Forschungszentrum Jülich

Zentralinstitut für Angewandte Mathematik

Untersuchung der Programmierschnittstelle OpenMP für Parallelrechner mit gemeinsamem Speicher

Guido Nieken

Jül-3716

Untersuchung der Programmierschnittstelle OpenMP für Parallelrechner mit gemeinsamem Speicher

Guido Nieken

Untersuchung der Programmierschnittstelle OpenMP für Parallelrechner mit gemeinsamem Speicher

Das Konzept eines gemeinsamen Adreßraumes vereinfacht die Parallelisierung von Programmen, indem an Stelle von Kommunikation mittels Nachrichtenaustausch gemeinsame Datenstrukturen verwendet werden können. Dieses Prinzip bildet die Grundlage für direktivenbasierte, proprietäre Programmiermodelle verschiedener Hersteller. Die Programmierschnittstelle OpenMP stellt im Gegensatz dazu ein portables Programmiermodell zur Verfügung, welches die sukzessive Parallelisierung eines sequentiellen Programms zuläßt.

Soll OpenMP als Basisplattform für parallele Applikationen dienen, sind effiziente Implementierungen dieser Schnittstelle auf verschiedenen Hardware-Architekturen erforderlich. In dieser Arbeit werden verschiedene Implementierungen zunächst applikationsunabhängig untersucht. Zu diesem Zweck wird der durch einzelne OpenMP-Primitive verursachte Overhead analysiert. Anschließend wird die Nutzbarkeit der Programmierschnittstelle OpenMP anhand von drei Anwendungen untersucht.

Analysiert werden OpenMP-Implementierungen auf unterschiedlichen Parallelrechnerarchitekturen: Vektorrechner (Cray J90, Cray T90), cc-NUMA-Rechner (SGI Origin2000) und SMP-Rechner (IBM R50).

Evaluation of the OpenMP Programming Interface for Parallel Computers with Shared Memory

The concept of a shared address space simplifies the parallelization of programs by using shared data structures instead of explicit communication. Based on this basic principle directive-based, proprietary programming models exist from different vendors. The OpenMP Application Program Interface instead offers a portable programming model which permits successive parallelization of sequential programs.

When intended as a base platform for developing parallel applications, highly efficient implementations of OpenMP are necessary on different hardware architectures. First, we analyze OpenMP-implementations application-independent. For this purpose the overhead caused by individual OpenMP primitives is analyzed. Then usability is examined for three applications.

OpenMP implementations on different parallel computer architectures are analyzed: vector-parallel computers (Cray J90, Cray T90), cc-NUMA computers (SGI Origin2000), and SMP-computers (IBM R50).

Inhaltsverzeichnis

Kapitel 1

Einleitung

Neben Parallelrechnern mit verteiltem Speicher, in denen Prozesse nur durch den Austausch von Nachrichten miteinander kommunizieren können, werden zunehmend Parallelrechner entwickelt, die einen gemeinsamen Adreßraum zur Verfügung stellen. Diese Unterstützung vereinfacht die Parallelisierung von Programmen, da statt Kommunikation mittels Nachrichtenaustausch gemeinsame Datenstrukturen genutzt werden können.

Mehrere Hard- und Softwarehersteller (z.B. Cray, SGI, SUN) haben Programmierschnittstellen entwickelt, welche auf Direktiven basieren und für die Programmierung von Parallelrechnern mit gemeinsamem Speicher eingesetzt werden. Die Direktiven der Programmiermodelle besitzen jedoch eine unterschiedliche Syntax und stellen zum Teil auch eine andere Funktionalität bereit. Dies hat zur Folge, daß ein parallelisiertes Programm nicht auf verschiedenen Hardware-Architekturen übersetzt und ausgeführt werden kann, ohne es vorher an den entsprechenden Dialekt anzupassen.

Mit dem OpenMP Application Program Interface (API) wurde eine Programmierschnittstelle definiert, die es ermöglicht, portable Programme für Parallelrechner mit gemeinsamem Speicher zu entwickeln. An der Spezifikation des OpenMP APIs waren mehrere Hersteller, wie z.B. Compaq, HP, IBM, Intel, SGI/Cray und SUN beteiligt, so daß eine breite Unterstützung durch die verschiedenen Hersteller zu erwarten ist. Die Programmierschnittstelle OpenMP basiert auf Direktiven und erlaubt eine sukzessive Parallelisierung eines sequentiellen Programms. Die Spezifikation des Fortran APIs [1, 3] wurde im Oktober 1997 fertiggestellt, die des C und C++ APIs [2] im Oktober 1998.

In dieser Diplomarbeit wird in Kapitel 2 zunächst auf die Programmierschnittstelle OpenMP eingegangen und das zugrundeliegende Ausführungsmodell sowie die einzelnen Direktiven, Laufzeitroutinen und Umgebungsvariablen für Fortran bzw. C und C++ beschrieben. Die proprietären Programmiermodelle der Firmen Cray [6], SGI [7, 8, 9] und SUN [5] werden in Kapitel 3 untersucht und mit dem OpenMP zugrundeliegenden Programmiermodell konzeptionell verglichen. Zu Beginn des Kapitels 4 werden die für nachfolgende Untersuchungen herangezogenen Hardware-Architekturen vorgestellt. Stellvertretend für parallele Vektorrechner werden die

Cray T90 und Cray J90 ausgewählt. Die IBM R50 dient als Referenz für SMP-Computer (*Symmetrical Multiprocessor*) und die SGI Origin2000 als Repräsentant für cc-NUMA Systeme (*cache coherent Non Uniform Memory Access*). Anschließend wird eine Methode zur Leistungsanalyse der OpenMP-Primitive vorgestellt und die erhaltenen Ergebnisse analysiert. Besondere Beachtung finden bei dieser Untersuchung parallele Schleifen, da sie bei wissenschaftlichen Anwendungen die Hauptquelle für Parallelität darstellen. In Kapitel 5 werden drei bereits parallelisierte Anwendungen beschrieben und für praxisnahe Performanceuntersuchungen herangesogen. Diese Anwendungen werden auf OpenMP-Direktiven umgestellt und analysiert. Die Firma SGI hat spezielle Direktiven zur expliziten Datenverteilung auf Parallelrechnern mit verteiltem gemeinsamen Speicher entwickelt [10, 11], die auf SGI-Systemen in Kombination mit den OpenMP-Direktiven verwendbar sind. Diese Direktiven werden bei den zu untersuchenden Anwendungen ebenfalls eingesetzt und hinsichtlich ihrer Nutzbarkeit untersucht. Abschließend erfolgt im letzten Kapitel eine Zusammenfassung der im Rahmen dieser Arbeit erreichten Ergebnisse.

Kapitel 2

Die Programmierschnittstelle OpenMP

Im Verlauf dieses Kapitels wird die Spezifikation des OpenMP APIs (*Application Program Interface*) beschrieben [1, 2, 4]. Für die Sprachen C und C++ liegt das gleiche Programmiermodell zugrunde wie für Fortran, allein die Notation der einzelnen Direktiven im Programmtext ist unterschiedlich. Die Spezifikation beschreibt Direktiven, Laufzeitroutinen und Umgebungsvariablen, die es dem Benutzer erlauben, seine Programme explizit parallel ausführen zu lassen. Automatische Parallelisierungen sowie eine Überprüfung von Datenabhängigkeiten, Verklemmungen (deadlocks) oder Wettlaufsituationen (race conditions) zwischen Befehlen ist nicht vorgesehen.

2.1 Ausführungsmodell

Das OpenMP API basiert auf dem *fork-join* Modell. Dies hat zur Folge, daß ein mit Hilfe von OpenMP-Direktiven parallelisiertes Programm seine Ausführung zunächst als einzelner Prozeß beginnt. Dieser zu Beginn der Ausführung einzelne Prozeß wird Hauptthread oder *master thread* genannt. Das OpenMP-API beinhaltet Direktiven, mit denen eine parallele Region geklammert wird. Zu Beginn einer solchen parallelen Region erzeugt der Hauptthread ein *Team* bestehend aus einer festen Zahl von Threads, die dann die parallele Region gemeinsam (redundant) abarbeiten. Am Ende der parallelen Region werden die Threads eines Teams wieder zu einem einzelnen Thread zusammengeführt, die weitere Abarbeitung des Programms erfolgt sequentiell. Abbildung 2.1 zeigt, wie ein Team erzeugt wird und wie sich die Threads am Ende der parallelen Region synchronisieren und nur der Hauptthread die Ausführung fortsetzt.

Die unmittelbar innerhalb einer parallelen Region zu findenden Anweisungen befinden sich in der statischen Erweiterung des parallelen Konstruktes (wie die Barriere in Abbildung 2.1). Die innerhalb der statischen Erweiterung aufgerufenen Funktionen und Prozeduren definieren die dynamische Erweiterung des parallelen Konstruktes.

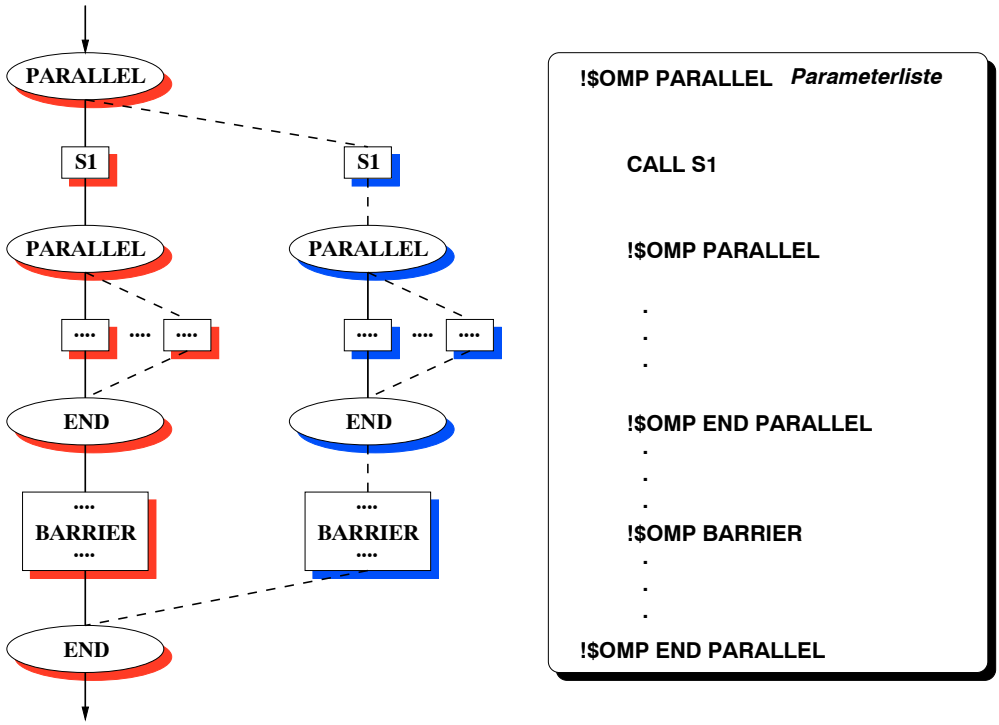

Abbildung 2.1: Fork-Join-Modell

Innerhalb eines Programms dürfen beliebig viele parallele Regionen, auch ineinander geschachtelt, vereinbart werden. Die Threads eines Teams arbeiten alle Programmbefehle, die in der dynamischen Erweiterung vorkommen, redundant ab. Trifft das Team auf ein Konstrukt zur Arbeitsteilung (*work sharing construct*), z. B. eine parallele Schleife, so wird die in der Schleife enthaltene Arbeit auf die Threads des Teams verteilt.

Das Speichermodell sieht vor, daß zu Beginn einer jeden parallelen Region mit Hilfe einer Parameterliste bestimmt werden kann, welche Variablen von den erzeugten Threads gemeinsam genutzt werden sollen und welche ausschließlich private Verwendung finden. Verändert ein einzelner Thread nun den Inhalt einer gemeinsamen Variablen, so müssen die anderen Threads davon in Kenntnis gesetzt werden. Zu diesem Zweck existieren spezielle Synchronisationsbefehle, mit denen explizit synchronisiert werden kann.

OpenMP erlaubt die Nutzung von Direktiven in Funktionen, die aus der statischen Erweiterung eines parallelen Konstruktes heraus aufgerufen werden (wie in Abbildung 2.1 die Subroutine S1). Die Direktiven in diesen Funktionen liegen in der dynamischen, jedoch nicht in der statischen Erweiterung einer parallelen Region und werden daher als verwaiste Direktiven (*orphaned directives*) bezeichnet. Dies hat zur Folge, daß große Teile eines Programms mit nur minimalen Änderungen am Programmtext parallel ausgeführt werden können. Mit Hilfe der verwaisten Direktiven wird dann die parallele Abarbeitung in den einzelnen Funktionen gesteuert.

2.2 Compilerdirektiven

Der syntaktische Aufbau der in C und C++ bzw. Fortran benutzten Direktiven ist identisch. Jede Direktive beginnt mit einer bestimmten Kennung und wird von der eigentlichen Direktive und deren Parametern gefolgt. Dieses Kennung ist eine Kennzeichnung dafür, daß nun eine zu OpenMP gehörende Direktive folgt und beginnt mit einem Kommentarzeichen bzw. mit einer Preprozessor-Kennzeichnung, um mit den Befehlen der eigentlichen Programmiersprache nicht in Konflikt zu geraten. Die Schreibweise der Direktiven ist bei C und C++ abhängig von Groß- und Kleinschreibung [2], bei Fortran jedoch nicht [1]. Desweiteren darf hinter einer solchen Direktive kein weiterer Programmcode folgen, d. h. jede Direktive steht in einer eigenen Zeile.

2.2.1 Direktivenformat

Für C und C++ ist `#pragma omp` als Kennung definiert.

```
#pragma omp Direktivenname [Parameterliste]
```

Fortran besitzt mit `!$OMP, C$OMP, *$OMP` gleich drei dieser Kennungen. Die Kennungen `C$OMP` und `*$OMP` werden nur von den Fortran Compilern akzeptiert, die die fixe Quelltextformatierung erfordern. Die Kennung `!$OMP` wird von allen Fortran Compilern akzeptiert.

```
!$OMP Direktivenname [Parameterliste]
```

Bedingte Übersetzung

Für eine bedingte Quelltextübersetzung sind ebenfalls Kennungen definiert. Diese beginnen (wie alle in OpenMP definierten Kennungen) in C und C++ mit einer Preprozessor-Kennzeichnung und in Fortran mit einem Kommentarzeichen.

Für C und C++ wird folgende Syntax verwendet.

```
#ifdef _OPENMP
    iam = omp_get_thread_num() + index;
#endif
```

Fortran benutzt folgende Syntax.

```
!$ IAM = OMP_GET_THREAD_NUM() + INDEX
```

Ein OpenMP-konformer Compiler ersetzt die entsprechenden Kommentarzeilen in der Art, daß der gewünschte Programmtext überbleibt. Bei C und C++ wird dies durch einen Vorablauf des Preprozessors erledigt, welcher abfragt ob das Preprozessor-Makro `_OPENMP` definiert ist und falls ja, den Quelltext entsprechend

vorbereitet. Bei Fortran müssen, wenn obige Syntax verwendet wird, nur die ersten zwei Zeichen gelöscht werden, um die Programmzeile zu aktivieren.

In der Fortran Spezifikation existieren drei verschiedene Kennungen für die bedingte Übersetzung. Die Kennungen `C$` und `*$` werden ausschließlich bei der fixen Quelltextnotation verwendet. Die Kennung `!$` findet sowohl in der fixen als auch in der freien Quelltextnotation Verwendung.

2.2.2 Parallele Region

Durch das folgende Konstrukt wird eine parallele Region definiert, welche von mehreren Threads parallel abgearbeitet wird. Die Anzahl der zu Beginn einer parallelen Region erzeugten Threads wird während der Ausführung dieser Region nicht verändert. Das komplette Team bearbeitet die in der dynamischen Erweiterung vorkommenden Anweisungen, wobei einzelne Threads unter Umständen (durch Direktiven gesteuert) verschiedene Programmstücke durchlaufen können.

Falls ein Thread innerhalb einer parallelen Region auf eine weitere parallele Region trifft, so erzeugt dieser Thread ein neues Team mit sich selbst als Hauptthread der neuen parallelen Region. Standardmäßig werden solche geschachtelten parallelen Regionen jedoch nicht von einem Team sondern nur von einem einzelnen Thread ausgeführt, d. h. das Team besteht nur aus dem Hauptthread dieser geschachtelten parallelen Region. Diese Standardvorgabe läßt sich ebenfalls mittels einer speziellen Direktive oder Umgebungsvariable beeinflussen.

Die parallele Region wird in C und C++ folgendermaßen vereinbart.

```
#pragma omp parallel [Parameterliste]
```

Für Fortran gelten ebenfalls die obigen Bedingungen, einzig die Kennungen sind verschieden.

```
!$OMP PARALLEL [Parameterliste]
    parallele Region
!$OMP END PARALLEL
```

Die *Parameterliste* hinter dem Direktivennamen kann die in Tabelle 2.1 gezeigten OpenMP spezifischen Schlüsselworte enthalten. Die Bedeutung der einzelnen Parameter wird auf den folgenden Seiten detailliert beschrieben.

Für die parallele Region ist noch zu beachten, daß nur dann ein Team von Threads erzeugt wird, falls in der Parameterliste keine Bedingung (`IF(`*Ausdruck*`)`) vorhanden ist oder diese zu einem positiven Wahrheitswert evaluiert werden kann. Bei einem negativen Wahrheitswert wird diese parallele Region nur vom Hauptthread abgearbeitet.

FORTRAN	C und C++
PRIVATE*(Liste)*	private*(Liste)*
SHARED*(Liste)*	shared*(Liste)*
DEFAULT*(PRIVATE | SHARED | NONE)*	default*(shared | none)*
FIRSTPRIVATE*(Liste)*	firstprivate*(Liste)*
REDUCTION*(Operator:Liste)*	reduction*(Operator:Liste)*
IF*(Ausdruck)*	if*(Ausdruck)*
COPYIN*(Liste)*	copyin*(Liste)*

Tabelle 2.1: Parameterliste für parallele Regionen

2.2.3 Konstrukte zur Arbeitsteilung

Mit Hilfe dieser Konstrukte kann die Abarbeitung des im Konstrukt eingeschlossenen Programmcodes auf die verschiedenen Threads eines Teams verteilt werden. Die Arbeitsteilungskonstrukte können keine neuen Threads erzeugen, sondern verteilen nur die vorhandene Arbeit auf die schon vorhandenen Threads eines Teams. Eine Synchronisation der beteiligten Threads erfolgt bei solchen Konstrukten nicht zu Beginn sondern nur zum Ende. Die implizite Barriere am Ende eines solchen Konstruktes kann durch einen speziellen Parameter deaktiviert werden.

Schleifenparallelisierung

Diese Direktive ist zur Parallelisierung von Zählschleifen gedacht. Die eigentliche Schleife wird nicht verändert, sondern nur von der Direktive geklammert.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp for [Parameterliste]
    for-Schleife
```

Fortran benutzt folgende Syntax.

```
!$OMP DO [Parameterliste]
    do-Schleife
[!$OMP END DO [NOWAIT]]
```

Da bei einer Zählschleife zu Beginn die Anzahl der Durchläufe berechnet werden kann, lassen sich diese Schleifenläufe leicht auf die Threads eines Teams aufteilen. Der Schleifenzähler eines jeden Threads ist daher auch defaultmäßig als privat definiert. Mit den in Tabelle 2.2 angegebenen Parametern kann das Laufzeitverhalten der parallelisierten Schleife beeinflußt werden.

FORTRAN	**C und C++**
`PRIVATE`*(Liste)*	`private`*(Liste)*
`FIRSTPRIVATE`*(Liste)*	`firstprivate`*(Liste)*
`LASTPRIVATE`*(Liste)*	`lastprivate`*(Liste)*
`REDUCTION`*(Operator:Liste)*	`reduction`*(Operator:Liste)*
`ORDERED`	`ordered`
`SCHEDULE`*(Typ[, Größe])*	`schedule`*(Typ[, Größe])*
	`nowait`

Tabelle 2.2: Parameterliste für die Schleifenparallelisierung

Mit Hilfe des *schedule* Parameters kann spezifiziert werden, wie die einzelnen Schleifendurchläufe auf die beteiligten Threads aufgeteilt werden. Für das Scheduling sind folgende Varianten vorgesehen. Der optionale Parameter *Größe* hat, außer bei der Variante `static`, defaultmäßig den Wert 1.

`static` Die Gesamtzahl aller Schleifendurchläufe wird in gleich große Blöcke aufgeteilt und mittels eines Round-Robin Verfahrens unter Berücksichtigung der Threadnummer auf die einzelnen Threads verteilt. Mit *Größe* kann die Blockgröße explizit vorgegeben werden.

`dynamic` Die Schleifeniterationen werden in Blöcke der angegebenen *Größe* unterteilt und dynamisch auf die Threads verteilt. Dies bedeutet, daß jeder Thread der seinen Block abgearbeitet hat, sofort einen neuen Block bekommt, unabhängig von der Threadnummer.

`guided` In diesem Fall wird dynamisches Scheduling angewendet, jedoch nimmt die Größe der Blöcke exponentiell ab. Mit *Größe* kann die Blockgröße für die Threads explizit auf eine Mindestgröße beschränkt werden.

`runtime` Ist diese Schedulingvariante aktiviert, so wird eine der drei oben genannten Varianten zur Laufzeit bestimmt. Welche Variante letztendlich zum Zuge kommt, kann über eine Umgebungsvariable gesteuert werden. Bei dieser Variante darf der Parameter *Größe* nicht angegeben werden.

Parallele Blöcke

Diese Direktive ist ein Konstrukt, mit dem Anweisungsblöcke definiert werden, die dann von verschiedenen Threads eines Teams abgearbeitet werden. Jeder Block wird genau einmal von einem Thread durchlaufen.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp sections [Parameterliste]
{
[#pragma omp section]
    Anweisungsblock
[#pragma omp section
    Anweisungsblock]
⋮
}
```

Fortran benutzt folgende Syntax.

```
!$OMP SECTIONS [Parameterliste]
[!$OMP SECTION]
    Anweisungsblock
[!$OMP SECTION
    Anweisungsblock]
⋮
!$OMP END SECTIONS [NOWAIT]
```

Die angegebene *Parameterliste* kann die in Tabelle 2.3 gezeigten Parameter enthalten. Nach Abarbeitung der einzelnen Programmblöcke synchronisieren sich die Threads des Teams am Ende des durch `sections` definierten Bereichs. Diese implizite Synchronisation kann mit Hilfe des Parameters `nowait` deaktiviert werden.

FORTRAN	**C und C++**
`PRIVATE`*(Liste)*	`private`*(Liste)*
`FIRSTPRIVATE`*(Liste)*	`firstprivate`*(Liste)*
`LASTPRIVATE`*(Liste)*	`lastprivate`*(Liste)*
`REDUCTION`*(Operator:Liste)*	`reduction`*(Operator:Liste)*
	`nowait`

Tabelle 2.3: Parameterliste für Sections

Single Konstrukt

Um die Abarbeitung eines Programmblocks innerhalb eines parallelen Konstruktes auf einen einzelnen Thread zu beschränken, kann folgende Direktive verwendet werden. Der Block wird vom ersten eintreffenden Thread ausgeführt.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp single [Parameterliste]
    Anweisungsblock
```

Fortran benutzt folgende Syntax.

```
!$OMP SINGLE [Parameterliste]
    Anweisungsblock
!$OMP END SINGLE [NOWAIT]
```

Als Parameter sind bei der Single-Direktive die in Tabelle 2.4 angegebenen Parameter zulässig. Die implizite Barriere dieser Direktive kann mit der Angabe von `nowait` deaktiviert werden.

FORTRAN	**C und C++**
`PRIVATE`*(Liste)*	`private`*(Liste)*
`FIRSTPRIVATE`*(Liste)*	`firstprivate`*(Liste)*
	`nowait`

Tabelle 2.4: Parameterliste für Single

Beispiel:

```
#pragma omp parallel
{
  #pragma omp for nowait
    for( i = 0; i < n; i++ )
      a[ i ] = b[ i ] * 2;

  #pragma omp for
    for( i = 0; i < n; i++ )
      c[ i ] = b[ i ];

  #pragma omp single
    printf( "Wert von c[ 0 ]: %d", c[ 0 ] );
}
```

Da zwischen der ersten und der zweiten parallelen Schleife keine Abhängigkeiten bestehen, kann mit Hilfe des Parameters `nowait` auf eine Synchronisation am Ende der ersten Schleife verzichtet werden. Die Ausgabe des Wertes von c[0] muß warten, bis die Kopie des Feldes vollständig gebildet wurde und darf auch nur durch einen einzelnen Thread geschehen.

2.2.4 Kombinierte Konstrukte zur Arbeitsteilung

Diese kombinierten Konstrukte sind Abkürzungen für Kombinationen von Direktiven. Falls eine parallele Region nur eine einzige Direktive enthält, um Arbeit auf mehrere Threads zu verteilen, so kann die Definition der parallelen Region und die Direktive in einem einzelnen Befehl zusammengefaßt werden.

Kombination mit Zählschleife

Für diese Kombination gelten alle Regeln und Parameter wie für die einzelnen Direktiven.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp parallel for [Parameterliste]
    for-Schleife
```

Fortran benutzt folgende Syntax.

```
!$OMP PARALLEL DO [Parameterliste]
    DO-Schleife
[!$OMP END PARALLEL DO [NOWAIT]]
```

Kombination mit parallelen Blöcken

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp parallel sections [Parameterliste]
{
[#pragma omp section]
    Anweisungsblock
[#pragma omp section
    Anweisungsblock
⋮
]
}
```

Fortran benutzt folgende Syntax.

