASPLOS IV, ACM, 1991, S.245-256
- [27] M. Gupta, P. Banerjee: *PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers*, in Proc. ICS '93, Tokio, S. 87-96
- [28] R.G. Hackenberg: *MaX - Investigating Shared Virtual Memory*, High-Performance Computing and Networking, Lecture Notes in Computer Science, Nr. 797, Springer Verlag, 1994, S. 308-315
- [29] High Performance Fortran Forum (HPFF): *High Performance Fortran Journal of Development*, Rice University, Houston Texas, 1993
- [30] High Performance Fortran Forum (HPFF): *High Performance Fortran Language Specification*, Rice University, Houston Texas, 1993
- [31] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng: *An Overview of the Fortran D Programming System*, in Proc. Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, 1991
- [32] S. Hiranandani, K. Kennedy, C.-W. Tseng: *Compiling Fortran D for MIMD Distributed-Memory Machines*, CACM, Vol. 35, No. 8, 1992, S. 66-80
- [33] S. Hiranandani, K. Kennedy, C.-W. Tseng: *Evaluating Compiler Optimizations for Fortran D*, J. Parallel and Distributed Computing, Vol. 21, 1994, S. 27-45
- [34] D.E. Hudak, S.G. Abraham: *Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops*, in Proc. ICS '90, Amsterdam, S. 187-200
- [35] ISO: *Fortran 90*, ISO—IEC 1539: 1991 (E), 1991
- [36] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, R. G. Sheldon: *Implementing a Cache Consistency Protocol*, in Proc. 12. Intl. Symposium on Computer Architecture, IEEE, 1985, S.276-283
- [37] B.W. Kernighan, D.M. Ritchie: *The C Programming Language*, Prentice Hall, Englewood Cliffs, 1978
- [38] Kendall Square Research, Technical Summary, 1992
- [39] K. Knobe, L.D. Lukas, G.L. Steele: *Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines*, J. Parallel and Distributed Computing, Vol. 8, 1990, S. 102-118
- [40] C. Koelbel: *Compile-Time Generation of Regular Communications Patterns*, in Proc. Supercomputing '91, Albuquerque, 1991, S. 101-110
- [41] C. Koelbel, P. Mehrotra: *Supporting Shared Data Structures on Distributed Memory Architectures*, in Proc. 2nd Symposium on Principles and Practice of Parallel Programming, Seattle, WA, 1990, S. 177-186

- [42] U. Kremer, J. Mellor-Crummey, K. Kennedy, A. Carle: *Automatic Data Layout for Distributed-Memory Machines in the D Programming Environment*, Technical Report CRPC-TR93298-S, Rice University, Houston Texas, 1993
- [43] K. Li: *Shared Virtual Memory on Loosely Coupled Multiprocessors*, PhD thesis, Technical Report YALEU-RR-492, Yale University, 1986
- [44] M.A. Linn: *Untersuchungen zur Ablaufplanung bei Parallelrechnern mit virtuell gemeinsamem Speicher*, Jül-2931, Forschungszentrum Jülich (KFA), 1994
- [45] V.M. Lo: *Heuristic Algorithms for Task Assignment in Distributed Systems*, IEEE Trans. Comput., Vol. 37, No. 11, 1988, S. 1384-1397
- [46] Y. Mahéo, J.-L. Pazat: *Distributed Array Management for HPF Compilers*, Internal Publication N° 787, IRISA, Campus Universitaire de Beaulieu, Rennes Cedex, Frankreich, 1993
- [47] C.M. Müllender: *Visualisierung der Speicheraktivitäten von parallelen Programmen mit virtuell gemeinsamem Speicher*, Jül-2911, Forschungszentrum Jülich (KFA), 1994
- [48] D.M. Nicol, P.F. Reynolds: *Optimal Dynamic Remapping of Data Parallel Computations*, IEEE Trans. Comput., Vol. 39, No. 2, 1990, S. 206-219
- [49] M.G. Norman, P. Thanisch: *Models of Machines and Computation for Mapping in Multicomputers*, ACM Computing Surveys, Vol. 25, No. 3, 1993, S. 263-302
- [50] H. Otto: *Entwicklung und Analyse von dynamischen Loop-Scheduling-Algorithmen für SVM-Fortran*, Diplomarbeit in Informatik, RWTH Aachen, in Vorbereitung
- [51] J. Ramanujam, P. Sadayappan: *Compile-Time Techniques for Data Distribution in Distributed Memory Machines*, IEEE Trans. Parallel Distributed Systems, Vol. 2, No. 4, 1991, S. 472-482
- [52] J. Reinhold: *Funktionalitätserweiterung des MAX-SVM-Servers*, Diplomarbeit, Institut für Informatik, TU München, 1994
- [53] J. Saltz, K. Crowley, R. Mirchandaney, H. Berryman: *Run-Time Scheduling and Execution of Loops on Message Passing Machines*, J. Parallel and Distributed Computing, Vol. 8, 1990, S. 303-312
- [54] J. Subhlok, D.R. O'Hallaron, T. Gross: *Task Parallel Programming in Fx*, Technical Report CMU-CS-94-112, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994
- [55] J. Subhlok, J.M. Stichnoth, D.R. O'Hallaron, T. Gross: *Exploiting Task and Data Parallelism on a Multicomputer*, ACM SIGPLAN NOTICES, Vol. 28, No. 7, 1993, S. 13-22