```
!$OMP PARALLEL SECTIONS [Parameterliste]
[!$OMP SECTION]
    Anweisungsblock
[!$OMP SECTION
    Anweisungsblock]
⋮
!$OMP END PARALLEL SECTIONS [NOWAIT]
```

2.2.5 Synchronisationskonstrukte

Master Konstrukt

Mit dieser Direktive kann ein Anweisungsblock markiert werden, der nur vom Hauptthread ausgeführt werden soll. Im Gegensatz zum Single-Konstrukt wird mit dieser Direktive keine Barriere impliziert.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp master
    Anweisungsblock
```

Fortran benutzt folgende Syntax.

```
!$OMP MASTER
    Anweisungsblock
!$OMP END MASTER
```

Kritische Regionen

Mit dem folgenden Konstrukt kann eine kritische Region geklammert werden, die immer nur von einem einzelnen Thread zur gleichen Zeit betreten werden darf. Um mehrere verschiedene kritische Regionen definieren zu können, ist die Vergabe von Namen für eine solche Region möglich.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp critical [(Name)]
    Anweisungsblock
```

Fortran benutzt folgende Syntax.

```
!$OMP CRITICAL [(Name)]
    Anweisungsblock
!$OMP END CRITICAL [(Name)]
```

Explizite Synchronisation aller Threads

Zur expliziten Synchronisation aller Threads in einem Team wird eine Barriere verwendet. Die Threads müssen warten, bis das Team vollständig die Barriere erreicht hat, um dann gemeinsam die Bearbeitung fortzusetzen.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp barrier
```

Fortran benutzt folgende Syntax.

```
!$OMP BARRIER
```

Atomare Anweisung

Falls eine Speicherstelle nur eine exklusive Änderung erlaubt, so kann dies durch eine atomare Anweisung erreicht werden.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp atomic
    Ausdruck
```

Fortran benutzt folgende Syntax.

```
!$OMP ATOMIC
    Ausdruck
```

Ersatzweise könnten alle atomaren Anweisungen in kritischen Regionen mit dem selben Namen geklammert werden. Diese Vorgehensweise läßt sich aber während der Übersetzung nicht so gut optimieren oder auf Instruktionen mit Hardwareunterstützung abbilden.

Eingeschränkte explizite Synchronisation

Falls die Implementierung an einer bestimmten Stelle im Programm erfordert, daß alle Threads eines Teams übereinstimmend mit den selben Daten von gemeinsamen Variablen arbeiten, so kann dies mit Hilfe der `flush` Direktive erreicht werden.

Das Problem bei der Nutzung von gemeinsamen Variablen besteht darin, daß ein Zugriff auf eine Speicherzelle nicht direkt erfolgt, sondern von Caches gepuffert werden kann. Wird eine im Cache gehaltene Speicherzelle geändert, so ist diese Änderung u. U. zunächst nur auf dem Prozessor sichtbar, dem dieser Cache gehört. Ändert sich jedoch der Inhalt einer Speicherzelle im Hauptspeicher, so ist der u. U. im Cache gehaltene Wert nicht mehr aktuell. Aus diesem Grund wird eine Möglichkeit benötigt, um die Daten zwischen Cache und Hauptspeicher zu synchronisieren.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp flush [(Liste)]
```

Fortran benutzt folgende Syntax.

```
!$OMP FLUSH [(Liste)]
```

Die zu synchronisierenden Objekte werden als Variablenliste angegeben. Bei Variablen wird der Variablenwert abgeglichen, bei Zeigern jedoch der Zeigerwert selbst. Falls keine Variablenliste angegeben wird, so werden standardmäßig alle Variablen abgeglichen, die als gemeinsam definiert sind.

Eine solche `flush` Direktive ohne Variablenliste wird implizit bei den in Tabelle 2.5 gezeigten Konstrukten ausgelöst.

FORTRAN	**C und C++**
BARRIER	barrier
CRITICAL und END CRITICAL	critical *Anfang und Ende*
END DO	for *Ende*
END PARALLEL	parallel *Ende*
END SECTIONS	sections *Ende*
END SINGLE	single *Ende*
ORDERED und END ORDERED	ordered *Anfang und Ende*

Tabelle 2.5: Konstrukte mit implizitem flush

Beispiel:

```
#pragma omp parallel private(iam, neighbor) shared(work, sync)
{
  iam = omp_get_thread_num();
  sync[ iam ] = 0;
  #pragma omp barrier

  work[ iam ] = ...;

  #pragma omp flush( work )
  sync[ iam ] = 1;
  #pragma omp flush( sync )

  neigbor = ( iam > 0 ? iam : omp_get_num_threads() ) - 1;
  while ( sync[ neigbor ] == 0 )
    #pragma omp flush( sync )

  ...= work[ neighbor ];
}
```

Nachdem die eigene Arbeit vollständig erledigt ist, wird durch die `flush-` Direktive garantiert, daß den anderen Threads die aktuellen Daten sichtbar sind. Anschließend wird signalisiert, daß eine Synchronisation mit dem Nachbarn erwünscht ist und gewartet, bis dieser seine Arbeit ebenfalls erledigt hat.

Geordnete Abarbeitung

Sollen Teile einer parallelisierten Zählschleife genau in der Reihenfolge abgearbeitet werden wie im streng sequentiellen Fall, so kann dies durch die `ordered`-Direktive erreicht werden.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp ordered
    Anweisungsblock
```

Fortran benutzt folgende Syntax.

```
!$OMP ORDERED
    Anweisungsblock
!$OMP END ORDERED
```

Dieses Konstrukt darf nur innerhalb einer parallelen Schleife eingesetzt werden, für die bereits in der Parameterliste angegeben wurde, daß eine teilweise geordnete Abarbeitung von Programmcode in dieser Schleife enthalten ist.

In einem solchen geordneten Bereich ist immer nur ein einzelner Thread zur gleichen Zeit erlaubt. Desweiteren dürfen die Threads eines Teams diesen geordneten Bereich nur in der Reihenfolge betreten, die der sequentiellen Abarbeitung entspricht. Dies bedeutet, daß kein Thread in den Bereich eintreten kann, wenn nicht alle vorherigen Schleifendurchläufe abgearbeitet sind oder garantiert werden kann, daß Threads die vorherige Schleifenläufe bearbeiten, den Bereich nicht betreten werden.

Beispiel:

```
!$OMP PARALLEL DEFAULT( SHARED )
  !$OMP DO ORDERED
    DO I = 0, 10, 1
      CALL AUSGABE( I )
    END DO
!$OMP END PARALLEL

    SUBROUTINE AUSGABE( K )
!$OMP ORDERED
    WRITE(*,*) K
!$OMP END ORDERED
    END
```

Die in der dynamischen Erweiterung liegende Ausgaberoutine sorgt für eine nach den Threadnummern geordnete Ausgabe. Die beiden Direktiven in der Routine werden auch als verwaiste Direktiven bezeichent, da sie sich nicht in der statischen Erweiterung befinden.

2.2.6 Datenkonstrukte

Das folgende Konstrukt ermöglicht es, eine globale Variable bzw. einen Common-Block als privat für einen Thread zu deklarieren. Die Angabe dieses Konstruktes erfolgt bei Fortran unmittelbar hinter der Definition der Common-Blöcke. Dies bedeutet, daß jeder neu erzeugte Thread eine nicht initialisierte Kopie der globalen

Variablen erhält, welche dann zu Beginn einer parallelen Region durch die (später erklärte) `copyin`-Direktive initialisiert werden kann. Der Hauptthread arbeitet mit den „originalen“ Variablen.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma omp threadprivate [(Liste)]
```

Fortran benutzt folgende Syntax.

```
!$OMP THREADPRIVATE [(Liste)]
```

Es ist zu beachten, daß ein Programmstück, welches nur vom Hauptthread ausgeführt werden darf, auch auf die Variablen des Hauptthreads zugreift. Die korrekte Arbeitsweise dieses Konstrukts kann nur garantiert werden, wenn die Anzahl der Threads für alle parallelen Regionen konstant ist. Dies bedeutet, daß die dynamische Threaderzeugung (`omp_set_dynamic`) deaktiviert sein muß.

2.2.7 Beeinflussung von Sichtbarkeitsbereichen in parallelen Regionen

Mit Hilfe der folgenden Parameter kann Einfluß auf die Sichtbarkeitsbereiche von Variablen genommen werden. Falls vom Programmierer nicht anders vorgesehen, sind alle in einer parallelen Region vorkommenden Variablen als gemeinsam genutzt definiert. Eine Ausnahme bilden hier nur die Schleifenzähler, welche logischerweise für jeden Thread eines Teams als privat genutzt deklariert sind.

Die Parameter werden in der Parameterliste angegeben die auf einem Konstrukt zur Definition einer parallelen Region bzw. einem Konstrukt zur Arbeitsteilung folgt.

Private Variablen

Zu Beginn einer parallelen Region wird für jeden Thread eine Variable erzeugt, die als privat angegeben ist. Für diese Variablen wird nur Speicherplatz angelegt, jedoch werden sie nicht mit Werten initialisiert. Dieses Vorgehen bewirkt, daß nun jeder Thread eine eigene (nicht initialisierte) Kopie der als privat deklarierten Variablen besitzt.

Der Zugriff auf eine private Variable innerhalb der statischen Erweiterung erfolgt immer auf die lokale Kopie der Variablen. Ein Zugriff innerhalb einer aufgerufenen Funktion ist nur dann möglich, wenn die Variable explizit als Argument an die Funktion übergeben wurde.

Laut der C- und C++- Spezifikation ist eine weitere Besonderheit bei geschachtelten parallelen Regionen zu beachten. Eine Variable, die in der äußeren Region als privat deklariert wurde, kann in der inneren Region dann vom neu erzeugten Team nur noch gemeinsam genutzt werden.

Für C und C++ wird folgende Syntax verwendet.

```
private [(Liste)]
```

Fortran benutzt folgende Syntax.

```
PRIVATE [(Liste)]
```

Private initialisierte Variablen

Die Initialisierung einer privat genutzten Variablen mit dem Wert, den die Variable vor der parallelen Region hatte, ist mit folgendem Konstrukt möglich.

Für C und C++ wird folgende Syntax verwendet.

```
firstprivate [(Liste)]
```

Fortran benutzt folgende Syntax.

```
FIRSTPRIVATE [(Liste)]
```

Private Variablen mit Endwert

Der folgende Parameter sorgt dafür, daß eine als privat deklarierte Variable nach Beendigung einer parallelen Schleife einen definierten Wert enthält. Der zugewiesene Wert entspricht dem Wert, den die Variable hätte, wenn die parallele Schleife sequentiell durchlaufen worden wäre.

Für C und C++ wird folgende Syntax verwendet.

```
lastprivate [(Liste)]
```

Fortran benutzt folgende Syntax.

```
LASTPRIVATE [(Liste)]
```

Gemeinsame Variablen

Sollen Variablen gemeinsam genutzt werden, so ist dies folgendermaßen zu erreichen.

Für C und C++ wird folgende Syntax verwendet.

```
shared [(Liste)]
```

Fortran benutzt folgende Syntax.

```
SHARED [(Liste)]
```

Vorgabewerte

Um nicht jedesmal alle Variablen, die in einer parallelen Region vorkommen, auflisten zu müssen, kann ein Vorgabewert definiert werden. Diese Vorgabe wird für alle Variablen angenommen, falls diese nicht anders explizit definiert sind.

Für C und C++ wird folgende Syntax verwendet.

```
default (shared | none)
```

Fortran benutzt folgende Syntax.

```
DEFAULT (PRIVATE | SHARED | NONE)
```

Falls `none` als Vorgabewert gewählt wird, muß für jede Variable explizit angegeben werden, ob sie gemeinsam oder privat genutzt wird.

Im Gegensatz zur Vorgabe durch die C- und C++- Spezifikation kann bei Fortran auch `PRIVATE` als Vorgabewert angegeben werden.

Automatische Reduktion

Mit Hilfe des folgenden Parameters kann erreicht werden, daß die Werte der in einer parallelen Schleife verwendeten privaten Variablen am Ende der Berechnung durch den angegebenen Operator zu einem einzigen Wert verknüpft werden.

Für C und C++ wird folgende Syntax verwendet.

`reduction (`*Operator:Liste*`)`

Fortran benutzt folgende Syntax.

`REDUCTION (`{*Operator*|*Intrinsic*}*:Liste*`)`

Die in der Liste vorkommenden Variablen müssen in der umgebenden Region so deklariert sein, daß sie für eine gemeinsame Nutzung zur Verfügung stehen. In z. B. einer parallel zu bearbeitenden Schleife, die mit dem Reduktionsparameter versehen ist, werden die für die Reduktion vorgesehenen Variablen als private Kopie für die einzelnen Threads eines Teams angelegt. Nach der parallelen Abarbeitung werden die Variablen mit den entsprechenden Operatoren verknüpft und das Ergebnis der entsprechenden gemeinsamen Variablen zugewiesen. Die für die Reduktion zulässigen Operatoren und Funktionen sind in Tabelle 2.6 aufgeführt.

Es ist zu beachten, daß der Wert einer gemeinsam genutzten Variablen, die ihren Wert erst durch eine Reduktion erhält, nach der parallelen Abarbeitung undefiniert sein kann. Dieses Verhalten kann jedoch nur dann auftreten, wenn die einzelnen Threads eines Teams durch die Angabe von `nowait` auf die Synchronisation am Schleifenende verzichten.

FORTRAN Operator	**Initialwert**	**C und C++ Operator**	**Initialwert**
+	0	+	0
*	1	*	1
-	0	-	0
.AND.	.TRUE.	&	~ 0
.OR.	.FALSE.	|	0
.EQV.	.TRUE.	^	
.NEQV.	.FALSE.		0
MAX	kleinste darstellbare Zahl		
MIN	größte darstellbare Zahl		
IAND	alle Bits 1	&&	alle Bits 1
IOR	0	||	0
IEOR	0		

Tabelle 2.6: Zulässige Operatoren bei Reduktionen

Beispiel:

```
#pragma omp parallel shared(a)
{
  #pragma omp for lastprivate(i) reduction(+:a)
    for( i = 0; i < n; i++ )
      a += b[ i ];

  c = b[ i-1 ];
}
```

Nach der parallelen Abarbeitung werden die in den privaten Kopien von `a` berechneten Werte durch die angegebene Operation + zu einem Wert aufsummiert und der gemeinsamen Variablen `a` zugewiesen. Durch die Angabe des Parameters `lastprivate` wird der Variablen `i` der Wert `n` zugewiesen. Deshalb bekommt `c` den letzten Eintrag des Feldes `b` als Wert zugewiesen. Ohne die Angabe des Parameters `lastprivate` wäre der Wert von `i` nicht vorhersehbar.

Private initialisierte Variablen eines Threads

Analog zur `firstprivate` Deklaration können auch die durch `threadprivate` deklarierten Kopien mit dem ursprünglichen Variableninhalt initialisiert werden.

Für C und C++ wird folgende Syntax verwendet.

`copyin (`*Liste*`)`

Fortran benutzt folgende Syntax.

`COPYIN (`*Liste*`)`

Die zu initialisierenden Variablen werden als *Liste* angegeben.

2.2.8 Bindungen der Direktiven

- Die Direktiven `for, single, master, barrier` bzw. `DO, SINGLE, MASTER, BARRIER` gehören immer zu der nächsten parallelen Region die sie umschließt. Es ist zu beachten, daß die angegebenen Direktiven nicht unmittelbar in der statischen Erweiterung liegen müssen, sondern auch in der dynamischen Erweiterung stehen dürfen. Falls sich gerade keine parallele Region in der Ausführung befindet, bleiben die genannten Direktiven ohne Wirkung.

- Die Direktive `ordered` bzw. `ORDERED` gehört zu der dynamischen Erweiterung des Konstruktes für Zählschleifen (`for/DO`).

- Die `atomic` bzw. `ATOMIC` Direktive garantiert exklusiven Zugriff unter Berücksichtigung aller atomaren Konstrukte in allen Threads (auch denen anderer Teams).

- Die `critical` bzw. `CRITICAL` Direktive garantiert exklusiven Zugriff unter Berücksichtigung der Namen der kritischen Regionen in allen Threads (auch denen anderer Teams).

- Eine Direktive kann nie zu einer anderen, als der sie unmittelbar umschließenden parallelen Region gehören.

2.2.9 Verschachtelte Direktiven

- Bei zwei ineinander geschachtelten parallelen Regionen sollte für die innere Region ein neues Team erzeugt werden. Standardmäßig besteht dieses neue Team aber nur aus dem bereits vorhandenen einzelnen Thread. Soll ein echtes neues Team mit mehreren Threads innerhalb einer vorhandenen parallelen Region erzeugt werden, so kann dies über eine Umgebungsvariable oder eine spezielle Funktion aktiviert werden.

- Die Direktiven `for, sections, single` bzw. `DO, SECTIONS, SINGLE` sind in der dynamischen Erweiterung der Konstrukte `critical, master` bzw. `CRITICAL, MASTER` nicht erlaubt.

- Kritische Bereiche mit dem gleichen Namen dürfen nicht ineinander verschachtelt werden.

- Barrieren sind in der dynamischen Erweiterung von `for, sections, single, master, critical` bzw. `DO, SECTIONS, SINGLE, MASTER, CRITICAL` nicht erlaubt.
- Die Abarbeitung auf den Hauptthread (`master`) zu beschränken, ist in der dynamischen Erweiterung von `for, sections, single` bzw. `DO, SECTIONS, SINGLE` nicht erlaubt.
- Die geordnete Abarbeitung ist in einer kritischen Region nicht erlaubt.

2.3 Laufzeitroutinen

Im folgenden werden die Laufzeitroutinen beschrieben, welche zur Steuerung der parallelen Programmausführung genutzt werden können.

Die Beschreibung der Routinen teilt sich in zwei Bereiche. Dies sind Routinen, die sich auf Umgebungsvariablen beziehen und Routinen zum Setzen von Sperren.

2.3.1 Steuerroutinen

Anzahl der Threads setzen

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    void omp_set_num_threads(int Anzahl)
```

Fortran benutzt folgende Syntax.

```
SUBROUTINE OMP_SET_NUM_THREADS(Anzahl)
```

Das Verhalten der Routine hängt davon ab, ob eine dynamische Threaderzeugung erlaubt ist. Bei einer dynamischen Threaderzeugung wird durch *Anzahl* die maximale Anzahl der Threads angegeben, welche für die nächsten parallelen Regionen erzeugt werden dürfen. Im anderen Fall wird bei den folgenden parallelen Regionen ein Team mit *Anzahl* Threads erzeugt. Die eingestellte Anzahl bleibt gültig, bis sie durch einen erneuten Aufruf der Routine `omp_set_num_threads` verändert wird.

Desweiteren ist zu beachten, daß diese Routine nur dann Auswirkungen hat, wenn sie außerhalb von parallelen Regionen aufgerufen wird, da die Anzahl der Threads innerhalb einer solchen Region nicht verändert werden kann.

Über das Verhalten bei geschachtelten parallelen Regionen bezüglich der Zahl der für die innere Region zu erzeugenden Threads findet sich weder in der C und C++ Spezifikation noch in der Fortran Spezifikation eine explizite Angabe.

Threadzahl im Team abfragen

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_get_num_threads(void)
```

Fortran benutzt folgende Syntax.

```
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

Diese Funktion liefert die Anzahl der Threads zurück, die zu dem aktuellen Team gehören. Dies hat zur Folge, daß der Rückgabewert bei einem Aufruf aus einem sequentiellen Programmteil heraus immer gleich eins ist.

Maximale Threadzahl Abfragen

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_get_max_threads(void)
```

Fortran benutzt folgende Syntax.

```
INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

Diese Funktion gibt die Anzahl der für die Abarbeitung eines parallelen Programms maximal zur Verfügung stehenden Threads zurück. Die tatsächlich zu benutzende Threadzahl kann durch die Routine `omp_set_num_threads` in den Grenzen 1 bis `omp_get_max_threads` eingestellt werden.

Eigene Threadnummer ermitteln

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_get_thread_num(void)
```

Fortran benutzt folgende Syntax.

```
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

Diese Funktion liefert die eigene Threadnummer zurück.

Dies bedeutet, daß bei einem Aufruf aus einer sequentiellen Region heraus Null als Wert zurückgeliefert wird, da die Numerierung der Threads bei Null beginnt. Eine Besonderheit ist bei geschachtelten parallelen Regionen zu beachten, da die inneren Regionen standardmäßig serialisiert werden. Durch die Serialisierung wird die innere Region nur von einem einzelnen Thread ausgeführt, welcher dann automatisch der

Hauptthread für diese parallele Region ist. Daher liefert die obige Funktion, wenn sie aus einer inneren (serialisierten) parallelen Region heraus aufgerufen wird, den Wert null zurück.

Prozessorzahl abfragen

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_get_num_procs(void)
```

Fortran benutzt folgende Syntax.

```
INTEGER FUNCTION OMP_GET_NUM_PROCS()
```

Diese Funktion liefert die Anzahl der für dieses Programm zur Verfügung stehenden Prozessoren zurück.

Zustandsabfrage

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_in_parallel(void)
```

Fortran benutzt folgende Syntax.

```
LOGICAL FUNCTION OMP_IN_PARALLEL()
```

Diese Funktion gibt genau dann einen positiven Wahrheitswert zurück, wenn sie aus der dynamischen Erweiterung einer parallelen Region aufgerufen wird, die sich in paralleler Abarbeitung befindet. Aus einem sequentiellen Programmbereich aufgerufen, liefert diese Funktion folglich einen negativen Wahrheitswert als Ergebnis zurück.

Eine Sonderstellung nehmen auch hier geschachtelte parallele Regionen ein, selbst wenn die innere Region serialisiert ist. In einem solchen Fall wird immer ein positiver Wahrheitswert zurückgeliefert, auch wenn der Funktionsaufruf aus der inneren Region heraus geschieht.

Threaderzeugung beeinflussen

Mit Hilfe dieser Routine wird vorgegeben, ob zur Laufzeit entschieden werden darf, wieviele Threads für eine parallele Region erzeugt werden.

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    void omp_set_dynamic(int Ausdruck)
```

Fortran benutzt folgende Syntax.

```
SUBROUTINE OMP_SET_DYNAMIC(Ausdruck)
```

Falls der angegebene Ausdruck zu einem positiven Wahrheitswert ausgewertet werden kann, wird dem Laufzeitsystem erlaubt, die Anzahl der Threads für die nachfolgenden parallelen Regionen selbst zu bestimmen. Während der Abarbeitung einer parallelen Region bleibt die Anzahl der Threads jedoch konstant. Die durch den Benutzer mit Hilfe von `set_num_threads` vorgegebene Anzahl von Threads wird in diesem Fall als obere Grenze für die Threadanzahl angesehen.

Threaderzeugung kontrollieren

Diese Funktion ist das Pendant zu der vorherigen Routine und hat als Rückgabe einen Wahrheitswert. Ist dieser Wahrheitswert positiv, so darf das System zur Laufzeit darüber entscheiden, wieviele Threads für eine parallele Region erzeugt werden.

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_get_dynamic(void)
```

Fortran benutzt folgende Syntax.

```
LOGICAL FUNCTION OMP_GET_DYNAMIC()
```

Schachtelung paralleler Regionen aktivieren

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    void omp_set_nested(int Ausdruck)
```

Fortran benutzt folgende Syntax.

```
SUBROUTINE OMP_SET_NESTED(Ausdruck)
```

Falls der angegebene Ausdruck zu einem negativen Wahrheitswert evaluiert wird, werden bei geschachtelten parallelen Regionen die inneren Regionen serialisiert. Bei einem positiven Wahrheitswert können für diese inneren Regionen neue Teams erzeugt werden. Diese neuen Teams müssen aber nicht zwangsläufig über mehrere Threads verfügen.

Schachtelung paralleler Regionen abfragen

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_get_nested(void)
```

Fortran benutzt folgende Syntax.

```
LOGICAL FUNCTION OMP_GET_NESTED()
```

Diese Funktion liefert genau dann einen negativen Wahrheitswert zurück, wenn innere parallele Regionen vom System serialisiert ausgeführt werden müssen.

2.3.2 Locks

Die nachfolgend beschriebenen Routinen werden benutzt, um einen Bereich exklusiv von nur einem Thread bearbeiten zu lassen.

Jede Funktion bekommt als Argument einen Zeiger bzw. eine Variable, welche eine Speicheradresse aufnehmen kann. Die zu setzende Sperre bezieht sich dann immer auf die durch das Argument angegebene Adresse. Dies hat zur Folge, daß zur gleichen Zeit mehrere Sperren (mit jeweils anderen Adressen) aktiv sein können. Desweiteren ist es nicht erlaubt, mit anderen als den angegebenen Routinen die Sperrvariablen zu manipulieren.

Bei der C- und C++- Spezifikation ist zu beachten, daß zwei verschiedene Arten von Sperren existieren.

Initialisierung einer Sperre

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    void omp_init_lock(omp_lock_t *Zeiger)
    void omp_init_nest_lock(omp_nest_lock_t *Zeiger)
```

Fortran benutzt folgende Syntax.

```
SUBROUTINE OMP_INIT_LOCK(Variable)
```

Diese Routinen initialisieren eine Sperre, welche dann an den Zeiger bzw. an die entsprechende Variable geknüpft ist. Die Sperre selbst wird so initialisiert, daß die Sperre „frei“ ist.

Die Sperre mit dem Zusatz `nest` besitzt zusätzlich einen Zähler, der mit dem Wert null initialisiert wird.

Da diese Routinen eine neue Sperre erzeugen, ist ein Aufruf mit einer bereits verknüpften Sperrvariablen weder sinnvoll noch zulässig.

Löschen einer Sperre

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    void omp_destroy_lock(omp_lock_t *Zeiger)
    void omp_destroy_nest_lock(omp_nest_lock_t *Zeiger)
```

Fortran benutzt folgende Syntax.

```
SUBROUTINE OMP_DESTROY_LOCK(Variable)
```

Diese Routinen lösen die Verknüpfung zwischen dem Zeiger bzw. der Variablen und geben den belegten Speicherplatz wieder frei. Nach Abarbeitung existiert die Sperrvariable nicht mehr.

Diese Routinen dürfen nur auf Sperren angewendet werden, die nicht gesetzt sind.

Sperre aktivieren

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    void omp_set_lock(omp_lock_t *Zeiger)
    void omp_set_nest_lock(omp_nest_lock_t *Zeiger)
```

Fortran benutzt folgende Syntax.

```
SUBROUTINE OMP_SET_LOCK(Variable)
```

Wird diese Routine von einem Thread aufgerufen, so darf er nur dann weiterarbeiten, wenn die angegebene Sperre erfolgreich gesetzt werden konnte. Ist die Sperre schon belegt, so muß der Thread warten, bis diese wieder frei ist.

Eine Ausnahme macht hier eine Sperre mit dem Zusatz `nest`. Möchte der Thread, der die Sperre gesetzt hat, die gleiche Sperre ein weiteres mal setzen, so darf er dies in diesem besonderen Fall tun. Die Anzahl der gesetzten Sperren wird in dem zusätzlich vorhandenen Zähler verwaltet.

Sperre deaktivieren

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    void omp_unset_lock(omp_lock_t *Zeiger)
    void omp_unset_nest_lock(omp_nest_lock_t *Zeiger)
```

Fortran benutzt folgende Syntax.

```
SUBROUTINE OMP_UNSET_LOCK(Variable)
```

Diese Routine gibt eine gesetzte Sperre wieder frei.

Auch hier erfolgt eine Sonderbehandlung für Sperren mit dem Zusatz `nest`. Der zu der jeweiligen Sperre gehörende Zähler wird solange erniedrigt, bis alle gezählten Sperren freigegeben wurden. Erst wenn dies geschehen ist, wird die Sperre entfernt.

Es ist zu beachten, daß nur der Thread, dem die Sperre gehört, einen Aufruf bezüglich der Sperre absetzen darf. Ebenso ist ein Aufruf mit einer nicht initialisierten Sperre verboten.

Sperre testen und aktivieren

Für C und C++ wird folgende Syntax verwendet.

```
#include <omp.h>
    int omp_test_lock(omp_lock_t *Zeiger)
    int omp_test_nest_lock(omp_nest_lock_t *Zeiger)
```

Fortran benutzt folgende Syntax.

```
LOGICAL FUNCTION OMP_TEST_LOCK(Variable)
```

Diese Funktionen arbeiten analog zu den Routinen zum Setzen einer Sperre, zwingen den aufrufenden Thread jedoch nicht dazu, seine Arbeit zu unterbrechen.

Die Funktionen versuchen eine Sperre zu setzen und liefern genau dann einen positiven Wahrheitswert zurück, wenn dies erfolgreich geschehen ist.

2.4 Umgebungsvariablen

Die Ausführung von parallelem Programmcode kann nicht nur durch Laufzeitroutinen beeinflußt werden, sondern auch durch Umgebungsvariablen. Die Bindung der Laufzeitroutinen ist jedoch stärker als die der Umgebungsvariablen.

Die Namen der Umgebungsvariablen müssen groß geschrieben werden. Für die zugewiesenen Werte existiert jedoch keine solche Konvention.

Schedulingvariante wählen

Die mit dieser Variablen gewählte Einstellung betrifft nur parallele Schleifen, bei denen durch Angabe des Parameters `RUNTIME` explizit die Schedulingvariante zur Laufzeit des Programms bestimmt werden darf.

Beispiel:

```
setenv OMP_SCHEDULE "GUIDED,4"
setenv OMP_SCHEDULE "dynamic"
```

Threadanzahl einstellen

Die mit dieser Variablen eingestellten Werte unterliegen den gleichen Verhaltensregeln wie bei der entsprechenden Laufzeitroutine.

Beispiel:

```
setenv OMP_NUM_THREADS 16
```

Dynamische Threadvergabe einstellen

Die Einstellungen mit Hilfe dieser Umbegungsariablen verhalten sich analog zu der entsprechenden Laufzeitroutine (`omp_set_dynamic(...)`). Der einzutragende Wahrheitswert kann TRUE oder FALSE sein.

Beispiel:

```
setenv OMP_DYNAMIC TRUE
```

Geschachtelte parallele Regionen steuern

Analog zur entsprechenden Laufzeitroutine kann mit Hilfe dieser Umgebungsvariablen die Serialisierung von inneren parallelen Regionen erzwungen werden.

Beispiel:

```
setenv OMP_NESTED FALSE
```

Kapitel 3

Systemspezifische parallele Programmiermodelle

3.1 SUN

Die Firma SUN bietet eine proprietäre Schnittstelle [5] zur Implementierung von SMP-Programmen.

Der SUN-Compiler bietet die Möglichkeit, den Quellcode durch den Programmierer explizit zu parallelisieren, indem man in den Quellcode bestimmte Direktiven einfügt und bei der Übersetzung zusätzlich den Compilerparameter `-explicitpar` angibt. Desweiteren kann der Compiler durch `-autopar` angewiesen werden, das zu übersetzende Programm automatisch zu parallelisieren. Bei dieser Variante sind keine Veränderungen am Quellcode nötig. Die beiden Möglichkeiten können durch Angabe des Parameters `-parallel` in Kombination verwendet werden.

3.1.1 Ausführungsmodell

Die Parallelisierung durch den SUN-Compiler findet auf der Ebene der Zählschleifen statt. Eine explizite Deklaration einer parallelen Region, die mehrere parallel zu bearbeitenden Schleifen in Folge enthält, ist nicht möglich. Bei ineinander geschachtelten Schleifen wird immer die am weitesten außen liegende (parallelisierbare) Schleife parallelisiert, da die Erzeugung einer weiteren parallelen Schleife in einer bereits vorhanden parallelen Schleife nicht vorgesehen ist.

Da im Compilerhandbuch keine expliziten Aussagen darüber gemacht wurden, wann die für die parallele Abarbeitung einer Schleife benötigten Threads erzeugt bzw. wieder gelöscht werden, ist an dieser Stelle noch keine gesicherte Aussage darüber möglich, ob die Threads nach einer parallelen Schleife zerstört oder geparkt werden.

3.1.2 Direktiven

Die in diesem Abschnitt beschriebenen Direktiven können in den bestehenden Quellcode vor einer parallel zu bearbeitenden Schleife eingefügt werden, was zur Folge hat, daß die Schleife durch den SUN-Compiler parallelisiert wird. Der Benutzer muß sich selbst davon überzeugen, daß keine Datenabhängigkeiten oder andere Probleme in dieser Schleife existieren, die bei einer parallelen Abarbeitung zu falschen Ergebnissen führen.

Mit dem Fortran77 Compiler können die Compilerdirektiven entweder im SUN-Format (Tabelle 3.1) oder im Cray-Format (Tabelle 3.2) benutzt werden. Der Compiler erwartet standardmäßig Direktiven im SUN-Format, kann jedoch mittels der Option `-mp=cray` angewiesen werden, das andere Direktivenformat zu akzeptieren. Im Gegensatz dazu werden von dem Fortran 90 Compiler ausschließlich Direktiven im Cray-Format akzeptiert.

Werden Direktiven im Cray-Format benutzt, so ist es erforderlich, für alle in einer zu parallelisierenden Schleife vorkommenden Variablen explizit anzugeben, ob diese privat oder gemeinsam genutzt werden sollen. Bei der Nutzung von Direktiven im SUN-Format ist dies nicht zwingend erforderlich, da folgende Standardvorgaben gesetzt sind.

- Felder sind standardmäßig als gemeinsam deklariert.
- Skalare Variablen werden als privat deklariert. Jeder Prozessor erhält seine eigene Kopie dieser Variablen.

Syntax	**Option**
`C$PAR DOALL` *Parameter*	parallelisiere nächste Schleife (falls möglich)
`C$PAR DOSERIAL`	nächste Schleife nicht parallelisieren
`C$PAR DOSERIAL*`	nächste geschachtelte Schleife nicht parallelisieren

Tabelle 3.1: Compilerdirektiven im SUN-Format

Syntax	**Option**
`!MIC$ DOALL` *Parameter*	parallelisiere nächste Schleife (falls möglich)

Tabelle 3.2: Compilerdirektive im Cray-Format

`C$PAR DOALL` *Parameter*

Diese Direktive weist den SUN-Compiler an, die nächste Schleife zu parallelisieren. Die hinter der Direktive möglichen Parameter beeinflussen die Art der parallelen Abarbeitung der jeweiligen Schleife.

`PRIVATE(`*Variablenliste*`)`
In der *Variablenliste* werden alle Skalare und Felder angegeben, die für die Dauer der parallelen Schleife privat genutzt werden sollen. Für jeden Thread werden daher Kopien der ursprünglichen Variablen angelegt.

`SHARED(`*Variablenliste*`)`
Die in der *Variablenliste* angegebenen Skalare und Felder werden für die Dauer der parallelen Schleife als gemeinsam genutzt deklariert. Dies bedeutet, daß alle Iterationen der parallelen Schleife auf die gleichen Variablen zugreifen.

`READONLY(`*Variablenliste*`)`
Dieser Parameter definiert eine Sonderform von gemeinsam genutzten Variablen, auf die ausschließlich lesend zugegriffen wird. Der Parameter `READONLY` wird zusätzlich zum Parameter `SHARED` angegeben, wobei eine gemeinsame Variable, auf die nur lesend zugegriffen wird, in den *Variablenlisten* beider Parameter eingetragen werden muß.

`STOREBACK(`*Variablenliste*`)`
Dieser Parameter wird zusätzlich zu `PRIVATE` angegeben und bezieht sich ausschließlich auf private Variablen. Da Schleifenzähler immer private Variablen sind, müssen diese nicht explizit als privat angegeben werden. Alle Variablen, die in der *Variablenliste* des Parameters `STOREBACK` angegeben sind, werden nach der parallelen Abarbeitung der Schleife mit den Werten gefüllt, welche die entsprechenden privaten Kopien in der letzten Iteration hatten.

`SAVELAST`
Dieser Parameter wirkt wie `STOREBACK`, nur mit dem Unterschied, daß alle privaten Variablen betroffen sind.

`REDUCTION(`*Variablenliste*`)`
Werden mehrere Teilergebnisse mittels einer mathematischen Operation zu einem einzigen Ergebnis verknüpft, so spricht man von einer Reduktion. Die Variable, in die die Teilergebnisse aufgesammelt werden heißt, Reduktionsvariable. Um eine Schleife zu parallelisieren, die ein solches Konstrukt enthält, wird der Parameter `REDUCTION` mit der Reduktionsvariablen in der *Variablenliste* vorangestellt. Im folgenden Beispiel werden die Werte des Feldes `a` zu einem Wert aufsummiert. Die entsprechende Reduktionsvariable heißt in diesem Fall `x`.

```
C$PAR DOALL REDUCTION(x)
  DO i = 1, n
    x = x + a(i)
  END DO
```

`SCHEDTYPE(`*Typ*`)`
Mit Hilfe des Scheduling-Parameters kann eingestellt werden, wie die einzelnen Iterationen einer Schleife in Blöcke aufgeteilt und abgearbeitet werden.

`SCHEDTYP(SELF(`*Größe*`))`
Bei einer Schleife mit 1000 Iterationen und einer angegebenen Blockgröße von 4 wer-

den jeweils 4 Iterationen auf alle verfügbaren Prozessoren verteilt. Fehlt die Größenangabe, so wählt der SUN-Compiler eine nicht näher beschriebene Angabe aus.

`SCHEDTYP(FACTORING(`m`))`
Bei k zur Verfügung stehender Prozessoren und n noch zu verteilender Iterationen werden dem nächsten Prozessor $\frac{n}{2k}$ Iterationen zugewiesen. Mit Ausnahme der letzten Zuweisung müssen jedoch mindestens m Iterationen zugewiesen werden.

`SCHEDTYP(GSS(`m`))`
Bei k zur Verfügung stehender Prozessoren und n noch zu verteilender Iterationen werden dem nächsten Prozessor $\frac{n}{k}$ Iterationen zugewiesen. Mit Ausnahme der letzten Zuweisung müssen jedoch mindestens m Iterationen zugewiesen werden.

`C$PAR DOSERIAL`

Diese Direktive bewirkt, daß die nächste Schleife bei der automatischen Parallelisierung nicht parallelisiert wird. Da sich diese Direktive immer nur auf die jeweils nächste Schleife bezieht, werden alle folgenden Schleifen vom Compiler untersucht und gegebenfalls parallelisiert. Dies gilt insbesondere auch für die Schleifen, die in der Schleife geschachtelt sind, welche von der Parallelisierung ausgenommen wurde.

`C$PAR DOSERIAL*`

Die Wirkung dieser Direktive bezieht sich, im Gegensatz zu der vorigen Direktive, auf die nächste folgende Schleife inklusive der darin geschachtelten Schleifen.

`!MIC$ DOALL[`*Parameterliste*`]`

Diese Direktive weist den SUN-Compiler an, die nächste Schleife zu parallelisieren. Die Parameter beeinflussen die Art der parallelen Abarbeitung der jeweiligen Schleife. In Tabelle 3.3 werden die zulässigen Parameter mit einer kurzen Bemerkung zusammengefaßt.

Die Bedeutung der einzelnen Parameter im Cray-Format ist analog zu der Bedeutung der entsprechenden Parameter im SUN-Format.

Die Verteilung der einzelnen Iterationen einer parallelen Schleife läßt sich auch mit den Direktiven im Cray-Format beeinflussen, jedoch werden die Blockgrößen, wie die Parameter der Direktiven, unmittelbar hinter der `DOALL`-Direktive angegeben. Die Beeinflussungsmöglichkeiten der Abarbeitung werden in Tabelle 3.4 zusammengefaßt.

Syntax	Bedeutung
PRIVATE(*v1, ...*)	Jeder Thread erhält die angegebenen Variablen als private Kopie.
SHARED(*v1, ...*)	Die Threads müssen die angegebenen Variablen gemeinsam nutzen.
MAXCPUS(*n*)	Nicht mehr als *n* Prozessoren benutzen.
SAVELAST	Die Werte der privaten Variablen im letzten Schleifendurchlauf werden gespeichert.

Tabelle 3.3: Parameter für Direktiven im Cray-Format auf SUN

Typ	Bedeutung
GUIDED	Analog zum GSS *guided self-scheduling* des SCHEDTYP im SUN-Format.
SINGLE	Jeder verfügbare Prozessor bekommt eine Iteration zugewiesen.
CHUNKSIZE[(*Größe*)]	Die Iterationen werden in Blöcke der angegebenen *Größe* unterteilt und abgearbeitet.
NUMCHUNKS[(*m*)]	Die Iterationen werden in *m* Blöcke unterteilt und abgearbeitet.

Tabelle 3.4: Parameter für Schleifenscheduling im Cray-Format auf SUN

3.1.3 Umgebungsvariablen

Die Anzahl der maximal für ein Programm zur Verfügung stehenden Threads wird mit Hilfe der Umgebungsvariablen PARALLEL eingestellt.

Die Stackgröße für jeden Thread kann über die Umgebungsvariable STACKSIZE beeinflußt werden. Standardmäßig wird für jeden Thread ein Stack der Größe 256 Kilobyte angelegt. Der Hauptstack verfügt standardmäßig über 8 Megabyte.

3.2 Cray

Im folgenden Abschnitt werden die durch den CF90 Compiler der Firma Cray gebotenen Möglichkeiten beschrieben [6].

Der CF90 Compiler ist dazu in der Lage, ein zu übersetzendes Programm automatisch zu parallelisieren. Da eine automatische Erkennung von Parallelität jedoch aufwendig und auch schwierig ist, kann der Benutzer durch in den Quelltext eingefügte Direktiven bestimmte Programmteile explizit parallelisieren.

3.2.1 Ausführungsmodell

Die Parallelisierung durch den Cray-Compiler findet nicht ausschließlich auf der Ebene der Zählschleifen statt. Um einen bestimmten Programmabschnitt explizit parallel abarbeiten zu lassen, wird dieser Abschnitt von Direktiven geklammert, welche somit eine parallele Region definieren. Für eine solche Region wird ein Team von Threads erzeugt, welches diese Region abarbeitet. Die enthaltenen Zählschleifen werden parallel ausführt. Außerhalb einer solchen parallelen Region wird das Programm von nur einem Thread abgearbeitet. Eine Schachtelung von parallelen Regionen ist nicht vorgesehen.

3.2.2 Direktiven

Die Direktiven selbst werden durch die vorangestellte Kennung `!MIC$` gekennzeichnet. Eine detaillierte Beschreibung der einzelnen Direktiven erfolgt im weiteren Verlauf dieses Abschnitts.

`!MIC$ PARALLEL[`*Parameterliste*`]`

Durch die Direktiven `PARALLEL` und `ENDPARALLEL` wird eine parallele Region geklammert. Der in dieser Region enthaltene Code wird von mehreren Prozessoren gleichzeitig ausgeführt. Treffen die Prozessoren auf eine parallel abzuarbeitende Schleife, so wird die darin enthaltene Arbeit auf die beteiligten Prozessoren verteilt.

Über die Angaben in der *Parameterliste* läßt sich festlegen, welche in der parallelen Region vorkommenden Variablen privat und welche gemeinsam genutzt werden. Die für die *Parameterliste* gültigen Angabe werden im folgenden kurz erläutert.

`AUTOSCOPE`
Alle Variablen, welche nicht explizit als gemeinsam bzw. privat genutzt deklariert sind, werden nach den Standardvorgaben behandelt.

`IF(`*Ausdruck*`)`
Falls diese Bedingung angegeben und zu einem negativen Wahrheitswert evaluiert werden kann, wird die parallele Region nicht von mehreren Prozessoren abgearbeitet. Ein Weglassen dieses Parameters hat die gleiche Auswirkung wie die Evaluierung der Bedingung zu einem positiven Wahrheitswert, da in beiden Fällen die parallele Region von mehreren Prozessoren abgearbeitet wird.

`MAXCPUS(`*n*`)`
Mit diesem Parameter wird für die parallele Region angegeben, wieviele Prozessoren maximal an der Ausführung der Region teilnehmen dürfen.

`PRIVATE(`*Variablenliste*`)`
Die in der *Variablenliste* angegebenen Variablen werden durch die einzelnen beteiligten Prozessoren privat genutzt. Dies bedeutet, daß jeder Prozessor eine Kopie der

ursprünglichen Variablen erhält und keinerlei Zugriff auf die Variablenkopien der anderen Prozessoren hat.

`SHARED(`*Variablenliste*`)`
Die in der *Variablenliste* angegebenen Variablen werden von den beteiligten Prozessoren gemeinsam genutzt. Dies bedeutet, daß jeder Prozessor Zugriff auf die gleiche Variable hat.

`!MIC$ CASE`

Enthält ein Programm eine bestimmte Anzahl von Programmblöcken, die parallel abgearbeitet werden sollen, so können diese Blöcke durch diese Direktive explizit angegeben werden.

Beispiel:

```
!MIC$ PARALLEL MAXCPUS(2)
!MIC$   CASE
          Programmblock 1
!MIC$   CASE
          Programmblock 2
!MIC$   ENDCASE
!MIC$ ENDPARALLEL
```

Im obigen Beispiel wird eine parallele Region definiert, welche von maximal zwei Prozessoren abgearbeitet werden darf. Die durch die `CASE`-Direktive geklammerten Programmblöcke 1 und 2 werden dann von jeweils einem Prozessor ausgeführt. Die Prozessoren, die ihre zugeteilte Arbeit erledigt haben, müssen am Ende des `CASE`-Konstruktes auf den letzten Prozessor warten, bevor die Programmausführung gemeinsam fortgesetzt wird.

`!MIC$ DOPARALLEL[`*Arbeitsteilung*`]`

Diese Direktive wird innerhalb einer parallelen Region eingesetzt, um den CF90-Compiler dazu anzuweisen, die folgende Zählschleife parallel abarbeiten zu lassen. Die in der Schleife vorhandene Arbeit wird auf die Prozessoren verteilt, welche die aktuelle parallele Region abarbeiten. Durch den Parameter *Arbeitsteilung* kann spezifiziert werden, wie die Arbeit auf die einzelnen Prozessoren verteilt werden soll.

Die zugehörige Direktive `!MIC$ ENDDO` erweitert die Kontrollstruktur dieses Konstruktes über die eigentliche Zählschleife hinaus und darf weggelassen werden. Falls diese Erweiterung weggelassen wird, warten alle Prozessoren am Ende der Zählschleife, bis alle Iterationen der Schleife abgearbeitet wurden. Durch Setzen dieser Erweiterung hinter die Schleife müssen sich die Prozessoren nicht am Ende der Schleife synchronisieren, sondern erst in der Zeile, in der diese Direktive steht. Dies hat zur

Folge, daß der Programmtext, welcher zwischen dem Schleifenende und der Erweiterung steht, schon von den Prozessoren ausgeführt werden kann, die ihre Arbeit erledigt haben.

Die möglichen Angaben durch den Parameter *Arbeitsteilung* inklusive ihrer Auswirkungen sind in Tabelle 3.5 kurz zusammengefaßt.

Typ	**Bedeutung**
`GUIDED[(`*vl*`)]`	`GSS` *guided self-scheduling* Zur Laufzeit werden die Schleifeniterationen abhängig von der Maschinenauslastung auf die beteiligten Prozessoren verteilt. Durch *vl* kann eine Vektorlänge angegeben werden. (Standard 1)
`SINGLE`	Jeder verfügbare Prozessor bekommt eine Iteration zugewiesen.
`CHUNKSIZE[(`*Größe*`)]`	Die Iterationen werden in Blöcke der angegebenen *Größe* unterteilt und abgearbeitet.
`NUMCHUNKS[(`*m*`)]`	Die Iterationen werden in *m* Blöcke unterteilt und abgearbeitet.
`VECTOR`	Die Anzahl der verteilten Iterationen ist abhängig von der Vektorlänge. Es werden entweder 64 oder 128 Iterationen gleichzeitig verteilt.

Tabelle 3.5: Parameter für das Schleifenscheduling auf Cray

`!MIC$ GUARD[`*n*`]`

Durch die beiden Direktiven `!MIC$ GUARD[`*n*`]` und `!MIC$ ENDGUARD[`*n*`]` wird eine kritische Region definiert, welche immer nur von einem einzelnen Prozessor betreten werden darf. Die kritischen Regionen selbst können durch Angabe einer ganzen Zahl (n) mit einem Namen versehen werden, was dazu führt, daß sich verschiedene kritische Regionen nicht mehr gegenseitig behindern.

`!MIC$ CNCALL`

Zählschleifen, welche einen Funktionsaufruf enthalten, werden vom CF90-Compiler nicht automatisch parallelisiert, da eine Überprüfung der aufgerufenen Funktionen auf Abhängigkeiten zwischen einzelnen Schleifeniterationen zu aufwendig ist.

Falls sichergestellt ist, daß die in der Zählschleife aufgerufenen Funktionen keine Abhängigkeiten bezüglich der einzelnen Schleifeniterationen enthalten, kann dies durch Voranstellen der obigen Direktive vor die Schleife angegeben werden. Der CF90-Compiler wird dann trotz des enthaltenen Funktionsaufrufs versuchen, die Schleife zu parallelisieren.

`!MIC$ MAXCPUS(`*n*`)`

Diese Direktive setzt die maximale Anzahl von Prozessoren, die für die nächsten parallelen Regionen bzw. für die nächsten `DOALL`-Konstrukte erlaubt sind. Erst durch ein erneutes Setzen der maximalen Prozessorzahl, verliert die vorherige Einstellung ihre Gültigkeit. Die Wirkung bezieht sich ausschließlich auf den aktuellen Programmabschnitt und nicht auf die aus diesem Abschnitt heraus aufgerufenen Funktionen.

Die Angabe der maximalen Prozessorzahl als Parameter hinter der Direktiven zur Definition einer parallelen Region bzw. hinter `DOALL` hat die gleiche Wirkung. Sie muß jedoch für jedes Konstrukt erneut angegeben werden.

Die Direktive `!MIC$ MAXCPUS(`*n*`)` wird mit Release 3.2 aus dem CF90 Compiler entfernt.

`!MIC$ NUMCPUS(`*n*`)`

Diese Direktive hat die gleiche Wirkung wie `MAXCPUS`, besitzt jedoch globale Wirkung. Falls mit `NUMCPUS` versucht wird, eine höhere Prozessorzahl einzustellen, als die Umgebungsvariable `NCPUS` angibt, wird diese Direktive ignoriert.

`!MIC$ PERMUTATION(`*Variablenliste*`)`

Wird ein Intergerfeld zur Indizierung eines zweiten Feldes verwendet, so könnten Probleme bei Schreibzugriffen auf das indizierte Feld entstehen, wenn das Indexfeld doppelte Werte enthält. Aus diesem Grund werden Schleifen, die solche Konstruktionen enthalten, nicht automatisch parallelisiert.

Beispiel:

```
!MIC$ PERMUTATION (IP)
  DO I = 1, N
    A(IP(I)) = A(IP(I)) + B(I)
  END DO
```

Falls angegeben wird, daß das Indexfeld keine doppelten Werte enthält, kann die Schleife im obigen Beispiel durch den CF90-Compiler automatisch parallelisiert werden.

`!MIC$ WAIT[ POINT(`*n*`) ][ SPAN(`*m*`) ]`

Falls eine zu parallelisierende Schleife einige Teile enthält, die in der gleichen Reihenfolge ausgeführt werden müssen wie im sequentiellen Fall, kann dies durch diese Direktive erreicht werden. Der sequentiell auszuführende Programmcode wird

in das Direktivenpaar `!MIC$ WAIT[ POINT(`n`) ][ SPAN(`m`) ]` und `!MIC$ SEND[ POINT(`n`) ][ IF(`*Bedingung*`) ]` geklammert.

`POINT(`n`)`
Durch die zusätzliche Angabe von `POINT` und einer ganzen Zahl für n wird der Direktiven quasi ein Name gegeben. Auf diese Weise lassen sich Paare von `WAIT` und `SEND` bilden.

`SPAN(`m`)`
Durch diese zusätzliche Angabe kann eingestellt werden, über wieviele Schleifeniterationen Datenabhängigkeiten zwischen den einzelnen Durchläufen existieren. Standardmäßig wird für m der Wert 1 angenommen. Mögliche Werte sind $1 \leq m \leq 64$.

`IF(`*Bedingung*`)`
Die angegebene *Bedingung* muß sich zu einem positiven Wahrheitswert evaluieren lassen, bevor die Programmausführung fortgesetzt werden kann.

`!MIC$ DOALL[`*Parameterliste*`][`*Arbeitsteilung*`]`

Diese Direktive ist eine Kombination aus zwei bereits vorgestellten Direktiven. Durch `DOALL` wird eine parallele Region definiert, die nur eine einzige Zählschleife beinhaltet. Die Angaben in der *Parameterliste* sind analog zu den Angabe in der Parameterliste bei der Definition von parallelen Regionen. Die Aufteilung der einzelnen Schleifeniterationen auf die beteiligten Prozessoren wird mit *Arbeitsteilung* gesteuert und ist analog zu den Angaben bei `DOPARALLEL`.

Eine Besonderheit ist bei der Parameterliste zu beachten, da ein zusätzlicher Parameter existiert. Dieser Parameter heißt `SAVELAST` und weist den Compiler an, die Werte aller privaten Variablen des letzten Schleifenlaufes in die entsprechenden Variablen (nach der Schleife) zu schreiben.

3.2.3 Umgebungsvariablen

Im folgenden Abschnitt werden einige wichtige Umgebungsvariablen beschrieben, die ebenfalls Einfuß auf die Anzahl der zu erzeugenden Threads haben.

MP_DEDICATED
Während einer dedizierten Meßzeit sollte die Umgebungsvariable MP_DEDICATED mit dem Wert 1 gesetzt werden.

MP_HOLDTIME
Falls für einen Thread keine Arbeit mehr vorhanden ist, wird er standardmäßig nach 50.000 CPU-Zyklen in den Zustand *blocked* überführt. Mit Hilfe der Umgebungsvariablen MP_HOLDTIME kann eine andere Wartezeit vorgegeben werden.

NCPUS
Die Umgebungsvariablen NCPUS stellt die bei der parallelen Ausführung eines Programms zu verwendende Anzahl der Threads ein.

3.3 SGI

Die in diesem Abschnitt gemachten Angaben beziehen sich auf den Silicon Graphics MIPSpro-Compiler ab der Release 7.2. Es existieren Compiler für Fortran 77, Fortran 90, C und C++ [7].

Im Gegensatz zu früheren Versionen des Compilers handelt es sich hier nicht mehr um einen *Source to Source* Compiler. Der parallelisierte Programmcode kann auf Wunsch wieder als Quelltext ausgegeben werden, wird aber aus der internen Darstellung des Compilers erzeugt. Dieser Compiler bietet drei Möglichkeiten, parallel ausführbaren Programmcode zu erzeugen.

- Automatische Parallelisierung durch den Compiler.
- Automatische Parallelisierung durch den Compiler mit unterstützenden Angaben über zu parallelisierende Schleifen durch den Benutzer.
- Automatische Parallelisierung durch den Compiler mit unterstützenden Angaben, sowie explizit durch den Benutzer parallelisierte Schleifen.

Bei der automatischen Parallelisierung können Statusmeldungen in eine Ausgabedatei geschrieben werden, die darüber Auskunft geben, welche Schleifen parallelisiert wurden. Bei nicht parallelisierten Schleifen wird zusätzlich der Grund für dieses Vorgehen mit angegeben. Dies gibt dem Benutzer Hinweise auf Probleme bei der Parallelisierung des eingegebenen Programmtextes.

Die Fortran Compiler unterstützen bereits OpenMP-Direktiven und sind dazu in der Lage, mit den älteren PCF (*Parallel Computer Forum*) und SGI `DOACROSS`-Direktiven wie auch mit den OpenMP-Direktiven umzugehen.

3.3.1 Ausführungsmodell

Das beschriebene Ausführungsmodell bezieht sich auf die älteren PCF konformen Direktiven.

Die Parallelisierung des SGI-Compilers findet auf der Ebene der Zählschleifen statt. Die Programmausführung startet zunächst als einzelner Thread, wird bei Erreichen einer parallelen Schleife jedoch als Team fortgesetzt. Nach der Schleife wird das Team nicht zwangsläufig wieder aufgelöst, da die Möglichkeit besteht, die überzähligen Threads in den Zustand *blocked* zu überführen. Bei der nächsten parallelen Schleife

Syntax	Bedeutung
`C*$* NO CONCURRENTIZE` `#pragma no concurrentize`	Abhängig vom Eintragungsort, wird das Programm oder die entsprechende Routine nicht parallelisiert.
`C*$* CONCURRENTIZE` `#pragma concurrentize`	Hebt die Wirkung von `NO CONCURRENTIZE` auf.
`C*$* ASSERT DO (CONCURRENT)` `#pragma concurrent`	Bei der Parallelisierung wird nicht auf Datenabhängigkeiten geachtet.
`C*$* ASSERT DO (SERIAL)` `#pragma serial`	Die nächste Schleife wird nicht parallelisiert.
`C*$* ASSERT CONCURRENT CALL` `#pragma concurrent call`	Bei der Parallelisierung wird nicht auf Funktionsaufrufe geachtet.
`C*$* ASSERT PERMUTATION (`*Feldname*`)` `#pragma permutation (`*Feldname*`)`	Das angegebene Feld enthält keine doppelten Werte.
`C*$* ASSERT DO PREFER (CONCURRENT)` `#pragma prefer concurrent`	Bevorzuge (falls möglich) die Parallelisierung der nächsten Schleife.
`C*$* ASSERT DO PREFER (SERIAL)` `#pragma prefer serial`	Parallelisiere die folgende Schleife nicht.

Tabelle 3.6: Unterstützende Direktiven in SGI-Fortran

werden diese schlafenden Threads dann einfach wieder aufgeweckt. Falls jedoch nur eine einzige zu parallelisierende Zählschleife vorhanden ist, kann das Team auch mittels *create/destroy* erzeugt bzw. aufgelöst werden.

3.3.2 Direktiven

Unterstützende Direktiven

Wie bereits im vorherigen Abschnitt erwähnt, existieren zwei Gruppen von Direktiven. Die nachfolgend erklärten Direktiven geben dem MIPSpro-Compiler eine Hilfestellung bei der automatischen Parallelisierung (Tabelle 3.6).

Diese unterstützenden Direktiven sorgen dafür, daß der Compiler bei der Übersetzung mehr Informationen zur Verfügung hat. Dies ist insbesondere bei geschachtelten Zählschleifen von Interesse, da es sinnvoller ist, eine Schleife mir vielen Iterationen zu parallelisieren. Durch Angabe der entsprechenden Direktiven kann eingestellt werden, welche Schleifen bei der Parallelisierung, falls dies möglich ist, zu bevorzugen sind.

Eine Sonderstellung nehmen die Direktiven `C*$* ASSERT DO (CONCURRENT)`, `C*$* ASSERT CONCURRENT CALL` und `C*$* ASSERT PERMUTATION (`*Feldname*`)` ein, da sie auch dann eine Wirkung haben, wenn das Programm nicht mit der Option `-apo`

Syntax	Bedeutung
C$DOACROSS[*Parameterliste*]	Die unmittelbar nachfolgende Zählschleife wird parallelisiert.
C$&	Erweitert das C$DOACROSS um 1 Zeile.
C$	Bedingte Übersetzung
C$MP_SCHEDTYPE=*Typ*	Schleifenscheduling einstellen
C$CHUNK=*Anzahl*	Blockgröße einstellen

Tabelle 3.7: Direktiven zur expliziten Parallelisierung in SGI-Fortran

übersetzt wird. Diese Direktiven erlauben in einem solchen Fall, daß der Compiler bei geschachtelten Zählschleifen eine Vertauschung vornimmt, falls dies effizienter ist.

Explizite Parallelisierung

Die dritte Möglichkeit ein Programm zu parallelisieren, wird durch die nachfolgend beschriebenen Direktiven (Tabelle 3.7) geboten [8][9]. Durch diese Direktiven wird der Compiler angewiesen, die jeweiligen Zählschleifen in der angegebenen Weise zu parallelisieren.

C$DOACROSS[*Parameterliste*]

Diese Direktive wird unmittelbar vor die zu parallelisierende Zählschleife in den Quellcode eingefügt. Mit Hilfe der *Parameterliste* kann angegeben werden, wie die einzelnen Iterationen der Schleife aufgeteilt werden sollen und wie die in der Schleife vorkommenden Variablen zu behandeln sind.

Falls die *Parameterliste* so lang ist, daß sie nicht mehr in erste Zeile paßt, so kann sie in der nächsten Zeile durch voranstellen von C$& fortgesetzt werden. Die in der Schleife vorkommenden Variablen werden standardmäßig alle als gemeinsam genutzt deklariert. Ausnahme ist des Schleifenzähler, welcher als privat mit Rückgabewert deklariert wird.

Bei ineinander geschachtelten Schleifen kann immer nur eine einzige Schleife pro geschachteltem Konstrukt parallelisiert werden.

Die für die *Parameterliste* gültigen Einträge werden nachfolgend aufgelistet und beschrieben.

SHARE(*Variablenliste*)
Alle in der *Variablenliste* angegebenen Variablen werden als gemeinsam genutzt deklariert.

`LOCAL(`*Variablenliste*`)`
Alle in der *Variablenliste* angegebenen Variablen werden als privat genutzt deklariert. Dies bedeutet, daß jeder Thread seine eigene Kopie der ursprünglichen Variablen erhält, welche jedoch noch nicht mit Werten initialisiert sind.

`LASTLOCAL(`*Variablenliste*`)`
Dies hat die gleiche Funktion wie `LOCAL`, jedoch mit dem Unterschied, daß die angegebenen Variablen nach der parallel abgearbeiteten Schleife einen definierten Wert besitzen. Die Werte der entsprechenden privaten Kopien der letzten Iteration, werden nach der Schleife in die ursprünglichen Variablen übernommen.

`REDUCTION(`*Liste skalarer Variablen*`)`
Soll ein Vektor innerhalb einer Schleife mit Hilfe der Operationen +, *, MIN oder MAX zu einem einzigen Wert reduziert werden, so muß die Reduktionsvariable als solche gekennzeichnet werden, damit die Schleife parallel ausgeführt werden kann. Alle Reduktionsvariablen werden in einer Liste hinter dem Parameter `REDUCTION` angegeben.

`IF(`*Bedingung*`)`
Falls die angegebene *Bedingung* zur Programmlaufzeit zu einem positiven Wahrheitswert evaluiert werden kann, wird die Schleife parallel abgearbeitet. Bei einem negativen Wahrheitswert erfolgt die Abarbeitung sequentiell. Diese Bedingung kann dazu genutzt werden, eine Schleife nur dann parallel abarbeiten zu lassen, wenn die zu verteilende Arbeit größer ist, als der durch die Parallelisierung erzeugte Overhead. Laut dem MIPSpro-Compilerhandbuch liegt die Grenze zur Zeit bei ungefähr 400 CPU Taktzyklen.

`CHUNK=`*ganze Zahl*
Über den Parameter `CHUNK=...` wird die für das Schleifenscheduling ggf. benötigte Blockgröße eingestellt. Wird dieser Parameter weggelassen, so wird der Vorgabewert 1 als Blockgröße angenommen.

`MP_SCHEDTYP=`*Typ*
Über diesen Parameter wird eingestellt, wie die einzelnen Iterationen der parallel auszuführenden Schleife auf die einzelnen Prozessoren verteilt werden sollen. Wird als *Typ* der Parameter `SIMPLE` angegeben, werde die Iterationen der Schleife gleichmäßig auf alle Prozessoren verteilt.

`INTERLEAVE` sorgt dafür, daß die durch `CHUNK=...` angegebene Anzahl von Iterationen zyklisch an die Prozessoren verteilt werden. Dies bedeutet, daß bei einer Blockgröße von zwei und vier beteiligten Prozessoren die Iterationen 1-2, 9-10, 17-18, ... an den ersten Prozessor und die Iterationen 3-4, 11-12, 19-20, ... an den zweiten Prozessor gegeben werden. Die Vergabe an die Prozessoren drei und vier erfolgt analog.

`DYNAMIC` funktioniert analog zu `INTERLEAVE`, jedoch werden die Iterationen zu Beginn der Schleife nicht starr auf die Prozessoren verteilt, sondern dynamisch zur Programmlaufzeit vergeben. Dies bedeutet, daß Prozessoren, die ihre Arbeit erledigt haben, sich den nächsten Block Iterationen abholen.

`GSS` (*guided self scheduling*) verteilt immer kleiner werdende Blöcke von Iterationen an die Prozessoren. Die Blockgrößen sind abhängig von der Gesamtzahl der in der Schleife vorkommenden Iterationen.

Durch die Angabe von `RUNTIME` wird das Schleifenscheduling nicht zur Compilezeit festgelegt, sondern erst zur Programmlaufzeit. Wurde dieser Parameter angegeben, so wird durch die Laufzeitumgebung eine der 4 Schedulingvarianten bestimmt.

`C$`

Durch diese Direktive wird eine bedingte Übersetzung des Quellcodes möglich, da jede Zeile, die mit `C$` beginnt, normalerweise als Kommentarzeile gewertet wird. Bei einer Übersetzung für ein Mehrprozessorsystem werden durch den Compiler einfach die ersten beiden Zeichen durch Leerzeichen ersetzt, was dazu führt, daß die hinter dieser Direktiven stehenden Anweisungen nun aktiviert sind.

`C$MP_SCHEDTYP=`*Typ*

Diese Direktive dient zur Änderung des Vorgabewertes für das Schleifenscheduling. Der eingestellte *Typ* wird nur dann verwendet, wenn die Angabe von `SCHEDTYP` hinter einer `DOACROSS`-Direktive weggelassen wird.

`C$CHUNK=`*ganze Zahl*

Die durch `CHUNK` eingestellte Vorgabe tritt ebenfalls erst dann in Kraft, wenn hinter einer `DOACROSS`-Direktive keine andere Angabe gemacht wird.

3.3.3 Laufzeitroutinen

`mp_block()`
Durch den Aufruf dieser Routine werden alle Threads, mit Ausnahme des Hauptthreads, in den Zustand *blocked* überführt. Dies führt dazu, daß diese Threads keine CPU-Zeit mehr in Anspruch nehmen und die freien Prozessoren anderweitig verwendet werden können. Bei Eintritt des Programms in ein paralleles Konstrukt (parallelisierte Schleife) werden die Threads geweckt.

`mp_unblock()`
Diese Routine weckt die zuvor in den Zustand *blocked* überführten Threads wieder auf.

`mp_setup()`
Diese Routine wird ohne Argumente aufgerufen und erzeugt die durch

`mp_set_numthreads` oder durch die Umgebungsvariable `MP_SET_NUMTHREADS` eingestellte Anzahl von Threads. Der Aufruf dieser Routine geschieht automatisch zu Beginn des ersten parallelen Konstruktes.

`mp_create(`*Anzahl*`)`
Der durch *Anzahl* übergebene Wert gibt die Gesamtzahl aller Threads an. Falls beim Aufruf dieser Routine nur der Hauptthread existiert, werden (*Anzahl - 1*) Threads neu erzeugt.

`mp_destroy()`
Diese Routine zerstört alle Threads mit Ausnahme des Hauptthreads.

`mp_blocktime(`*Anzahl*`)`
Diese Routine stellt ein, wie lange ein leerlaufender Thread warten soll, bevor er sich selbst in den Zustand *blocked* überführt, um CPU-Zeit zu sparen. Die Standardvorgabe beträgt 10.000.000 CPU-Zyklen, was ungefähr einer Wartezeit von drei Sekunden gleichkommt. Durch die Angabe von 0 kann dieser Automatismus deaktiviert werden.

`INTEGER mp_numthreads()`
Diese Funktion liefert als Rückgabewert die Gesamtzahl aller am aktuellen Programm beteiligten Threads. Der Hauptthread wird hierbei mitgezählt.

`mp_set_numthreads(`*Anzahl*`)`
Diese Routine setzt die Anzahl der zu erzeugenden Threads neu, hat jedoch keine Auswirkung auf schon vorhandene Threads.

`INTEGER mp_my_threadnum()`
Diese Funktion liefert als Rückgabewert die eigene Threadnummer zurück. Der Hauptthread trägt immer die Nummer Null.

`mp_setlock(), mp_unsetlock(), mp_barrier()`
Durch diese Routinen kann eine Sperre gesetzt und wieder aufgehoben werden. Die Sperre selbst wird beim Programmstart automatisch initialisiert und kann daher sofort benutzt werden.

3.3.4 Umgebungsvariablen

`MP_SET_NUMTHREADS`
Durch diese Angabe wird die Anzahl der Threads angegeben, die für ein paralleles Programm erzeugt werden sollen. Es können durchaus mehr Threads erzeugt werden, als Prozessoren vorhanden sind.

`MP_BLOCKTIME`
Die Einstellung durch diese Umgebungsvariable hat die gleiche Wirkung wie der Aufruf der Laufzeitroutine `mp_blocktime` zu Beginn des auszuführenden Programms.

`MP_SCHEDTYPE`
Die Angabe in dieser Umgebungsvariablen wirkt sich nur auf parallelisierte Schleifen

aus, bei denen das Schleifenscheduling erst zur Laufzeit festgelegt wird. Für diese Schleife wird der mit Hilfe dieser Umgebungsvariablen angegebene Schleifenschedulingtyp verwendet.

`MP_CHUNK`
Durch die Umgebungsvariable `CHUNK` kann die Blockgröße für das Schleifenscheduling eingestellt werden.

`MP_PROFILE`
Durch Setzen dieser Umgebungsvariablen werden langsamere Routinen zur Synchronisation der einzelnen Threads verwendet. Zur besseren Zuordnung werden für die Subroutinen lange Namen verwendet. Dieses Verhalten dient dazu, bei einem Profilinglauf mehr Informationen über die einzelnen Zeitaufwände innerhalb des parallelen Programms zu erhalten. Durch `unsetenv MP_PROFILE` wird diese Option wieder deaktiviert.

3.4 Ergänzungen zu OpenMP auf SGI-Systemen

Die von SGI entwickelten Direktiven zur Datenverteilung sind in Kombination mit den OpenMP-Direktiven verwendbar. Die zusätzlichen Direktiven dienen dazu, gemeinsame Variablen explizit über die lokalen Speicher der beteiligten Prozessoren eines Mehrprozessorsystems mit verteiltem gemeinsamen Speicher (z. B. SGI Origin2000) zu verteilen und den Zugriff darauf innerhalb einer parallelen Schleife zu steuern [10, 11].

3.4.1 Direktiven zur expliziten Datenverteilung

Die zur Verfügung stehenden Direktiven sind in Tabelle 3.8 zusammengefaßt und werden im folgenden Abschnitt genauer erklärt.

!$SGI DISTRIBUTE / #pragma distribute

Mit Hilfe dieser Direktive kann eine Verteilung von Daten über verschiedene Prozessoren erreicht werden. Die ursprüngliche Struktur der zu verteilenden Daten bleibt dabei erhalten, da nur die Abbildung der Seiten des virtuellen Speichers auf den verteilten physikalischen Speicher verändert wird. Dies hat zur Folge, daß eine solche Art der Datenverteilung nur dann sinnvoll ist, wenn die einem Prozessor zugewiesene Portion wesentlich größer als eine Speicherseite (16 KB auf der Origin2000) ist.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma distribute array[dist1][[dist2]...] [onto (dim1, dim2[, dim3])]
```

Syntax	Bedeutung
`!$SGI DISTRIBUTE` `#pragma distribute`	Spezifiziert die Datenverteilung
`!$SGI REDISTRIBUTE` `#pragma redistribute`	Spezifiziert die dynamische Neuverteilung der Daten
`!$SGI DISTRIBUTE_RESHAPE` `#pragma distribute_reshape`	Spezifiziert die Datenverteilung mit Modifikationen an der Variablenstruktur
`!$SGI DYNAMIC` `#pragma dynamic`	Teilt dem Compiler mit, daß die Variablen u. U. neu verteilt werden
`!$SGI PAGE_PLACE` `#pragma page_place`	Explizite Plazierung der Daten

Tabelle 3.8: Direktiven zu Datenverteilung in SGI-Fortran

Fortran benutzt folgende Syntax.

`!$SGI DISTRIBUTE` *array*`[`*dist1*`][[`*dist2*`]...] [ONTO (`*dim1, dim2*`[,` *dim3*`])]`

Mit *array* wird das zu verteilende (u. U. mehrdimensionale) Feld bezeichnet. Der folgende Parameter *dist* gibt für jede Dimension des Feldes die Art der Datenverteilung an. Die Daten können in Blöcken (`block`) gleichmäßig auf die Prozessoren verteilt werden oder auch zyklisch (`cyclic[(`*chunk*`)]`) mit einer optionalen Blockgröße. Soll eine Dimension nicht verteilt werden, so wird dies durch einen Stern (`*`) gekennzeichnet. Über den optionalen Parameter `onto` kann das Verhältnis der Prozessorzahlen zu den einzelnen verteilten Dimensionen eingestellt werden. Diese Direktive muß im Deklarationsteil eines Programms unmittelbar hinter der Deklaration des zu verteilenden Feldes stehen. Die Daten werden auf die durch die Umgebungsvariable `MP_SET_NUMTHREADS` angegebene Prozessorzahl verteilt.

Beispiel:

```
float A[100][200]
#pragma distribute A[block][block] onto (1, 2)
```

Die Matrix `A` wird blockweise aufgeteilt, wobei die zweite Dimension von doppelt so vielen Prozessoren bearbeitet wird, wie die erste Dimension.

!$SGI REDISTRIBUTE / #pragma redistribute

Diese Direktive darf überall im Programmtext stehen und verteilt eine bereits bestehende Datenverteilung neu. Die Parameter und deren Syntax sind analog zu `distribute`.