- [56] P. N. Swarztrauber: *The Dynamics of Finite-Difference Models of the Shallow-Water equations*, J. Atmospheric Science, Vol. 32, No. 4, 1975
- [57] C.P. Thacker, L.C. Stewart, Edwin H. Satterthwaite: *Firefly: A Multiprocessor Workstation*, IEEE Trans. Comput., Vol. 37, No. 8, 1988, S.909-920
- [58] C.-W. Tseng: *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*, PhD Thesis, Rice University, Houston Texas, 1993
- [59] S. Zeisset: *Evaluation and Enhancement of the Paragon Multiprocessor's Shared Virtual Memory System*, Diplomarbeit, TU München, 1993
- [60] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald: *Vienna Fortran - A Language Specification Version 1.1*, Technical Report ACPC/TR 92-4, Universität Wien, 1992

Anhang A

Syntax von SVM-Fortran

In diesem Anhang sind die Syntaxdiagramme aller Direktiven der Sprache SVM-Fortran abgebildet. Kursiv gedruckte Bezeichner sind Nicht-Terminalsymbole und nicht kursiv gedruckte Bezeichner sind Terminalsymbole. [] kennzeichnet eine Option. Das Präfix CSVM\$ von Direktiven wird im folgenden weggelassen.

Syntax der SVM-Fortran-Direktiven:

<i>svmf-directive</i>	is	<i>declaration-directive</i>
	or	<i>execution-directive</i>
	or	<i>intrinsic-call</i>
<i>declaration-directive</i>	is	<i>distribute-directive</i>
	or	<i>processor-directive</i>
	or	<i>sharing-directive</i>
	or	<i>template-directive</i>
<i>execution-directive</i>	is	<i>atomic-update-directive</i>
	or	<i>barrier-directive</i>
	or	<i>coherence-directive</i>
	or	<i>coordinated-call-directive</i>
	or	<i>copy-directive</i>
	or	<i>create-directive</i>
	or	<i>critical-section-directive</i>
	or	<i>default-directive</i>
	or	<i>destroy-directive</i>
	or	<i>exclregion-directive</i>
	or	<i>lock-directive</i>
	or	<i>parloop-directive</i>
	or	<i>prefetch-directive</i>
	or	<i>psection-directive</i>
	or	<i>redistribute-directive</i>
	or	<i>region-directive</i>
	or	<i>undef-directive</i>
<i>intrinsic-call</i>	is	<i>intrinsic</i>

Im folgenden werden die einzelnen Direktiven alphabetisch aufgeführt. Nicht erklärte Regeln wie z.B. *int-expr* sind im ISO-Standard Fortran 90 [35] definiert.

Syntax der ATOMIC-UPDATE-Direktive:

```
atomic-update-directive  is  ATOMIC_UPDATE
                           assignment-statement
```

Syntax der BARRIER-Direktive:

```
barrier-directive   is  BARRIER [ barrier-option ]
                        or  BARRIER_CHECKIN [ barrier-option ]
                           statement-list
                           BARRIER_CHECKOUT
barrier-option      is  ON processors-name-list
```

Syntax der COHERENCE-Direktive:

```
coherence-directive is  COHERENCE ( VARS ( p-name-list ) [ , coherence-option ] )
coherence-option   is  WEAK
                        or  STRONG
```

Syntax der COORDINATED-CALL-Direktive:

```
coordinated-call-directive is  COORDINATED_CALL [ ( subprogram-name-list ) ]
```

Syntax der COPY-Direktive:

```
copy-directive   is  COPY :: p-name-list
```

Syntax der CREATE-Direktive:

```
create-directive is  CREATE :: template-name ( explicit-shape-spec-list )
```

Syntax der CRITICAL-SECTION-Direktive:

```
critical-section-directive is  CRITICAL_SECTION
                               statement-list
                               CRITICAL_SECTION_END
```

Syntax der DEFAULT-Direktive:

```
default-directive is  DEFAULT ( sharing )
sharing           is  SHARED
                        or  PRIVATE
```

Syntax der DESTROY-Direktive:

destroy-directive is DESTROY :: *template-name-list*

Syntax der DISTRIBUTE-Direktive:

distribute-directive is DISTRIBUTE *dist-strategy* :: *template-name-list*
redistribute-directive is REDISTRIBUTE *dist-strategy* :: *template-name-list*
dist-strategy is *standard-distribution*
or *indirect-distribution*
or *linked-distribution*
standard-distribution is (*dist-format-list*) ONTO *processor-section*
indirect-distribution is (*index-var-list*) ONTO *processor-section*
linked-distribution is (*index-var-list*) ONTO *template-home*
template-home is HOME (*template-name* (*templ-align-spec-list*))
dist-format is BLOCK
or CYCLIC [(*int-expr*)]
or GENERAL_BLOCK (*int-array*)
or *
processor-section is *processors-name*
or *processors-name* (*dist-spec-list*)
templ-align-spec is *int-expr*
or *
index-var is *variable-name*