!$SGI DYNAMIC / #pragma dynamic

Der Compiler geht standardmäßig davon aus, daß ein Feld während der Abarbeitung des Programms seine Speicherverteilung nicht ändert. Ist die Verteilung eines Feldes aber unbekannt, oder soll es zur Laufzeit umverteilt werden, so muß dies dem Compiler durch diese Direktive mitgeteilt werden.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma dynamic array
```

Fortran benutzt folgende Syntax.

```
!$SGI DYNAMIC array
```

!$SGI DISTRIBUTE_RESHAPE / #pragma distribute_reshape

Diese Direktive hat die gleiche Funktion und auch die gleichen Parameter wie `distribute`. Der Unterschied besteht darin, daß der Compiler ein mit `distribute_reshape` verteiltes Feld viel aggressiver optimiert. Diese Optimierung beinhaltet unter anderem eine Veränderung des Speicherlayouts der verteilten Variablen, damit die gewünschte Datenverteilung über die Prozessoren garantiert werden kann. Dies hat zur Folge, daß ein Programm keine Annahmen über das Speicherlayout (z. B. zusammenhängender Speicherbereich) eines Feldes machen darf. Eine mit dieser Direktive gewählte Verteilung kann nachträglich nicht mehr verändert werden (`redistribute_reshape` existiert nicht). Ein dynamisch erzeugtes (`malloc`) Feld kann ebenfalls nicht mit `distribute_reshape` verteilt werden.

!$SGI PAGE_PLACE / #pragma page_place

Mit Hilfe dieser Direktive können Daten explizit in den physikalischen Speicher des Prozessors plaziert werden, auf dem ein bestimmter Thread abgearbeitet wird. Diese Direktive darf überall im Programmcode stehen und verteilt statische wie auch dynamische Felder.

Für C und C++ wird folgende Syntax verwendet.

```
#pragma page_place (object, size, threadnum)
```

Fortran benutzt folgende Syntax.

```
!$SGI PAGE_PLACE (object, size, threadnum)
```

Mit *object* wird das erste Element des zu verteilenden Feldes bezeichnet. Der zu verteilende Block wird mit *size* in Bytes angegeben und in den Hauptspeicher des Prozessors geschrieben, auf dem der Thread mit der Nummer *threadnum* abgearbeitet wird.

Beispiel:

```
double A[8192]
#pragma page_place (A[0], 32768, 0)
#pragma page_place (A[4096], 16384, 1)
```

Die erste Direktive plaziert die erste Hälfte des Feldes `A` in den Hauptspeicher des Prozessors, auf dem der Thread null arbeitet. Die zweite Direktive plaziert das nächste Viertel in den Speicher des Prozessors auf dem der Threads eins arbeitet. Das letzte Viertel des Feldes `A` wird nicht explizit verteilt.

3.4.2 Ergäzungen zu OpenMP

Der SGI MIPSpro 7 Fortran Compiler unterstützt einige Direktiven, die die bereits vorgestellten SGI und auch OpenMP Direktiven in ihrer Funktion erweitern. Aus Kompatibilitätsgründen beginnen diese zusätzlichen Direktiven alle mit `!$SGI+` und müssen in einer eigenen Zeile unmittelbar hinter der Direktive stehen, die sie erweitern. Die folgenden Ergänzungen sind zur Zeit in C nur in Kombination mit SGI-Direktiven verwendbar, in Fortran jedoch auch in Kombination mit den OpenMP-Direktiven.

`!$SGI+AFFINITY`

Diese Direktive beeinflußt die Aufteilung der Schleifeniterationen einer parallelen Schleife so, daß die Iterationen an genau den Thread gegeben werden, der die jeweils benötigten Daten in seinem lokalen Hauptspeicher hat.

Fortran benutzt folgende Syntax.

```
!$OMP PARALLEL DO
!$SGI+AFFINITY (do_variable) = DATA(array_element)
```

```
!$OMP PARALLEL DO
!$SGI+AFFINITY (do_variable) = THREAD(expr)
```

Mit Hilfe von *do_variable* wird der Schleifenzähler der parallelen Schleife eingestellt. Bei datenabhängigem Scheduling bekommt der Thread die Iteration zugewiesen, der das durch *array_element* angegebene Element in seinem lokalen Speicher hat. Bei der zweiten Variante wird die aktuelle Iteration an den Thread mit der Nummer „*expr* modulo Threadzahl“ gegeben.

Beispiel:

```
!$SGI DISTRIBUTE A(BLOCK)
!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(A(I))
  DO I=1, N
    A(I) = 0
  END DO
```

Die Iteration `I` wird auf dem Prozessor ausgeführt, der das entsprechende Element `A(I)` in seinem lokalen Speicher hat. Das Schleifenscheduling basiert also auf die durch `DISTRIBUTE` erreichte Verteilung des Feldes `A`.

!$SGI+NEST

Mit dieser Direktive können (wenn auch eingeschränkt) mehrere ineinander geschachtelte Schleifen gleichzeitig parallelisiert werden. Die zu parallelisierenden Schleifen müssen dazu perfekt geschachtelt sein.

Fortran benutzt folgende Syntax.

```
!$OMP PARALLEL DO
!$SGI+NEST (do1, do2[, do3] ...) [ONTO (dim1, dim2[, dim3] ...)]
```

Mit *do1, do2 ...* werden die Schleifenzähler der zu parallelisierenden perfekt geschachtelten Schleifen bezeichnet. Durch die Angabe von (*dim1, dim2, ...*) kann analog zu `DISTRIBUTE` ein Verhältnis eingestellt werden, das angibt wie die beteiligten Prozessoren auf die zu parallelisierenden Schleifen aufgeteilt werden sollen.

Beispiel:

```
!$OMP PARALLEL DO
!$SGI+NEST(I, J) ONTO(2, *)
  DO I = 1, N
    DO J = 1, M
      A(J,I) = ...
    END DO
  END DO
```

In diesem Beispiel bearbeitet ein Team, welches aus zwei Threads besteht, die äußere Schleife parallel ab. Die innere Schleife wird von den restlichen Threads parallel abgearbeitet.

Die `NEST`-Direktive darf nur eingesetzt werden, wenn die Schleifen perfekt geschachtelt sind. Dies bedeutet, daß kein Code zwischen den `DO` oder den `END DO`-Anweisungen stehen darf.

Eine Kombination von `NEST` mit Direktiven, die das Schleifenscheduling beeinflussen, ist ebenfalls nur teilweise erlaubt. Nur die Kombination mit `AFFINITY` oder mit `SCHEDULE(STATIC,` *chunk*`)` ist möglich.

3.4.3 Umgebungsvariablen

Im folgenden Abschnitt werden einige wichtige Umgebungsvariablen beschrieben, die ebenfalls Einfluß auf die Datenverteilung haben.

_DSM_PLACEMENT
Diese Umgebungsvariable kann 2 Werte annehmen und beeinflußt die Strategie, mit der das Betriebssystem der Origin2000 standardmäßig Variablen eines Programms allokiert. Bei der Strategie FIRST_TOUCH werden die Variablen im Hauptspeicher des Prozessors angelegt, der diese Variablen als erstes benutzt. Die Strategie ROUND_ROBIN sorgt dafür, daß die Variablen mittels eines round-robin Verfahrens über die lokalen Speicher der Prozessoren verteilt angelegt werden. Die Voreinstellung für diese Variable ist FIRST_TOUCH.

_DSM_MIGRATION
Das Betriebssystem der Origin2000 kann erkennen, welche Prozessoren wie oft auf welche Speicherseiten (lokale oder entfernte) zugreifen. Standardmäßig werden die Speicherseiten, welche zum Hauptspeicher eines Prozessors gehören, nicht an den Hauptspeicher eines anderen Prozessors gegeben, auch wenn hauptsächlich dieser darauf zugreift.

ON Seitenzugriffskontrolle aktivieren und eine Speicherseite an den Hauptspeicher des Prozessors geben, der diese Seite am häufigsten benutzt. Diese Einstellung betrifft *nicht* die Daten, die mit Hilfe von `page_place` explizit verteilt werden.

ALL_ON Aktiviert die Seitenzugriffskontrolle für *alle* Daten.

OFF Seitenzugriffskontrolle deaktivieren.

_DSM_MIGRATION_LEVEL
Über einen ganzzahligen Wert zwischen 0 und 100 (einschließlich) kann eingestellt werden, wie aggressiv die Umverteilung von Speicherseiten bei entfernten Zugriffen vorgenommen wird. Je größer der Wert, desto effizienter arbeitet die Seitenzugriffskontrolle. Die Standardvorgabe für diese Variable ist 100.

_DSM_WAIT
Falls ein Prozessor an einem Synchronisationspunkt warten muß, kann er aktiv warten (SPIN) oder in der Zwischenzeit periodisch an einem anderen lauffähigen Job weiterarbeiten (YIELD). Die Standardeinstellung ist YIELD.

MP_SUGNUMTHD
Durch setzen dieser Umgebungsvariablen wird die dynamische Threaderzeugung aktiviert. Dies bedeutet, daß das Betriebssystem abhängig von der momentanen Auslastung des Systems die Anzahl der für eine parallele Schleife zu erzeugenden Threads selbst bestimmt.

3.5 Konzeptioneller Vergleich der vorgestellten Programmiermodelle

	OpenMP	SUN	Cray	SGI
inkrementelle Parallelisierung	Ja	Ja	Ja	Ja
loop level	Ja	Ja	Ja	Ja
parallele Regionen	Ja	-	Ja	-
parallele Sektionen	Ja	-	Ja	-
verwaiste Direktiven	Ja	-	-	-
geschachtelte parallele Schleifen	Ja	-	-	eingeschränkt
geschachtelte parallele Regionen	Ja	-	-	-
Direktiven zur Datenverteilung	-	-	-	Ja
Datenabhängiges Schleifenscheduling	-	-	-	Ja

Tabelle 3.9: Vergleich der Programmiermodelle

Die Compiler der Firmen SUN, Cray und SGI bieten die Möglichkeit, einen eingegebenen sequentiellen Programmcode automatisch parallelisieren zu lassen. Diese automatische Parallelisierung kann bei allen Compilern mit den in den Quelltext einzufügenden Direktiven kombiniert werden.

Die vorgestellten Programmiermodelle sind sich sehr ähnlich und verfolgen im Grunde genommen den gleichen Ansatz, um sequentielle in parallele Programme zu überführen, bzw. direkt parallele Programme zu schreiben. Alle Modelle bieten die Möglichkeit, einen bereits vorhandenen sequentiellen Programmcode mit Hilfe von Direktiven Zug um Zug zu parallelisieren. Die entsprechenden Direktiven sind in Kommentarzeilen verpackt, so daß der Quellcode auch weiterhin von einem *normalen* Compiler übersetzt werden kann.

Alle diese Programmiermodelle basieren auf dem *fork-join* Modell und beginnen die Programmabarbeitung mit einem einzelnen Thread. Bei Erreichen eines parallelen Konstruktes werden neue Threads erzeugt und zu einem Team zusammengefaßt. Dieses Team führt die im parallelen Konstrukt enthaltene Arbeit gemeinsam aus und wird anschließend aufgelöst, so daß die weitere Programmausführung sequentiell erfolgt.

Der gemeinsame Ansatzpunkt für die Erzeugung von parallelem Programmcode ist die Zählschleife (loop level Parallelismus). Falls das zugrundeliegende Programmiermodell nicht die Definition von parallelen Regionen vorsieht, muß zu Beginn einer jeden parallelen Schleife ein neues Team erzeugt und am Ende wieder aufgelöst werden. Da dies jedoch relativ zeitaufwendig sein kann, werden die überzähligen Threads im allgemeinen nicht zerstört, sondern nur geparkt. Der SGI Compiler erzeugt mit der ersten parallelen Schleife ein Team von Threads, welches nach der Schleife nicht automatisch aufgelöst, sondern die überzähligen Threads geparkt werden. Dies bedeutet, daß die Threads eines Teams noch existieren, jedoch der Code nach einer

parallelen Schleife, im Gegensatz zu dem einer parallelen Region, nicht redundant durch das Team abgearbeitet wird. Das Team kann explizit durch `mp_destroy` aufgelöst werden.

Eine weitere Erzeugungsmöglichkeit für parallelen Code wird durch die OpenMP und Cray zugrundeliegenden Programmiermodelle geboten, da hier die Definition von Programmblöcken vorgesehen ist, welche unabhängig voneinander parallel abgearbeitet werden können (`SECTIONS, CASE`). Dies erlaubt, die parallele Abarbeitung eines Programms auf einem höheren Niveau zu beginnen.

Die Definition einer parallelen Region ist bei OpenMP und Cray explizit durch eigene Direktiven vorgesehen. Die Compiler der Firmen SUN und SGI beschränken sich auf einzelne Zählschleifen und erlaubt keine explizit definierten parallelen Regionen.

Im Gegensatz zu OpenMP erlauben die drei vorgestellten Ausführungsmodelle der Firmen Cray, SGI und SUN jedoch keine Schachtelung von parallelen Regionen oder parallelen Schleifen. Alleine SGI bietet zumindest die Möglichkeit, perfekt geschachtelte Schleifen zu parallelisieren.

Die OpenMP-Spezifikation erlaubt es, daß Direktiven zur Arbeitsteilung auch außerhalb der statischen Erweiterung einer parallelen Region stehen dürfen. Dies hat zur Folge, daß die Definition einer parallelen Region auf sehr hohem Level erfolgen kann, obwohl die Konstrukte zur Arbeitsteilung erst in Unterprogrammen auftauchen. Da Direktiven, die zur Arbeitsteilung verwendet werden, außerhalb einer parallelen Region ignoriert werden, ist es möglich auf diese Art und Weise Unterprogramme zu konstruieren, die nur dann parallel arbeiten, wenn sie aus einer parallelen Region heraus aufgerufen werden.

Als einziger Hersteller bietet SGI Direktiven an, die sich auf die Verteilung von Daten über die lokalen Hauptspeicher eines Parallelrechners mit physikalisch verteiltem gemeinsamen Hauptspeicher beziehen. Desweiteren sieht SGI Direktiven vor, die das Scheduling einer parallelen Schleife in Abhängigkeit einer gewählten Datenverteilung beeinflussen. Dies hat zur Folge, daß die Iterationen einer Schleife so auf die Prozessoren verteilt werden, daß die zur Berechnung benötigten Daten lokal vorliegen.

Direktiven dieser Art fehlen in der OpenMP Spezifikation, so daß SGI (zumindest für die Sprache Fortran) die Kombination von Direktiven zur Datenverteilung und Beeinflussung des Schleifenschedulings in Abhängigkeit von einer gewählten Datenverteilung mit den OpenMP Direktiven zuläßt.

Kapitel 4

Leistungs- und Skalierbarkeitsanalyse von OpenMP-Primitiven

Im folgenden Abschnitt wird untersucht, wie effizient die Programmierschnittstelle OpenMP auf verschiedenen Hardwarearchitekturen implementiert ist. Zu diesem Zweck wurde der jeweilige Overhead für die einzelnen OpenMP-Primitive analysiert. Die Konstrukte für parallele Schleifen wurden besonders intensiv untersucht, da sie im wissenschaftlichen Rechnen die Hauptquelle für Parallelität darstellen.

4.1 Hardwarearchitekturen

OpenMP ist ein Programmiermodell für Parallelrechner mit gemeinsamem Speicher. Dieser gemeinsame Speicher muß vom benutzten Rechnersystem entweder in Software oder in Hardware zur Verfügung gestellt werden. Bei dieser Untersuchung kommen Rechner mit unterschiedlichen, in Hardware realisierten, Implementierungen des gemeinsamen Speichers zum Einsatz.

4.1.1 SGI/CRAY T90

Die T90 ist ein paralleler Vektorkomputer mit bis zu 32 Prozessoren. Die Anbindung eines Prozessors an den Hauptspeicher (SRAM) erfolgt über ein komplexes Netzwerk, welches jeder CPU den Zugriff auf jede Speicherstelle mit der gleichen Latenzzeit ermöglicht. Diese Art der Speicheranbindung wird als Uniform Memory Access (UMA) bezeichnet. Die für die Messung verwendete Maschine hat 10 Prozessoren, die mit 475 MHz getaktet sind. Für die Übersetzung des Quellcodes wurde der CRAY CF90 Fortran Compiler in der Version 3.3 mit der Option *-O3* benutzt.

4.1.2 SGI/CRAY J90

Die CRAY J90 ist der T90 sehr ähnlich, verfügt jedoch über einen langsameren Hauptspeicher (DRAM). Die für die Messung verwendete Maschine hat 16 Prozessoren, die mit 100 MHz getaktet sind (*J90 classic*). Für die Übersetzung des Quellcodes wurde ebenfalls der CRAY CF90 Fortran Compiler in der Version 3.3 mit der Option *-O3* benutzt.

4.1.3 SGI Origin2000

Die Origin2000 ist ein cache-coherent Non-Uniform Memory Access (cc-NUMA) Mehrprozessorsystem mit bis zu 256 Prozessoren. Diese Maschine besteht aus Boards, welche jeweils 2 MIPS R12000 Prozessoren, 8 MB L2 Cache, einen lokalen Hauptspeicher und einen Hub enthalten, welcher den globalen Adreßbereich über alle Rechenknoten realisiert. Falls keine explizite Verteilungsstrategie für Daten angegeben wird, wird Speicher für gemeinsame Variablen auf dem Knoten reserviert, der diese Variablen als erstes benutzt (first touch strategy). Dies hat zur Folge, daß ein anderer Rechenknoten diese gemeinsamen Variablen zunächst in seinen eigenen Cache kopieren muß, bevor er darauf zugreifen kann. Die für die Messung verwendete Maschine hat 128 Prozessoren mit 300 MHz. Für die Übersetzung des Quellcodes wurde der SGI MIPSpro Fortran Compiler in der Version 7.2.1 mit den Optionen *-mp -O3* benutzt.

4.1.4 IBM R50

Die IBM R50 ist eine Maschine mit gemeinsamem Speicher, welche aus 4 Boards mit jeweils 2 PowerPC 604e Prozessoren (200 MHz) und 2 MB L2 Cache aufgebaut ist. Der Hauptspeicherzugriff erfolgt über einen Kreuzschienenverteiler. Für die Übersetzung des Quellcodes wurde der XLF95_r Fortran Compiler in der Version 6.1 mit den Optionen *-qsmp=omp -O4* benutzt.

4.2 Analyse der Primitive

Alle Messungen wurden auf den jeweiligen Systemen in einer dedizierten Meßzeit mit den Standardeinstellungen vorgenommen. Die einzelnen Konstrukte wurden in 5 Meßläufen gemessen und der beste Wert ausgewählt. Die Zeiterfassung fand üblicherweise mit dem folgenden Konstrukt statt:

```
!$OMP BARRIER
  t0 = get_time()
  OpenMP Konstrukt
  t1 = get_time()
  time = (t1 - t0) - overhead
```

Die nötigen Variablen zur Zeiterfassung waren alle als *PRIVATE* deklariert, um Einflüsse durch das Speichersubsystem zu vermeiden.

4.2.1 Laufzeitroutinen

Der für die Laufzeitroutinen ermittelte Zeitbedarf ist in Tabelle 4.1 zusammengefaßt.

	T90	J90	O2000	R50
OMP_SET_NUM_THREADS	0,700	2,720	22,400	—[1]
OMP_GET_NUM_THREADS	0,293	0,640	0,117	—[1]
OMP_GET_MAX_THREADS	0,266	0,470	0,183	0,371
OMP_GET_THREAD_NUM	0,441	0,640	0,117	0,347
OMP_GET_NUM_PROCS	0,405	0,470	0,117	0,357
OMP_IN_PARALLEL	0,411	0,620	0,117	0,352
OMP_SET_DYNAMIC	0,895[2]	1,490[2]	7,200[2]	—[1]
OMP_GET_DYNAMIC	0,645	1,390	1,594	—[1]
OMP_SET_NESTED	—[1]	—[1]	—[1]	—[1]
OMP_GET_NESTED	0,327	1,040	0,117	—[1]

[1] nicht unterstützt
[2] keine Auswirkungen

Tabelle 4.1: Zeiten für Laufzeitroutinen in μs

In Bezug auf das Zusammenspiel von Umgebungsvariablen und Laufzeitroutinen weichen die zur Messung herangezogenen Systeme in einigen wenigen Fällen leicht von der Spezifikation im OpenMP Fortran API ab.

Auf den Systemen SGI/CRAY J90 und SGI/CRAY T90 wird mit der Umgebungsvariablen `OMP_NUM_THREADS` eine Obergrenze für die Anzahl der zu erzeugenden Threads festgelegt. Die tatsächliche Anzahl kann zur Programmlaufzeit durch die Laufzeitroutine `OMP_SET_NUM_THREADS()` nachträglich in den Grenzen 1 bis `OMP_NUM_THREADS` eingestellt werden.

Für den IBM XLF95-Compiler [14] wird die Anzahl der zu erzeugenden Threads durch die Umgebungsvariable `XLSMPOPTS="parthds=`*Anzahl*`"` eingestellt. Die Anzahl der Threads kann zur Laufzeit eines Programm nicht mehr beeinflußt werden, da die entsprechenden Laufzeitroutinen in der vorliegenden Version nicht implementiert sind.

4.2.2 Parallele Regionen

Das OpenMP zugrundeliegende Ausführungsmodell basiert darauf, zu Beginn einer parallelen Region ein Team von Threads zu erzeugen, welche die Region gemeinsam abarbeiten.

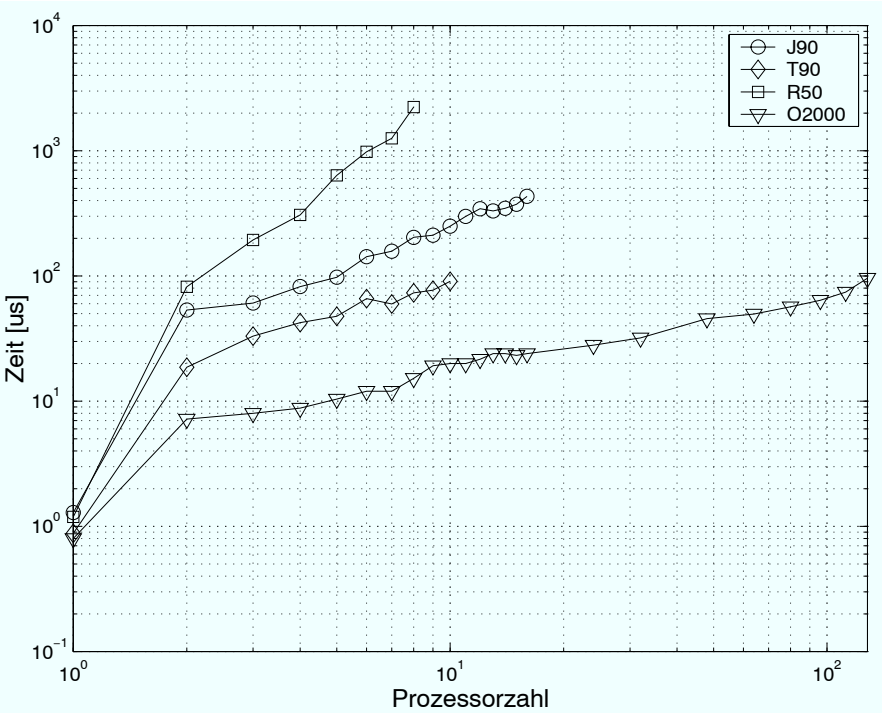

Abbildung 4.1: Zeitbedarf zur Erzeugung paralleler Regionen

Abbildung 4.1 zeigt die benötigte Zeit für das Konstrukt `!$OMP PARALLEL` in Abhängigkeit von der benutzten Prozessorzahl.

Bei der Origin2000 ist der Overhead sehr gering und wächst auch bei höheren Prozessorzahlen nicht übermäßig an. Im Gegensatz dazu ist die gleiche Operation auf der IBM R50 relativ zeitaufwendig, auch wenn nur wenige Prozessoren beteiligt sind. Da OpenMP jedoch erlaubt, Direktiven zur Arbeitsteilung in der dynamischen Erweiterung einer parallelen Region zu benutzen (*verwaiste Direktiven*), können parallele Regionen größeren Umfangs definiert werden, wodurch der Overhead zum Starten der parallelen Region relativ kleiner wird. Aus diesem Grund verlieren die höheren Kosten, zumindest bei Applikationen welche große parallele Regionen enthalten, an Bedeutung.

4.2.3 Parallele Schleifen

Parallele Schleifen sind die Hauptquelle für Arbeitsteilung in dem OpenMP zugrundeliegenden Ausführungsmodell. Daher ist eine effiziente Implementierung der

Schedulingalgorithmen, welche die Iterationen einer parallelen Schleife auf die beteiligten Prozessoren verteilen, erforderlich. Für das Schleifenscheduling stehen vier Verfahren zur Auswahl.

keine Angabe: Das Schedulingverfahren ist dem Compilerhersteller überlassen.

static: Die Gesamtzahl aller Schleifendurchläufe wird in möglichst gleich große Blöcke aufgeteilt und an die einzelnen Prozessoren verteilt.

dynamic: Solange noch nicht alle Schleifeniterationen abgearbeitet sind, bekommt der jeweils erste leerlaufende Prozessor einen Block Iterationen zugeteilt. Über einen optionalen Parameter kann die Größe eines Blocks eingestellt werden. Die Grundeinstellung dafür ist eins.

guided: Die Gesamtzahl aller Schleifendurchläufe wird in Blöcke mit abnehmender Größe aufgeteilt und analog zur Variante dynamic an die Prozessoren weitergegeben. Dies hat zur Folge, daß früher ankommende Prozessoren mehr Iterationen zugeteilt bekommen. Über einen optionalen Parameter kann eine minimale Blockgröße eingestellt werden. Die Grundeinstellung dafür ist eins.

runtime: Eine der Varianten `static`, `dynamic` oder `guided` wird zur Laufzeit in Abhängigkeit einer Umgebungsvariablen ausgewählt.

In Applikationsprogrammen, welche auf einer großen Anzahl von Prozessoren laufen, spielt der Datenzugriff eine wichtige Rolle und wird unter Umständen sogar zum dominierenden Faktor, der die Skalierbarkeit begrenzt. Bei dieser Untersuchung sollte jedoch die Skalierbarkeit der Schleife auf verschiedenen Systemen und der durch die OpenMP-Primitive verursachte Overhead bestimmt werden, ohne Einflüsse durch entfernte Zugriffe in die Hauptspeicher anderer Prozessoren.

Zu diesem Zweck wurden parallele Schleifen in allen Kombinationen untersucht, die sich aus der nachfolgenden Aufzählung ergeben. Die Schedulingvariante `static` wurde nur in Kombination mit konstant verteilter Arbeit analysiert.

Rechner: SGI/Cray J90, SGI/Cray T90, IBM R50, SGI Origin2000

Iterationszahl: 100, 1.000, 10.000, 100.000

Arbeitsverteilung: aufsteigend, absteigend, konstant

Schedulingtyp: dynamic, guided, static

Jede Schleife enthielt bei Messungen von 1 bis 16 Prozessoren Arbeit für ca. 20 Sekunden, welche aufsteigend, absteigend oder konstant über die jeweilige Iterationszahl verteilt sein konnte. Bei Messungen mit mehr als 16 Prozessoren wurde die Arbeit auf ca. 60 Sekunden erhöht. Innerhalb einer zu untersuchenden Schleife wurden ausschließlich private Variablen verwendet.

Außer bei den im folgenden diskutierten Ergebnissen wurde auf allen Systemen ein nahezu linearer Speedup erreicht, da keinerlei Konflikte beim Datenzugriff vorhanden waren und die Arbeit in den einzelnen Iterationen relativ groß (0,2 sec - 200 μs bei über 100 - 100.000 Iterationen konstant verteilter Arbeit). Bei feinerer Granularität der Schleifen wird der Overhead eine größere Rolle spielen und insofern kein linearer Speedup zu erzielen sein. Desweiteren war die Anzahl der verwendeten Prozessoren, mit Ausnahme auf der Origin2000, eher moderat.

Bei statischem Schleifenscheduling können die Threads eines Teams die von ihnen zu bearbeitenden Blöcke mit Hilfe der Schleifengrenzen, der Anzahl der beteiligten Threads und ihrer eigenen Threadnummer bestimmen. Zwischen den einzelnen Threads ist keine Kommunikation oder Synchronisation nötig, was zu einem sehr geringen Overhead führt. Nachteilig ist, daß eine parallele Schleife mit statischem Scheduling nur dann gut skalieren kann, wenn die in den einzelnen Iterationen enthaltene Arbeit konstant ist.

Im Folgenden werden die von einem linearen Skalierungsverhalten abweichenden Fälle mit jeweils 100.000 Schleifeniterationen genauer analysiert. Für kleinere Iterationszahlen, kleinere Prozessorzahlen oder andere Schedulingalgorithmen wird ansonsten nahezu linearer Speedup erreicht.

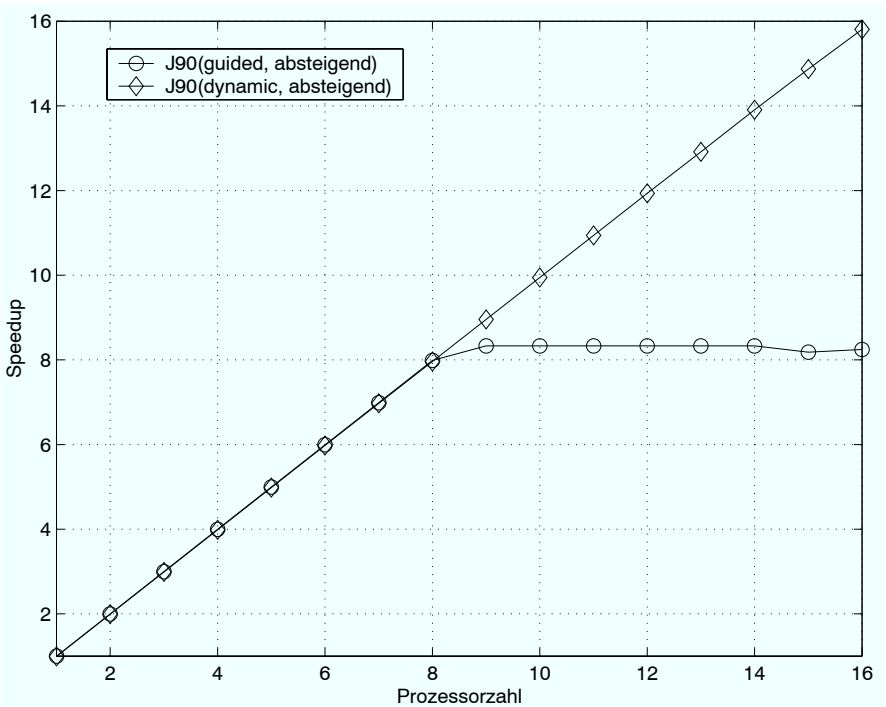

Abbildung 4.2: Schleifenmessung auf Cray J90

Abbildung 4.2 zeigt ab 8 Prozessoren einen waagerechten Verlauf der Speedupkurve der J90 bei guided self scheduling(GSS) und mit über 100.000 Iterationen absteigend verteilter Arbeit. Im Vergleich dazu zeigt die Speedupkurve der J90 bei dynamischem

Schleifenscheduling und ebenfalls über 100.000 Iterationen absteigend verteilter Arbeit keinerlei Degeneration, sondern linearen Speedup. Um dies zu erklären, ist ein kleiner Exkurs zum Guided Self Scheduling Algorithmus nötig.

Der GSS-Algorithmus berechnet die Anzahl der Iterationen x_i, welche an den i-ten Thread der ohne Arbeit ist gegeben werden, basierend auf einer rekursiven Formel: $x_i = \frac{1}{p} R_i$ wobei R_i die Anzahl der restlichen noch zu vergebenden Iterationen angibt und p die Anzahl der beteiligten Threads. Bei der CRAY J90 und T90 findet ein leicht modifizierter Algorithmus Verwendung, bei dem $\frac{1}{p}$ durch $1 - \frac{p-1}{p}$ ersetzt wurde, wobei der Nenner zur leichteren Division immer auf die nächste passende Zweierpotenz eingestellt wird (Divisionen durch 2 lassen sich sehr effizient durch Verschieben der Bits einer Zahl realisieren). Die Kombination aus dem modifizierten Algorithmus und der absteigend verteilten Arbeit führt zu dem in Abbildung 4.2 sichtbaren Knick in der Meßkurve, da der erste Thread zu viel Arbeit zugeteilt bekommt und daher eine Skalierung über 8 in diesem speziellen Fall unmöglich macht.

In Tabelle 4.2 sind für 100.000 Iterationen mit absteigend verteilter Arbeit, 16 Prozessoren und guided self scheduling auf der Cray J90 und T90 zusammengefaßt. Der erste Thread bekommt 6250 Iterationen zugeteilt, was bei der gewählten Arbeitsverteilung (insgesamt 20 Sekunden absteigend über alle Iterationen verteilt) ca. 2,42 Sekunden entspricht. Diese 2,42 Sekunden machen aber bereits über 12 Prozent der gesamten Arbeit aus, so daß die Gesamtlaufzeit für diese parallele Schleife alleine durch die Laufzeit des ersten Threads bestimmt wird.

restliche Iterationen R_i	verteilter Block X_i	Arbeitszeitanteil in Sekunden	Anteil in % (von 20 sec)	insges. verteilte Arbeit in Sekunden (von 20 sec)
100.000	6250	2,421874939	12,11	2,42
93.750	5859	2,128469185	10,64	4,55
87.891	5493	1,870794908	9,35	6,42
82.398	5150	1,644353760	8,22	8,07
77.248	4828	1,445194173	7,23	9,51
72.420	4526	1,270122298	6,35	10,78
67.894	4243	1,116290844	5,58	11,90
63.651	3978	0,981165721	4,91	12,88
59.673	3730	0,862495340	4,31	13,74
55.943	3496	0,757862863	3,79	14,50
52.447	3278	0,666194481	3,33	15,16
49.169	3073	0,585498677	2,93	15,75
46.096	2881	0,514609971	2,57	16,26
43.215	2701	0,452304048	2,26	16,72
40.514	2532	0,397503736	1,99	17,11
37.982	...			

Tabelle 4.2: GSS (J90 und T90) mit 100.000 Iterationen, 16 CPUs und absteigend verteilter Arbeit.

Abbildung 4.3 zeigt Messungen auf der SGI Origin2000 mit jeweils 100.000 Schleifeniterationen und konstant verteilter Arbeit. Bei statischem Schleifenscheduling wird aufgrund des sehr geringen Overheads linearer Speedup erreicht. Die Variante der Schleife mit dynamischem Schleifenscheduling zeigt nahezu linear bis zu 40 Prozessoren, um dann sehr stark einzubrechen. Diese Degenerierung ist durch die Implementierung des dynamischen Schedulings zu erklären.

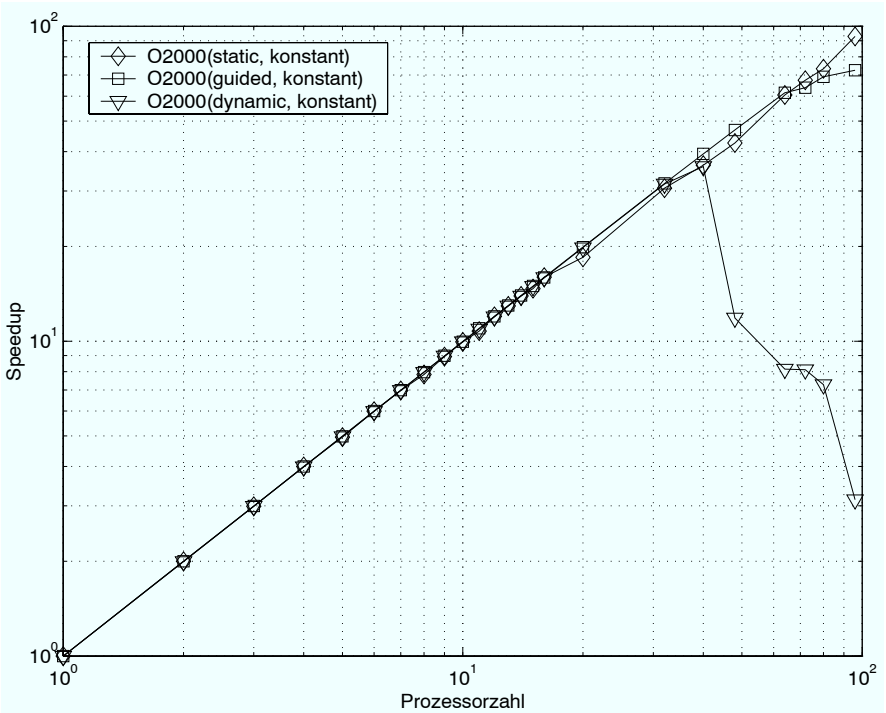

Abbildung 4.3: Schleifenmessung auf Origin2000

Bei dynamischem Schleifenscheduling wird die Verteilung der Iterationen im allgemeinen durch einen atomaren Zugriff auf eine gemeinsame Variable geregelt, welche den gerade aktuellen Wert des Schleifenzählers beinhaltet. Bei einer Blockgröße von eins (Standardeinstellung bei OpenMP) bedeutet dies, daß nach jeder Iteration ein atomarer Zugriff auf diese Variable erfolgen muß. Der Einbruch im Speedup liegt daran, daß der atomare Zugriff auf die gemeinsame Variable zum Flaschenhals wird, da viele Prozessoren um den exklusiven Zugriff auf diese Variable konkurrieren. Der Punkt, an dem die Speedupkurve einbricht, wird durch die Menge der Arbeit pro Iteration, die Anzahl der teilnehmenden Prozessoren und durch die Zugriffshäufigkeit auf die Variable beeinflußt. Die Zugriffshäufigkeit kann z. B. durch eine Erhöhung der Blockgröße gesenkt werden, was den Punkt, an dem der Speedup einbricht, auf eine höhere Prozessorzahl verschiebt.

Als Beispiel dafür kann die in Abbildung 4.3 dargestellte Speedupkurve der Origin2000 mit Guided Self Scheduling dienen. Die Kurve hat ebenfalls eine ähnliche Degenerierung wie bei dynamischem Schleifenscheduling, jedoch viel später und nicht so ausgeprägt. Dies ist darauf zurückzuführen, daß der GSS-Algorithmus zu

Beginn größere Blöcke an die Threads verteilt, wodurch der Zugriff auf die gemeinsame Variable eingeschränkt wird. Bei einer weiteren Erhöhung der Prozessorzahl ist jedoch auch in diesem Fall mit einem Zusammenbruch des Speedups zu rechnen.

4.2.4 Barrieren

Eine nicht unerhebliche Rolle kommt der effizienten Implementierung einer Barriere zu, da im OpenMP zugrundeliegenden Programmiermodell jedes Konstrukt zur Arbeitsteilung von einer impliziten Barriere abgeschossen wird, falls nicht explizit anders angegeben.

Die Messung für den Zeitbedarf einer Barriere wurde mit Hilfe einer in einer parallelen Region enthaltenen Schleife ermittelt. Da der Code innerhalb einer parallelen Region redundant abgearbeitet wird, muß jeder Thread des Teams die Schleife mit der Barriere ausführen.

```
!$OMP PARALLEL PRIVATE(t0,t1,i,j) FIRSTPRIVATE(overhead) SHARED(time)
  j = OMP_GET_THREAD_NUM()
  !$OMP BARRIER
  t0 = get_time()
  DO i=1, 1000
    !$OMP BARRIER
  END DO
  t1 = get_time()
  time% start(j) = t0
  time% stop(j) = t1 - overhead
!$OMP END PARALLEL
```

Der Zeitbedarf für die Bearbeitung des obigen Programmcodes wurde in Abhängigkeit von der Threadnummer in dem Feld `time` aufgesammelt. Die insgesamt benötigte Zeit berechnet sich als Differenz aus der Stopzeit des letzten Threads und der Startzeit des ersten Threads. Die Zahl der tatsächlich ausgeführten `BARRIER`-Operationen beträgt in diesem Fall 1000 multipliziert mit der Anzahl der beteiligten Threads.

Abbildung 4.4 zeigt den Zeitbedarf einer Barriere auf den verschiedenen Systemen in Abhängigkeit von der Prozessorzahl. Die Kurven der J90, T90 und Origin2000 zeigen einen sehr ähnlichen Verlauf, unterscheiden sich jedoch durch einen gewissen Offset. Der Kurvenverlauf bei der R50 zeigt im Gegensatz dazu einen steileren Anstieg und läßt auf einen leicht höheren Aufwand bei der Kommunikation zwischen den Prozessoren schließen.

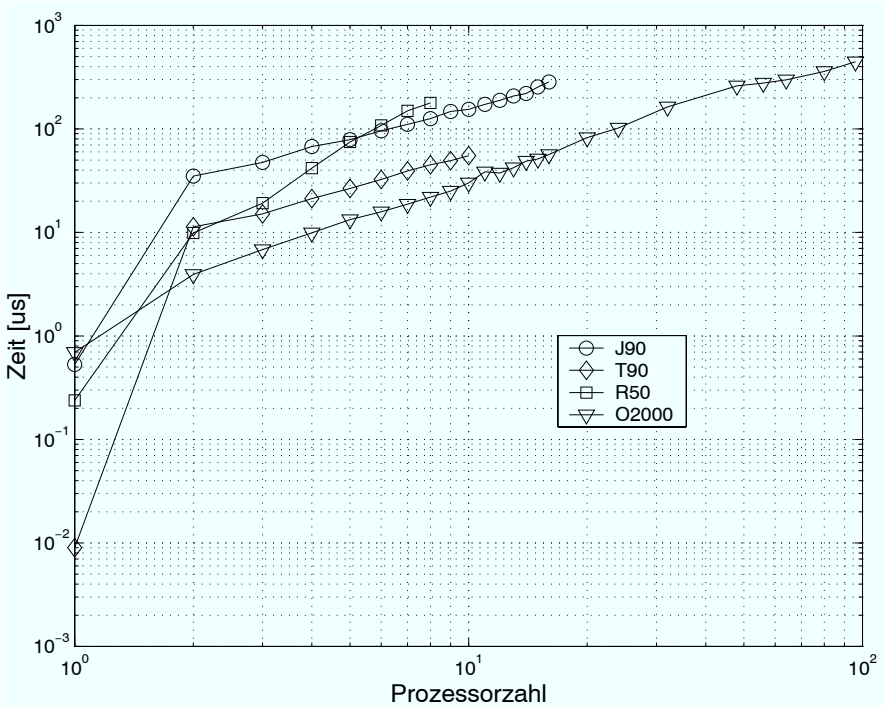

Abbildung 4.4: Zeitbedarf einer Barriere

4.2.5 Kritische Regionen und Sperren

Alle Messungen für Synchronisationskonstrukte ohne Einflüsse durch konkurrierende Threads wurden von nur einem Thread 10.000 mal ausgeführt und die verbrauchte Zeit durch die Anzahl der Operationen dividiert. Die Zeiten zum Initialisieren, Setzen, Aufheben, Testen und Löschen einer Sperre wie auch die Zeit für eine kritische Region sind ohne konkurrierende Threads sehr gering. Die beiden Cray Systeme benötigen für diese Operationen jedoch wesentlich mehr Zeit. Die genauen Werte sind in Tabelle 4.3 zusammengefaßt.

Operation	J90	T90	R50	O2000
`OMP_INIT_LOCK`	1.083	0.587	0.088	1.108
`OMP_DESTROY_LOCK`	1.574	0.802	0.065	0.966
`OMP_SET_LOCK`	2.108	1.279	0.491	0.314
`OMP_UNSET_LOCK`	2.121	1.299	0.501	0.123
`OMP_TEST_LOCK`	1.900	1.183	0.395	0.234
`!$OMP CRITICAL`	8.528	3.367	0.880	0.412

Tabelle 4.3: Synchronisationskonstrukte ohne Konkurrenzbedingungen (Zeit in μs)

Abbildung 4.5 zeigt den Zeitbedarf für eine kritische Region unter Konkurrenzbedingungen in Abhängigkeit von der Prozessorzahl. Alle Threads führen in einer parallelen Region eine Schleife mit 10.000 `!$OMP CRITICAL` Operationen (jeder Thread) aus und konkurrieren so um diese kritische Region. Die verbrauchte Zeit wurde durch die

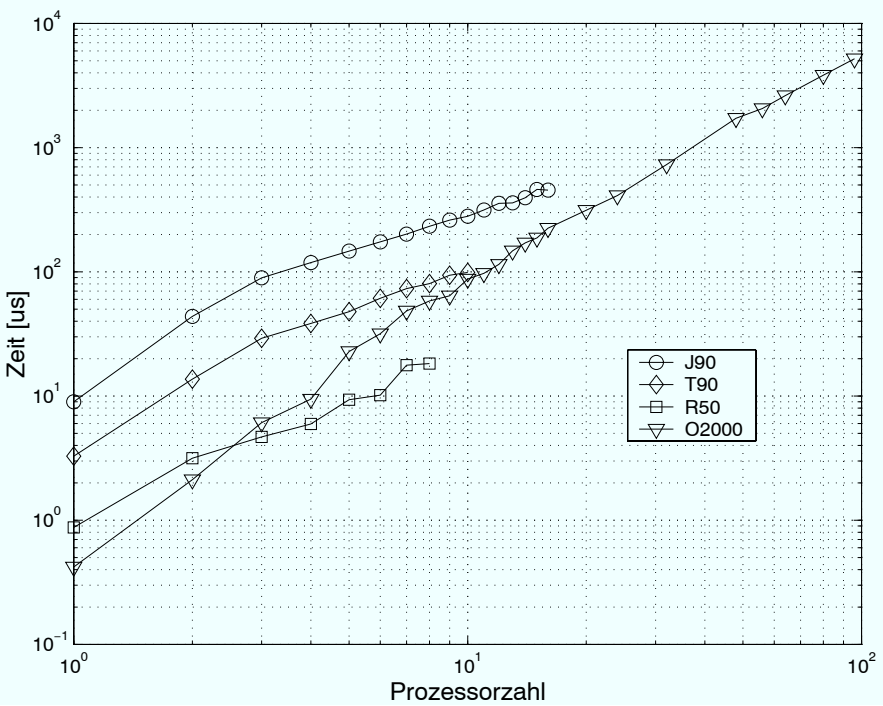

Abbildung 4.5: Konkurrenz um eine kritische Region

Anzahl der Operationen dividiert, um die durchschnittlich benötigte Zeit für eine Operation unter Konkurrenzbedingungen zu ermitteln. Die Kurven für den Zugriff auf eine atomare Variable (`!$OMP ATOMIC`) zeigen einen fast identischen Verlauf.

Die Meßkurve der R50 zeigt einen ähnlichen Verlauf wie die der Cray Systeme, jedoch auf einem wesentlich niedrigeren Zeitniveau. Die Origin2000 benötigt zwar für das Setzen einer Sperre mit nur einer CPU die wenigste Zeit, jedoch zeigt ihre Meßkurve auch den steilsten Anstieg bei wachsender Prozessorzahl. Die Meßkurve der R50 hat mehrere Sprünge. Der erste Sprung begründet sich auf den Wechsel von einer auf zwei CPUs, da mit einem Prozessor keinerlei Synchronisation erforderlich ist. Die weiteren Sprünge erfolgen immer dann, wenn ein weiteres Prozessorboard hinzugenommen wird und die Kommunikation zur Synchronisation über das Verbindungsnetzwerk zwischen den Boards erfolgen muß.

Abbildung 4.6 zeigt den Zeitbedarf zum Setzen einer Sperre unter Konkurrenzbedingungen in Abhängigkeit von der Prozessorzahl. Die Messung erfolgte analog zur Messung der kritischen Region und zeigt nahezu identische Werte. Dies legt die Vermutung nahe, daß für die Implementierung von Sperren, kritischen Regionen und atomaren Zugriffen sehr ähnliche Algorithmen verwendet werden.

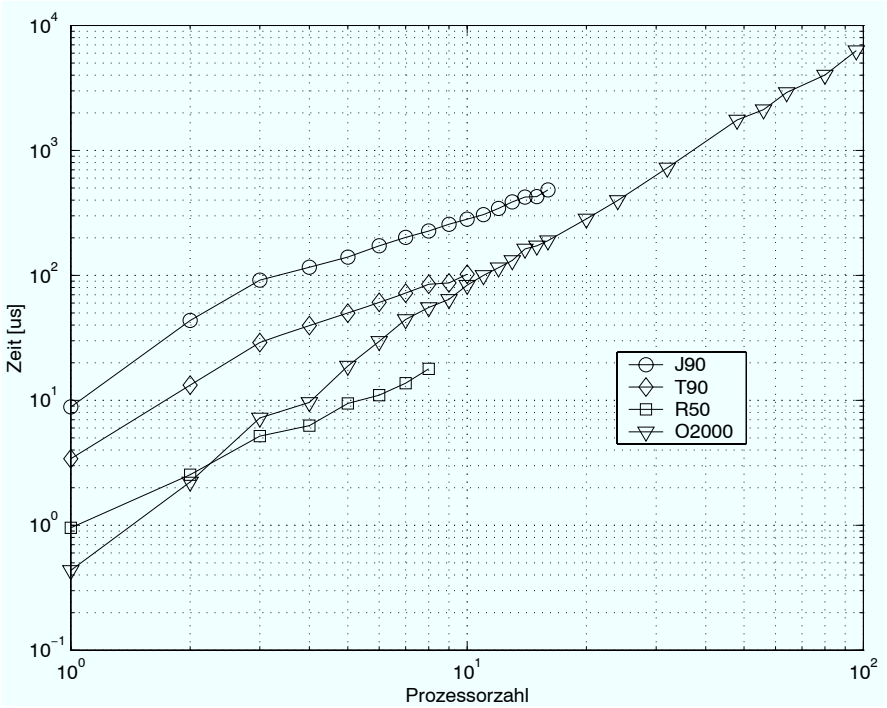

Abbildung 4.6: Konkurrenz um eine Sperre

4.2.6 Reduktionsoperation

In der OpenMP Programmierschnittstelle ist nur die Reduktion skalarer Variablen vorgesehen und stellt, wie auch schon in [15] aufgezeigt, eine relativ restriktive Einschränkung für den Programmierer dar.

Die Reduktionsoperation wurde jeweils mit einer und mit zehn skalaren Variablen in einer einfachen parallelen Schleife mit statischem Schleifenscheduling untersucht. Die Anzahl der ausgeführten Iterationen entsprach der Anzahl der beteiligten Threads in der parallelen Region, so daß jeder Thread eine Iteration ausführen mußte. Jede dieser Schleifen wurde 10.000 mal abgearbeitet und die gemessene Zeit durch 10.000 dividiert. Abbildung 4.7 zeigt die Meßergebnisse für die Reduktion von 10 skalaren Variablen.

Die J90, T90 und R50 haben bei diese Untersuchung relativ ähnliche Ergebnisse gezeigt. Bei der Origin2000 zeigt sich ein steilerer Anstieg der Meßkurve als bei den anderen Maschinen. Besonders hervorzuheben ist hier der sprunghafte Anstieg der benötigten Zeit um den Faktor drei beim Wechsel von 32 auf 48 oder 64 Prozessoren. Die Begründung für dieses Verhalten findet sich in der Hardwarearchitektur der Maschine und wird im folgenden Kapitel bei der Untersuchung der Applikation FIRE näher analysiert.

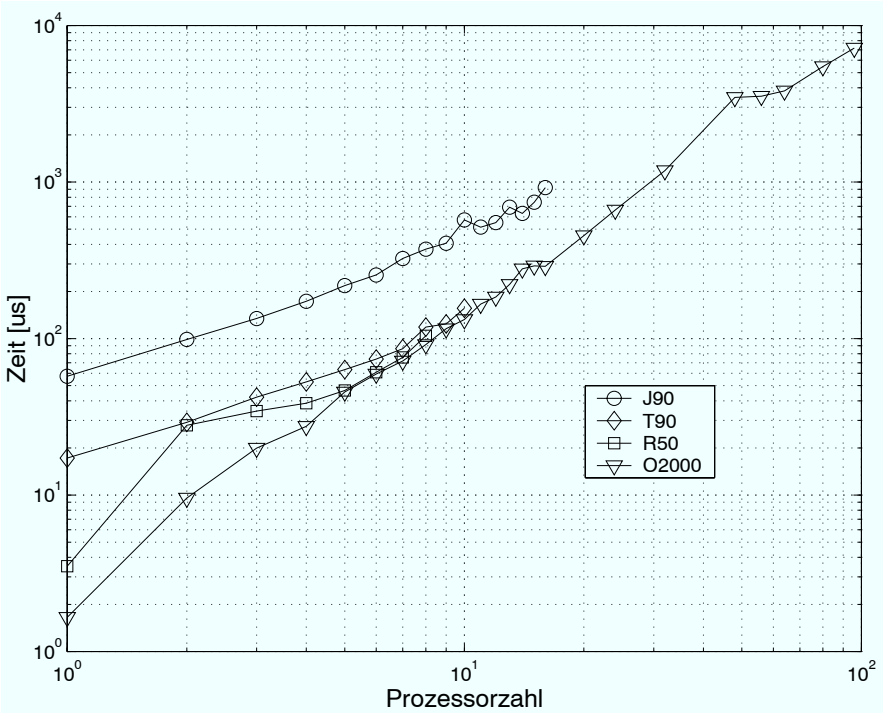

Abbildung 4.7: Reduktionsoperation mit 10 Skalaren

4.2.7 Firstprivate-Konstrukt

Sollen private Kopien von Variablen mit ihren vorherigen Werten initialisiert werden, wird bei der Deklaration einer parallelen Region dazu der Zusatz `FIRSTPRIVATE` für diese Variablen verwendet. Für diese Messung wurde eine 10 KB große Variable benutzt, welche als `FIRSTPRIVATE` deklariert wurde.

Abbildung 4.8 zeigt den Zeitbedarf für die Erstellung einer parallelen Region inklusive der Initialisierung der jeweils 10 KB großen Variablen. Vergleicht man diese Kurven mit denen aus Abbildung 4.1, so ist zu erkennen, daß die Cray J90 und T90 nur wenig mehr Zeit benötigen, was auf das zugrundeliegende schnelle Speichersubsystem der beiden Systeme zurückzuführen ist. Da der Zeitbedarf für den Beginn einer parallelen Region auf der R50 ohnehin schon relativ hoch ist, schlägt sich auch hier der durch die zusätzlichen Initialisierungen verursachte Kommunikationsaufwand nicht nieder. Die Origin2000 reagiert am empfindlichsten auf die `FIRSTPRIVATE` Deklaration bei einer parallelen Region und zeigt im Vergleich den größten Zeitzuwachs. Dies liegt daran, daß die 10 KB der als `FIRSTPRIVATE` deklarierten Variablen beim Beginn der parallelen Region gleichzeitig an alle beteiligten Prozessoren gesendet werden müssen. Dieser Kommunikationsaufwand spiegelt sich in den Meßergebnissen der Origin2000 wieder.

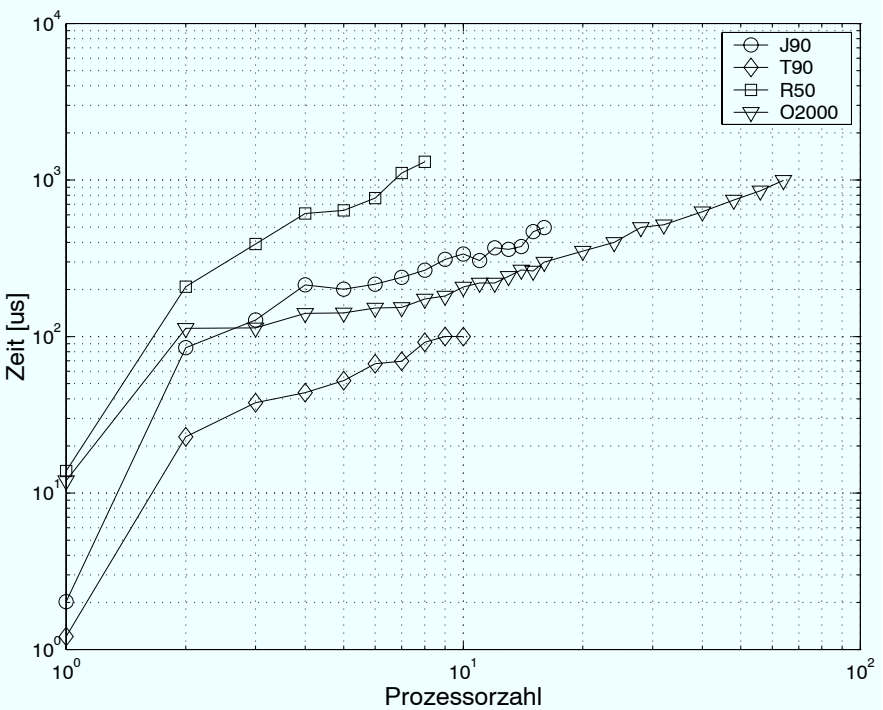

Abbildung 4.8: Parallele Region mit `FIRSTPRIVATE` (10 KB)

4.2.8 Flush-Konstrukt

Mit Hilfe der `FLUSH` Direktive werden Synchronisationspunkte definiert, an denen alle für den Thread sichtbaren gemeinsamen Variablen mit dem Hauptspeicher abgeglichen werden, um eine konsistente Sicht der Variablen zu garantieren.

Für diesen Test wurde eine gemeinsame Variable verwendet, die 10 KB Speicherplatz benötigt. Jeder Prozessor mußte innerhalb einer parallelen Region das folgende Codesegment 10 mal hintereinander ausführen, wobei `me` für die eigene Threadnummer steht.

```
data(me + 1) = data(me)
data(me) = data(me - 1)
!$OMP FLUSH(data)
```

Die Cray J90 und T90 gehören zu den sogenannten UMA-Maschinen und besitzen keine Caches, die mit dem Hauptspeicher abgeglichen werden müßten. Was evtl. abgeglichen werden muß, sind die Inhalte der Prozessorregister.

Abbildung 4.9 zeigt, daß bei cc-NUMA Systemen wie der Origin2000 der Zeitbedarf für die Synchronisation von gemeinsamen Variablen durch `FLUSH` bei steigender Prozessorzahl auf Grund des verursachten Kommunikationsaufwandes (jeder Prozessor mußte bei diesem Test die in seinem Cache befindliche 10 KB große Variable mit dem Hauptspeicher synchronisieren) ansteigt. Im Gegensatz dazu zeigen die Kurven

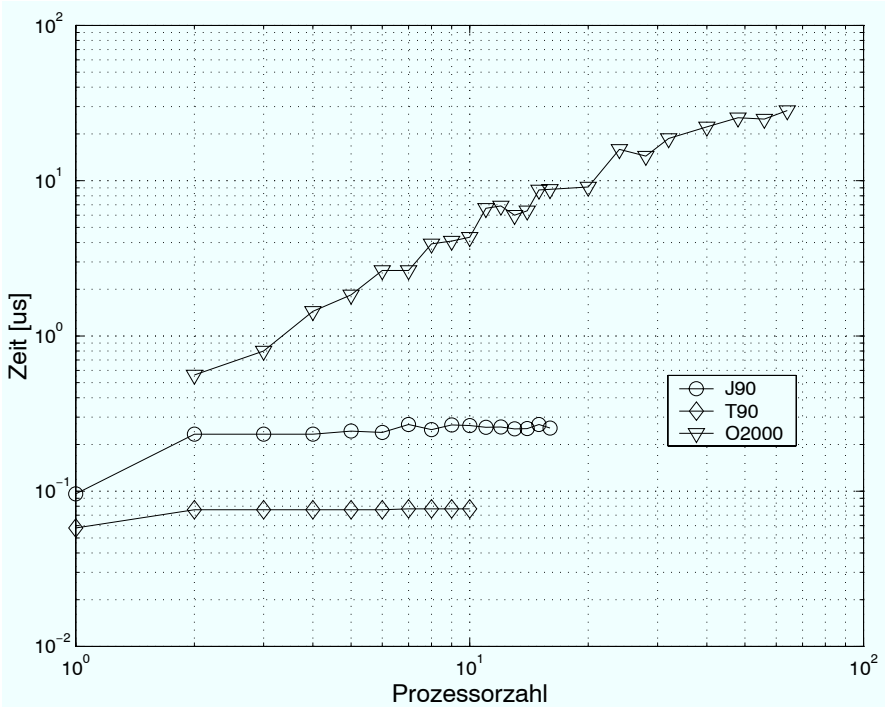

Abbildung 4.9: Zeitbedarf für Flush (10 KB)

der J90 und T90 einen fast waagerechten Verlauf, was auf die fehlenden Caches der beiden Maschinen zurück zu führen ist.

Die Werte der R50 fehlen in dieser Meßreihe, da die `FLUSH` Direktive nicht implementiert ist, obwohl bei jeder Barriere implizit *alle* gemeinsamen Variablen mit dem Hauptspeicher abgeglichen werden.

Kapitel 5

Untersuchung von OpenMP-Anwendungen

Die Leistungsfähigkeit der Programmierschnittstelle OpenMP bei realen Anwendungen soll im folgenden Kapitel genauer untersucht werden. Zu diesem Zweck wurden mehrere bereits parallele Anwendungen auf OpenMP Direktiven umgestellt und der auf den verschiedenen Hardwarearchitekturen erreichte Speedup analysiert.

Die Messungen fanden auf allen bereits vorgestellten Parallelrechnern mit Ausnahme der T90 statt. Auf eine Untersuchung der Programme in Kombination mit der T90 wurde verzichtet, da sich bereits bei der Analyse der OpenMP-Implementierungen zeigte, wie ähnlich sich die Cray T90 und J90 bezüglich der zu erwartenden Performance (bezogen auf den Speedup) sind.

Von jedem im folgenden vorgestellten Programm wurden zwei Versionen erzeugt. Eine OpenMP-Version und eine Version mit den durch SGI zusätzlich zur Verfügung gestellten Direktiven zur Datenverteilung für weitere Untersuchungen auf der Origin2000.

5.1 Shallow Water Equations Benchmark

Das zuerst untersuchte Programm „SHALLOW“ ist der *Shallow Water Equations Benchmark*, welcher ein finites Differenzenschema benutzt. Die zu berechnende Diskretisierung wurde auf Gittern unterschiedlicher Größe untersucht. Als Grundlage dient eine mit SVM-Fortran Direktiven [18, 19] parallelisierte Version.

Für die Berechnung mit einer vorgegebenen Gittergröße (z. B. 1024 x 1024) werden in dem Programm SHALLOW zu Beginn 14 Matrizen der entsprechenden Größe (z. B. 14 Matrizen a 1024 x 1024 Elemente) vereinbart. Das komplette Programm besteht aus einer einzigen parallelen Region, in der diese Marizen als gemeinsame Variablen deklariert sind. Zu Beginn werden alle Matrizen in parallelen Schleifen

mit Werten initialisiert, bevor die eigentliche Berechnung mit einer vorgegebenen Iterationszahl startet. Die Berechnung selbst wird von einem Programmblock ausgeführt, der fast ausschließlich aus parallelen Schleifen besteht. Für alle parallelen Schleifen wird statisches Schleifenscheduling verwendet. Nachfolgend ist eine dieser Initialisierungsschleifen abgebildet.

```
!$OMP DO DEFAULT(SHARED) PRIVATE(i,j) SCHEDULE(STATIC)
  DO j=1, IY
    DO i=1, IX
      U(i,j) = 0D0
      V(i,j) = 0D0
      ...
    END DO
  END DO
!$OMP END DO
```

Für die Origin2000 wurde eine zweite Programmversion erstellt, bei der alle Matrizen zu Beginn in parallelen Schleifen mit Werten initialisiert werden. Bei diesen Initialisierungsschleifen wurde statisches Schleifenscheduling verwendet. Die Reihenfolge der Variablen in den Initialisierungsschleifen wurde so gewählt, daß diese mit Zugriffsreihenfolge in der in der Berechnung bestmöglich übereinstimmt. Die vom Betriebssystem der Origin2000 standardmäßig verwendete *first touch*-Strategie sorgt für die Verteilung der in den parallelen Schleifen initialisierten Variablen über die lokalen Hauptspeicher der Prozessoren. Die erreichte Verteilung der Daten entspricht einer (*, BLOCK)-Verteilung, da bei der Initialisierung die zweite Dimension der Matrizen parallel initialisiert wird. Bei allen anderen parallelen Schleifen wurde die Schedulingstrategie mit Hilfe der zusätzlichen SGI-Direktive `AFFINITY` folgendermaßen beeinflußt.

```
!$OMP DO DEFAULT(SHARED) PRIVATE(i,j)
!$SGI+AFFINITY(j) = DATA(UOLD(i,j))
  DO j=1, IY
    DO i=1, IX
      UOLD(i,j) = U(i,j)
      VOLD(i,j) = V(i,j)
      ...
    END DO
  END DO
!$OMP END DO
```

Dies führt dazu, daß die Iteration `j` auf dem Prozessor ausgeführt wird, der das Element `UOLD(i,j)` in seinem lokalen Speicher hat. Das Scheduling in Abhängigkeit einer Datenverteilung zu wählen, vermeidet entfernte Datenzugriffe und damit Speicherkonflikte.

Insbesondere mit großen Gittern skaliert der Shallow Water Equations Benchmark sehr gut. Auf der Origin2000 skaliert die speziell angepaßte Version gleich gut wie die reine OpenMP-Version, ist jedoch absolut gesehen etwas langsamer.

Dieses Verhalten liegt daran, daß die Überprüfung des Speicherortes einer gemeinsamen Variablen mittels `!$SGI+AFFINITY(...) = DATA(...)` einen gewissen Overhead erzeugt, der in der reinen OpenMP-Version nicht existiert, da hier ausschließlich die Schedulingstrategie `STATIC` verwendet wurde. Das Betriebssystem der Origin2000 sorgt dafür, daß die in einer parallelen Region benutzten gemeinsamen Variablen normalerweise in dem lokalen Speicher des Prozessors abgelegt werden, der diese Variable als erstes benutzt. Da der Benchmark so implementiert ist, daß die gemeinsamen Variablen immer in der gleichen Reihenfolge von dem gleichen Prozessor verwendet werden, ergibt sich für die OpenMP-Version ganz automatisch die gleiche Speicherverteilung wie bei der speziell für die Origin2000 angepaßten Version.

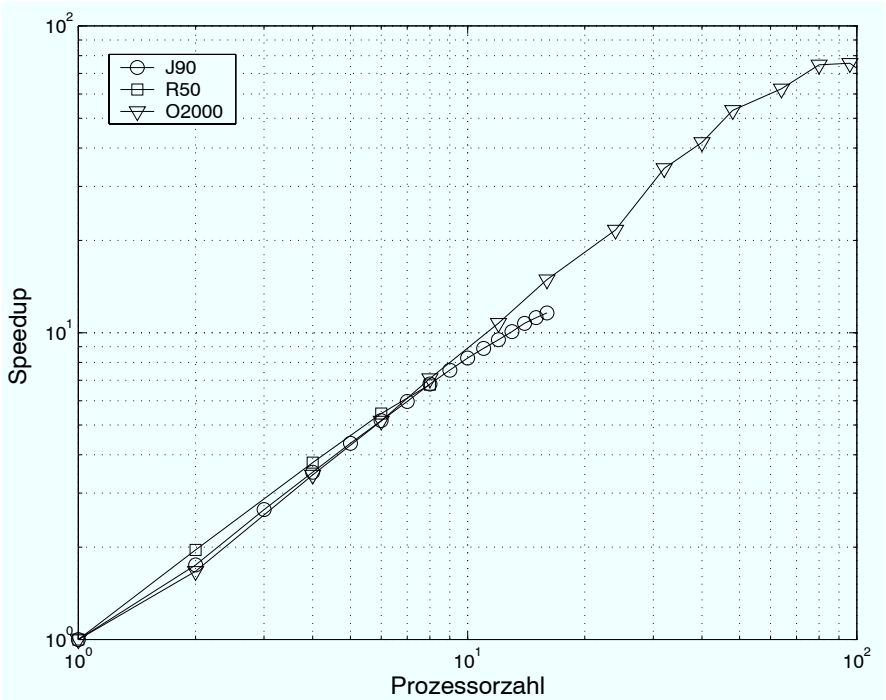

Abbildung 5.1: Shallow: Gittergröße 1024 x 1024

Abbildung 5.1 zeigt eine Messung des Shallow Water Equations Benchmarks auf einer Gittergröße von 1024 x 1024. Die J90 und R50 zeigen einen sehr guten Speedup über alle Prozessoren. Die Origin2000 erreicht ab 30 Prozessoren superlinearen Speedup und zeigt bei 80 Prozessoren noch nahezu linearen Speedup. Mit mehr als 80 Prozessoren kann keine weitere Verbesserung des Speedups erreicht werden.

Der superlineare Speedup ist auf die Caches der Origin2000 zurückzuführen, welche die für die Berechnung nötigen Variablen komplett enthalten. Die für die Berechnung verwendeten parallelen Schleifen enthalten bei dieser Gittergröße nicht genug Arbeit, um mehr als 100 Prozessoren ausreichend zu beschäftigen (Rechenzeit mit

96 Prozessoren: 0,86 Sekunden). Daher hebt der anfallende Synchronisations- und Kommunikationsaufwand mit mehr als 100 Prozessoren in diesem Fall die erreichte Zeitersparnis durch die parallele Berechnung auf und verhindert einen Speedup über 75,55. Eine Kontrollmessung auf der Origin2000 mit einer Gittergröße von 2048 x 2048 zeigte eine sehr gute Performance und erreichte bei 128 Prozessoren einen Speedup von 126,3. Dies liegt daran, daß die einzelnen Schleifen bei dieser Problemgröße noch genügend Arbeit enthalten. Eine wesentliche Steigerung des Speedups ist bei einer weiteren Erhöhung der Prozessorzahl auch mit dieser Problemgröße nicht zu erwarten, da die einzelnen parallelen Schleifen bei einer Berechnung auf 128 Prozessoren nur noch wenig Arbeit enthalten (Rechenzeit mit 128 Prozessoren: 1,24 Sekunden).

Die Messung für das Gitter mit der Auflösung 2048 x 2048 fehlt bei den anderen Rechnersystemen, da das Programm SHALLOW mit dieser Problemgröße weder auf der J90 noch auf der R50 ausgeführt werden kann.

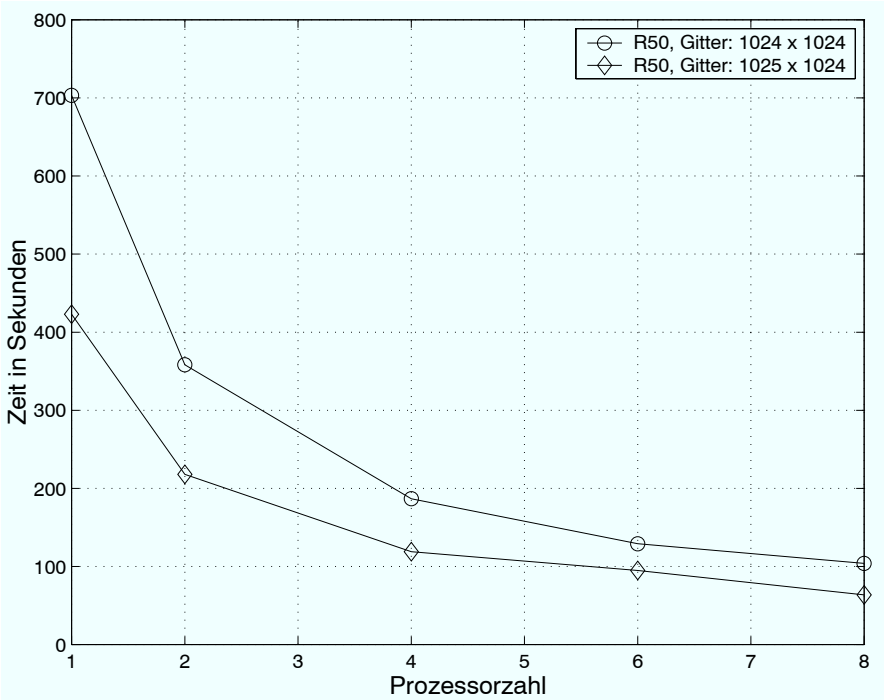

Abbildung 5.2: Shallow: verschiedene Gittergrößen

Ein interessantes Ergebnis lieferte der Programmlauf mit einer Gittergröße von 1025 x 1024. Die erreichten Speedups sind nahezu identisch mit den erreichten Werten bei einer Gittergröße von 1024 x 1024, jedoch unterscheiden sich die Programmlaufzeiten zum Teil erheblich. Abbildung 5.2 zeigt die benötigte Gesamtlaufzeit für die Berechnung auf der IBM R50 mit jeweils einer Gittergröße von 1024 x 1024 und 1025 x 1024. Die Unterschiede in der Gesamtlaufzeit sind bei den anderen untersuchten Parallelrechnern nicht ganz so gravierend.

Die Origin2000 benötigt zur Abarbeitung des Programms SHALLOW mit einer Gittergröße von 1025 x 1024 nur ca. 95% der Zeit wie für die Berechnung mit einem 1024 x 1024 großen Gitter. Die J90 benötigt das etwas größer Gitter noch ca. 90% der Zeit. Bei der IBM R50 zeigt sich mit ca. 60% die stärkste Abweichung (Abbildung 5.2) in der Gesamtlaufzeit.

Die Erklärung für dieses Verhalten ist in der Organisation der Hauptspeicher und Caches der Rechner in Kombination mit der Problemgröße zu finden. Die Größe einer Hauptspeicherseite oder einer Cacheline ist immer eine Zweierpotenz. Falls der zur Berechnung eines Problems benötigte Speicherplatz ebenfalls eine Zweierpotenz ist, kann es zu Problemen bei Speicherzugriffen kommen, die u. U. ein ständiges Ein- und Auslagern verschiedener Cachelines zu Folge haben. In diesem speziellen Fall scheint die Cachegröße von 2 Megabyte bei der IBM R50 nicht ausreichend, um wirksam ein ständiges Ein- und Auslagern von Cachelines zu vermeiden.

Abschließend kann gesagt werden, daß für die Applikation SHALLOW eine sehr gute Parallelisierung erreicht wurde, die sich in entsprechenden Speedupwerten wiederspiegelt. Eine weitere Verbesserung konnte durch die speziell für die Origin2000 angepaßte Version nicht erreicht werden, da die Datenverteilung auch für die reine OpenMP-Version bereits optimal war.

5.2 Molekulardynamiksimulation in Flüssigkeiten

Die als nächstes untersuchte Anwendung „MCMD“ wurde am Forschungszentrum Jülich entwickelt und ist dazu in der Lage, Startkonfigurationen für Molekular- und Monte Carlo Simulationen mit großen Teilchenzahlen zu erzeugen [20]. Die Steuerung der Applikation MCMD erfolgt durch zwei Textdateien, die alle für die Berechnung benötigten Teilchendaten und Simulationsoptionen beinhalten.

Das Programm ist in C implementiert und existiert als schared memory Version für SUN, Cray und SGI wie auch als MPI-Version. Eine sequentielle Version ist ebenfalls verfügbar. Die Übersetzung des Programms wird über in einem Makefile eingestellte Compileroptionen gesteuert. Als Grundlage für die Portierung des Programms MCMD dient die auf SGI Direktiven basierende Version.

Die Untersuchung dieser Applikation wurde ausschließlich auf der SGI Origin2000 gemacht, da das OpenMP C und C++ API bisher nur durch den MIPSpro Compiler der Firma SGI in der Version 7.3 unterstützt wird.

Zu Beginn der Programmabarbeitung werden das vorgegebene Volumen und die vorgegebene Teilchenzahl in Subsysteme unterteilt, die nicht mehr als 1.000 Teilchen enthalten dürfen. Für die weitere Berechnung wird zunächst nur ein solches Subsystem betrachtet, welches Teil des Gesamtsystem ist. Die dem Subsystem zugeteilten Teilchen werden in dem Teilvolumen zufällig verteilt. Mit Hilfe von mehreren Monte Carlo Simulationsschritten werden die Teilchen in dem Teilvolumen so verteilt, daß sich die einzelnen Atome nicht mehr überlappen. Das Gesamtsystem wird aus

dem berechneten Subsystem zusammengesetzt und führt zu einem System, in dem das Subsystem periodisch immer wieder vorkommt. Anschließend werden mehrere Molekulardynamiksimulationsschritte auf dem Gesamtsystem ausgeführt, um zu einem System zu gelangen, das keinerlei periodisch wiederkehrende Subsysteme mehr enthält.

Die Molekulardynamiksimulation wird durch die Funktion *MD_Rechne* ausgeführt und hat folgende Struktur.