Syntax der EXCLUSIVE-REGION-Direktive:

exclregion-directive is EXCLUSIVE_REGION [(*region-option-list*)]
statement-list
EXCLUSIVE_REGION_END
region-option is PRIVATE (*p-name-list*)
or NOBARRIER

Syntax der INTRINSICS-Funktionen:

intrinsic is NUMPROC()
or NUMPROCSET()
or MYPROC()
or MYPROCSET()
or IPAGESIZE()
or IPROCID(*processors-name* [*int-expr-list*])
or MYPROCIND(*processors-name*, *dimension*, *variable-name*)
dimension is *int-expr*

Syntax der LOCK-Direktive:

```

lock-directive  is  LOCK ( loc-region-list )
                   statement-list
                   LOCK_END
loc-region      is  variable-name [ ( region-list ) ]

```

Syntax der PDO-Direktive:

```

parloop-directive is  PDO [ ( parloop-option-list ) ]
parloop-option    is  LOOPS ( loop-var-list )
                   or  PRIVATE ( p-name-list )
                   or  PSHARED ( p-name-list )
                   or  PSHARED_COPY ( p-name-list )
                   or  REDUCTION ( variable-name-list )
                   or  NOBARRIER
                   or  PROCESSORS ( processors-name [ ( dist-spec-list ) ] )
                   or  STRATEGY ( strat-spec )
strat-spec         is  strat-spec1-list
                   or  ON_HOME ( template-name ( templ-spec-list ) )
                   or  DISTRIBUTE_ONCE ( template-name
                                           ( templ-spec-list ), ( strat-spec1-list ) )
strat-spec1       is  *
                   or  BLOCK
                   or  REPBLOCK
                   or  CYCLIC [ ( int-expr ) ]
                   or  REPCYCLIC [ ( int-expr ) ]
                   or  GRAB [ ( int-const ) ]
dist-spec         is  [ lower-bound ] : [ upper-bound ]
                   or  int-expr
                   or  loop-var
                   or  *
templ-spec        is  int-expr
                   or  *
loop-var          is  variable-name
template-name     is  object-name

```

Syntax der PREFETCH-Direktive:

```

prefetch-directive is  PREFETCH ( VARS ( p-name-list ) [ , prefetch-option ] )
prefetch-option    is  READ
                   or  WRITE

```

Syntax der PROCESSORS-Direktive:

```

processors-directive    is  PROCESSORS :: processors-directive-stuff
processors-directive-stuff is  processors-decl-list
                           or  processors-decl RESHAPE :: processors-decl-list
processors-decl        is  processors-name ( explicit-shape-spec-list )
explicit-shape-spec    is  [ lower-bound : ] upper-bound
lower-bound           is  specification-expr
upper-bound           is  specification-expr

```

Syntax der TEMPLATE-Direktive:

```

template-directive    is  TEMPLATE [ , STATIC ] :: template-decl-list
template-decl         is  template-name ( explicit-shape-spec-list )
                           or  template-name ( deferred-shape-spec-list )
deferred-shape-spec  is  :

```

Syntax der PSECTION-Direktive:

```

psection-directive  is  PSECTION [ ( psection-option-list ) ]
                           psection-list
                           PSECTION_END
psection           is  SECTION [ section-option-list ]
                           statement-list
psection-option    is  PRIVATE ( p-name-list )
                           or  PSHARED ( p-name-list )
                           or  PSHARED_COPY ( p-name-list )
                           or  NOBARRIER
section-option      is  ON processor-region
processor-region    is  processors-name [ ( region-list ) ]
region             is  [ lower-bound ] : [ upper-bound ]
                           or  int-expr
processors-name     is  object-name

```

Syntax der REPLICATED-REGION-Direktive:

```

repregion-directive is  REPLICATED_REGION [ ( region-option-list ) ]
                           statement-list
                           REPLICATED_REGION_END
region-option      is  PRIVATE ( p-name-list )
                           or  NOBARRIER

```

Syntax der SHARING-Direktive:

```
sharing-directive  is  SHARED [ ,ALIGN ] :: p-name-list  
                   or  PRIVATE [ ,ALIGN ] :: p-name-list  
p-name            is  variable-name  
                   or  / common-block-name /
```

Syntax der UNDEF-Direktive:

```
undef-directive  is  UNDEF:: template-name-list
```


Danksagung

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

Dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich (KFA) erstellen zu können. Mein besonderer Dank gilt Herrn R. Berrendorf für die Betreuung der Arbeit und die vielen konstruktiven Diskussionen.

Weiterhin möchte ich dem SVM-Fortran-Team, insbesondere Herrn Dr. M. Gerndt und Herrn H. Otto, für die gute Zusammenarbeit danken sowie allen, die mir bei der Diplomarbeit hilfreich zur Seite standen.

Meinen Eltern danke ich an dieser Stelle für ihre Unterstützung während meiner gesamten Studienzeit, ohne die dieses Studium nicht möglich gewesen wäre.