```
/* Vorarbeiten */
...
/* Molekulardynamikzyklen /*
for ( i=1 ; i<=Schrittzahl ; i++)
{
  ...
  FORCEcell();
  ...
  /* Auswertung */
  ...
}
```

Dies hat zur Folge, daß die Funktion *FORCEcell*, welche die zwischen den einzelnen Teilchen auftretenden Kräfte berechnet, für jeden Molekulardynamikschritt separat aufgerufen wird.

Abbildung 5.3 zeigt die ermittelten Speedupwerte für 100 Molekulardynamikschritte mit 100.000 Molekülen. Die erreichte Performance ist mit einem maximalen Speedup in der Größenordnung um drei sehr schlecht.

Dieses Verhalten liegt an dem ungünstigen Verhältnis zwischen der benötigten Rechenzeit in sequentiellen bzw. parallelen Teilen des Programms. Zur Berechnung eines Molekulardynamikschritts wird die Funktion *FORCEcell* verwendet und für die Berechnung von 100 Schritten auch 100 mal aufgerufen. Diese Funktion hat im sequentiellen Fall einen Anteil von ca. 90% an der gesamten Rechenzeit und ist zu großen Teilen parallelisiert.

Ein Aufruf der Funktion *FORCEcell* benötigt im sequentiellen Fall 10,54 Sekunden, von denen jedoch nur 9,31 Sekunden innerhalb der in dieser Funktion enthaltenen parallelen Region anfallen. Die restlichen Sekunden werden in einer Initialisierungsschleife (0,01 Sekunden) vor und einer Schleife (1,22 Sekunden) nach der parallelen Region verbraucht.

Zur weiteren Analyse der Funktion wurden für alle enthaltenen Schleifen die jeweils benötigte Zeit bei Programmläufen mit verschiedenen Prozessorzahlen ermittelt. Dies zeigte, daß die parallele Region einen guten Speedup erreicht und folglich für Optimierungen nicht weiter betrachtet werden muß. Der Versuch, die im Anschluß an die parallele Region auszuführende Schleife durch Direktiven zu parallelisieren,

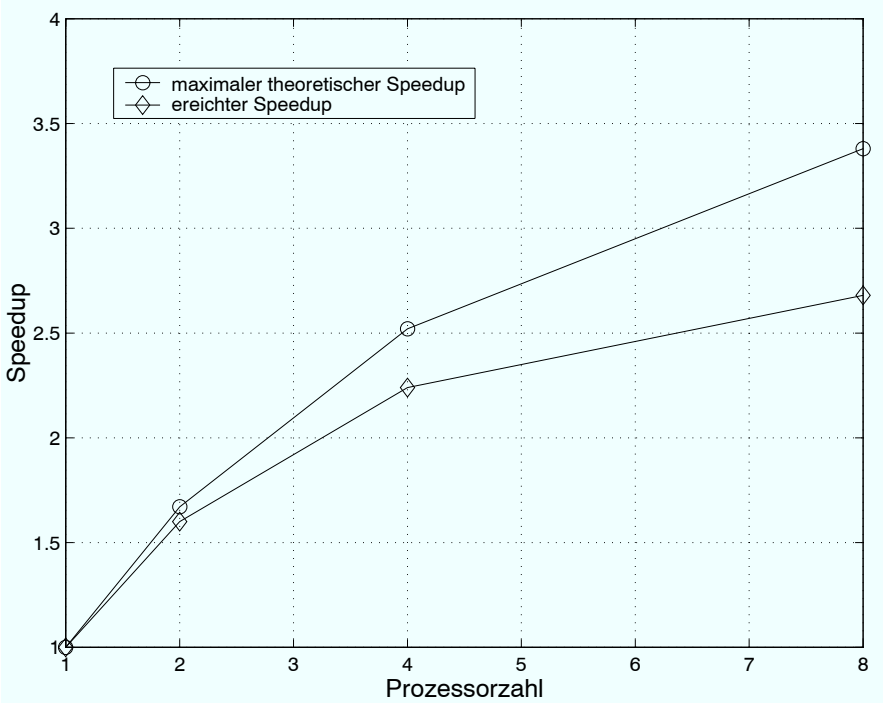

Abbildung 5.3: 100 MD-Schritte mit 1000.000 Molekülen

führte nicht zum Erfolg. Eine genauere Betrachtung dieser Schleife mit Hilfe eines Profilingtools zeigte einen rapiden Anstieg von Cache Misses, sobald mehr als eine CPU verwendet wurde.

Die Schleife nach der parallelen Region besteht aus zwei ineinander geschachtelte Schleifen und ist nachfolgend abgebildet.

```
for ( i=0 ; i<ncell3 ; i++ )
{
  for ( j=0 ; j<ACM ; j++ )
  {
    index_j = acid[i][j];
    b[index_j].fx += acfx[i][j];
    b[index_j].fy += acfy[i][j];
    b[index_j].fz += acfz[i][j];
  }
}
```

Die innere Schleife enthält einen indirekten Zugriff auf das Feld `b` über den Index `index_j`. Dieser indirekte Zugriff auf des Feld `b` ist für die steigende Zahl von Cache Misses verantwortlich, da aufgrund der indirekten Zugriffe die Prozessoren nicht auf die Feldelemente zugreifen, die sich in ihren lokalen Caches oder Hauptspeichern befinden. Eine passende Verteilung für das Feld `b` zu finden ist jedoch problematisch,

da der Index mit jedem Molekulardynamikschritt neu bestimmt wird. Ein weiteres Problem besteht darin, daß auch für verschiedene Paare (`i, j`) durchaus Zugriffe auf die gleiche Position im Feld `b` erfolgen können, da die Matrix `acid` doppelte Werte enthält. Dies bedeutet, daß der Zugriff auf das Feld `b` atomar erfolgen muß, sollen falsche Ergebnisse bei der Berechnung vermieden werden. Die bei einer Parallelisierung dieser beiden Schleifen nötigen atomaren Zugriffe und die unter Umständen falsche Speicherverteilung des Feldes `b` verursachen jedoch mehr Overhead, als durch die parallele Abarbeitung gewonnen wird.

Der in Abbildung 5.3 eingetragene theoretisch maximal zu erreichende Speedup läßt sich aufgrund der Laufzeiten im sequentiellen Fall, wie nachfolgend beschrieben, berechnen. Die Applikation benötigt bei sequentieller Abarbeitung durch einen einzelnen Thread insgesamt 1142,61 Sekunden zur Berechnung, von denen 919,01 Sekunden in der in *FORCEcell* enthaltenen parallelen Region anfallen. Wird für die parallele Region ein linearer Speedup angesetzt, so lassen sich die in Tabelle 5.1 angegebenen Werte errechnen.

Prozessorzahl	Zeit in paralleler Region	Zeit in sequentiellen Teilen	Gesamtzeit	Speedup
1	919,01	223,60	1142,61	1,00
2	459,51	223,60	683,10	1,67
4	229,75	223,60	453,35	2,52
8	114,88	223,60	338,48	3,38

Tabelle 5.1: Erreichbarer Speedup der Applikation MCMD

5.3 FIRE

Das Programm „FIRE“ ist ein Löser für lineare Gleichungssysteme nach dem CG-Verfahren aus dem Programmpaket FIRE der Firma AVL. Die Diskretisierung des Volumens bestimmt ein Gitter mit 318.044 Gitterknoten, welches für die Messung auf den verschiedenen Parallelrechnern herangezogen wurde. Die resultierenden Matrizen sind sehr dünn besetzt und werden daher komprimiert gespeichert. Der Zugriff auf eine solche Matrix erfolgt durch indirekte Adressierung.

Die für die Berechnung benötigten Variablen werden zu Beginn der Applikation FIRE vereinbart und teilweise mit den Werten des für die Berechnung eingelesenen Datensatzes initialisiert. Das Programm besteht, mit Ausnahme der Ein- und Ausgaberoutinen, aus einer einzigen parallelen Region, in der die meisten Variablen gemeinsam genutzt werden. Die restlichen, durch die Einleseroutine noch nicht initialisierten Variablen, werden in der parallelen Region durch parallele Schleifen initialisiert. Die eigentliche Berechnung erfolgt in einer Schleife, die erst dann terminiert, wenn eine vorgegebene Genauigkeit für das zu berechnende Ergebnis unterschritten wird. Diese Berechnungsschleife enthält fast ausschließlich parallelisierte

Schleifen, die teilweise Reduktionsoperationen ausführen. Für alle parallelisierten Schleifen wurde die Schedulingstrategie `STATIC` verwendet.

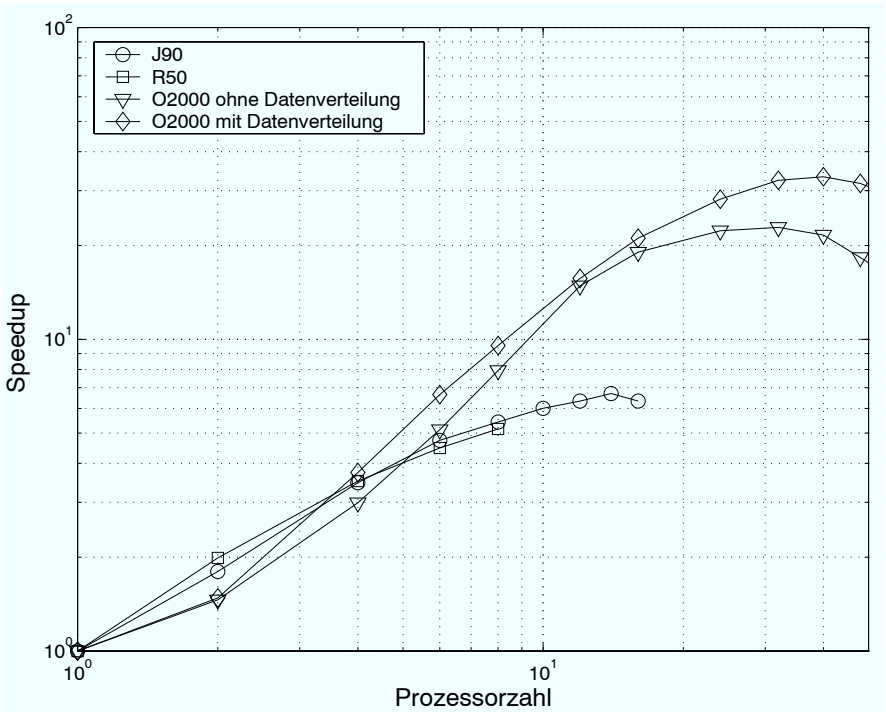

Abbildung 5.4: Datensatz: Cojack (318.044 Gitterknoten)

Abbildung 5.4 zeigt die Speedupkurven für den FIRE-code mit dem Datensatz Cojack (318.044 Gitterknoten). Die Speedupkurven der J90 und R50 sind nahezu identisch. Die R50 erreicht einen maximalen Speedup von 5,16 und die J90 von 6,70. Die Origin2000 zeigt bei der Messung ohne explizite Datenverteilung ab 8 Prozessoren superlinearen Speedup, was auch in diesem Fall auf die Caches der Maschine zurückzuführen ist. Ab 24 Prozessoren beginnt die Speedupkurve abzuflachen, um dann ab 32 Prozessoren einzubrechen.

Ungefähr die Hälfte der in dem Programm FIRE benutzten Variablen wird in sequentiellen Programmteilen initialisiert, so daß diese Variablen im lokalen Hauptspeicher der ersten CPU angelegt werden und während einer parallelen Berechnung von anderen Prozessoren, zumindest beim ersten Zugriff, nur durch einen entfernten Zugriff in den Speicher des ersten Prozessors erreicht werden können. Dies legte nahe, mit der Performanceoptimierung bei der Verteilung der gemeinsamen Variablen zu beginnen. Diese Variablen wurden unter Ausnutzung der first touch Strategie des Betriebssystems der Origin2000 über die lokalen Speicher der Prozessoren verteilt, indem alle Variablen in einer parallelen Schleife mit statischen Schleifenscheduling initialisiert wurden. Das Schleifenscheduling der restlichen Schleifen wurde mit der ergänzenden SGI-Direktive `AFFINITY = DATA(...)` beeinflußt. Dies hat zur Folge, daß die Iterationen der betreffenden Schleife durch die Prozessoren ausgeführt werden, die die zur Berechnung benötigten Variablen im lokalen Speicher haben.

Die Speedupkurve der Origin2000 mit expliziter Datenverteilung zeigt nun schon ab 6 Prozessoren superlinearen Speedup. Die Kurve flacht erst bei 32 Prozessoren ab.

Als Ursache für den Einbruch des Speedups wurden mehrere parallele Schleifen identifiziert, in denen eine Reduktion in eine skalare Variable ausgeführt wird. Der Zeitbedarf für diese Reduktionen steigt (wie bei der Untersuchung der OpenMP-Primitive) beim Wechsel von 32 auf 64 Prozessoren sprunghaft um den Faktor drei an. Für weitere Untersuchungen wurde das Profilingtool *ssrun* aus dem Programmpaket *SpeedShop* [13] verwendet. Ein Profilinglauf mit jeweils 32 bzw. 64 Prozessoren zur Bestimmung der L2 Cache Misses zeigte keine signifikante Veränderung, so daß eine falsche Datenverteilung hier nicht die Ursache für den sprunghaften Anstieg des Zeitbedarfs für die Reduktionen sein kann. Der Profilinglauf ergab jedoch, daß der FIRE-Code mit 32 Prozessoren 7,7% der Gesamtlaufzeit damit verbringt, auf einen Lock zu warten. Bei 64 Prozessoren stieg die Wartezeit um etwa den Faktor drei und betrug 26,3% der Gesamtlaufzeit.

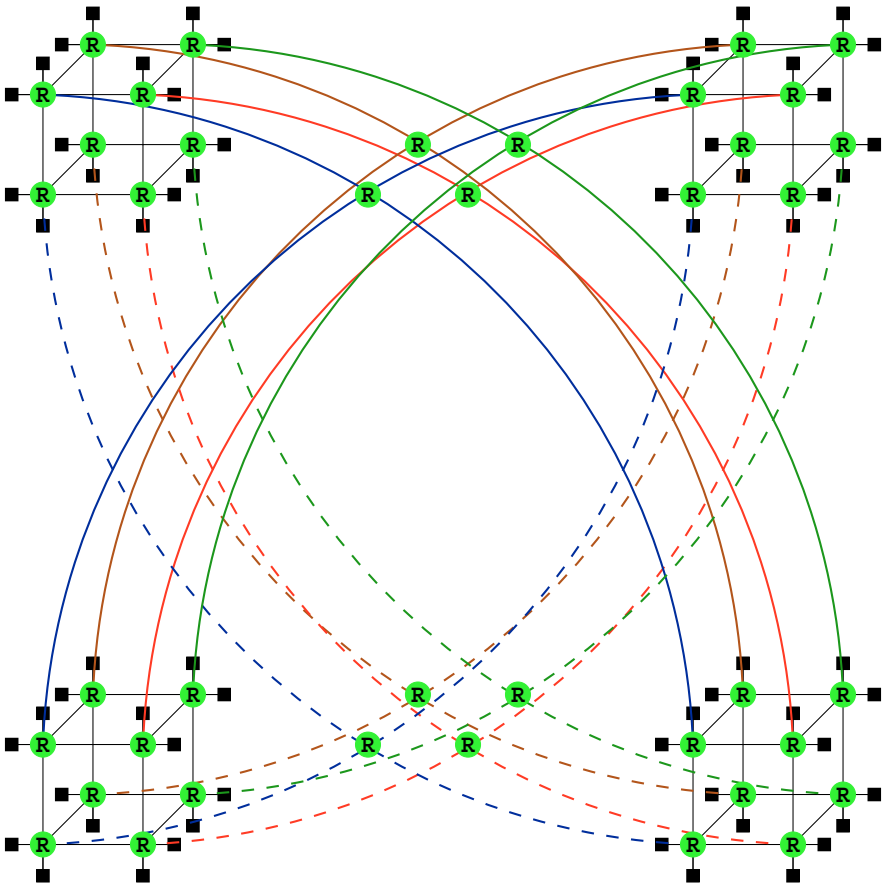

Abbildung 5.5: Konfiguration einer Origin2000 mit 128 CPUs

Abbildung 5.5 zeigt die Konfiguration einer Origin2000 mit 128 Prozessoren [12]. Die schwarzen Quadrate stellen Rechenknoten dar, die aus jeweils zwei Prozessoren mit lokalem Speicher bestehen. Die Verbindung der einzelnen Rechenknoten untereinander wird durch die hier mit „R“ gekennzeichneten Router realisiert, wobei jeder

Router maximal zwei Verbindungen zu Rechenknoten und vier zu Routern haben kann. Bei einer Konfiguration mit 128 Prozessoren werden daher jeweils 32 Prozessoren in einem Hypercube zusammengefaßt und eine Verbindung zwischen diesen Hypercubes über sogenannte Metarouter realisiert. Die Verbindung mit Hilfe der zusätzlichen Metarouter ist nötig, da die in den Hypercubes befindlichen Router nicht mehr genügend Verbindungen frei haben.

Eine Reduktion mit bis zu 32 Prozessoren kann innerhalb eines solchen Hypercubes abgewickelt werden und benötigt keinerlei Verbindungen zwischen Prozessoren über einen Metarouter. Bei mehr als 32 Prozessoren ist eine Synchronisation der beteiligten CPUs über die Metarouter hinweg erforderlich, so daß der Zeitbedarf für diese Synchronisation durch den zusätzlichen Hop über den Metarouter sprunghaft ansteigt.

Abschließend kann gesagt werden, daß insbesondere die speziell für die Origin2000 angepaßte Version der Applikation FIRE mit bis zu 32 Prozessoren eine sehr gute Performance aufweist. Die reine OpenMP-Version erreicht aufgrund einer nicht ganz optimalen Datenverteilung keinen so guten Speedup, wie die für die Origin2000 angepaßte Version. Dies macht deutlich, wie wichtig die von SGI zur Verfügung gestellten Direktiven zur Datenverteilung auf Parallelrechnern mit verteiltem Speicher sind, die in Kombination mit den OpenMP-Direktiven benutzt werden können.

Kapitel 6

Zusammenfassung

Die im Rahmen dieser Diplomarbeit untersuchte Programmierschnittstelle OpenMP ermöglicht es, portable SMP-Programme für Parallelrechner mit gemeinsamem Speicher zu entwickeln. Das OpenMP zugrundeliegende Programmiermodell ist leicht verständlich und basiert auf Direktiven und Laufzeitroutinen, die in den Quellcode eingefügt werden. Mit Hilfe dieser Direktiven kann ein sequentielles Programm sukzessive in ein paralleles Programm für Parallelrechner mit gemeinsamem Speicher transformiert werden. Als Ansatzpunkt für die Parallelisierung dienen in erster Linie Zählschleifen, obwohl OpenMP auch die Definition von parallelen Regionen und parallelen Sektionen vorsieht. Besonders hervorzuheben ist, daß alle parallelen Konstrukte auch ineinander geschachtelt verwendet werden dürfen, sowie weiterhin die Möglichkeit besteht, verwaiste Direktiven zu verwenden. Solche Direktiven finden sich in Unterprogrammen, die aus parallelen Regionen wie auch aus sequentiellen Programmteilen heraus aufgerufen werden können und steuern ggf. die parallele Abarbeitung (z. B. einer Schleife) in diesem Unterprogramm.

Das OpenMP Fortran API wird zur Zeit dieser Untersuchung relativ gut von existierenden Compilern unterstützt, obwohl nicht immer alle Direktiven bzw. Laufzeitroutinen von allen Compilerherstellern implementiert wurden. Die verschiedenen Hersteller haben aber angekündigt, die Unterstützung für die Programmierschnittstelle OpenMP weiter zu verbessern.

OpenMP zeigt eine gute Performance auf allen getesteten Hardwareplattformen. Die bei der Analyse der einzelnen OpenMP-Primitive gefundenen Probleme sind vermutlich durch Modifikationen in den OpenMP Bibliotheksroutinen zu beheben. Mit Ausnahme der Origin2000 war der durch das Starten einer parallelen Region erzeugte Overhead relativ hoch, so daß bei der Implementierung von OpenMP-Programmen darauf geachtet werden sollte, große parallele Regionen zu definieren und die weitere Steuerung des Programmablaufs durch verwaiste Direktiven zu realisieren. Als problematisch haben sich parallele Schleifen in Kombination mit dynamischem Schleifenscheduling auf der Origin2000 mit großen Prozessorzahlen herausgestellt, wenn die einzelnen Schleifeniterationen nur sehr wenig Arbeit enthalten. Weiterhin ist noch zu erwähnen, daß Reduktionsoperationen auf der Origin2000 mit mehr als 32

Prozessoren einen erhöhten Zeitbedarf bei der Synchronisation mit sich bringen und nach Möglichkeit vermieden werden sollten.

Der auf OpenMP-Direktiven umgestellte *Shallow Water Equations Benchmark* zeigte eine sehr gute Performance. Dieses Programm enthält nur eine parallele Region und die zur Berechnung benötigten parallelen Schleifen weisen statisches Scheduling auf, so daß nur sehr wenig Overhead durch die parallelen Konstrukte erzeugt wird. Desweiteren ist die Datenverteilung auch für ein System mit physikalisch verteiltem Speicher (Origin2000) sehr günstig. Dies hat zur Folge, daß nur sehr wenige Konflikte beim Datenzugriff auftreten und die Performancewerte negativ beeinflussen. Bei dem Programm FIRE konnte die Performance mit Hilfe der speziellen SGI-Direktiven zur expliziten Datenverteilung bei Parallelrechnern mit verteiltem gemeinsamen Speicher verbessert werden. Dennoch brach der Speedup bei mehr als 32 Prozessoren ein. Dies lag an Synchronisationsproblemen in Schleifen die, eine Reduktion ausführten.

Abschließend kann gesagt werden, daß OpenMP sich gut zur Implementierung von portablen Programmen für Parallelrechner mit gemeinsamem Speicher eignet, sofern diese Systeme nur über kleinere bis mittlere Prozessorzahlen verfügen. Für Systeme mit großen Prozessorzahlen und verteiltem gemeinsamen Speicher (wie die getestete Origin2000) sind Direktiven zur expliziten Datenverteilung wichtig und sollten daher mit in die Programmierschnittstelle OpenMP aufgenommen werden.

Literaturverzeichnis

[1] OpenMP Architecture Review Board
Fortran Application Program Interface
Technical Report, Version 1.0, Oktober 1997
http://www.openmp.org

[2] OpenMP Architecture Review Board
C und C++ Application Program Interface
Technical Report, Version 1.0, Oktober 1998
http://www.openmp.org/

[3] OpenMP Architecture Review Board
OpenMP: A Proposed Industry Standard API for Shared Memory Programming
Technical Report, Oktober 1997
http://www.openmp.org/mp-documents/paper/paper.html

[4] OpenMP Architecture Review Board
Fortran interpretations
Technical Report, Version 1.0, April 1999
http://www.openmp.org/index.cgi

[5] SUN Microsystems
Fortran 77 5.0 - Fortran 90 2.0
ISBN 805-4940, 1999, S. 133-174

[6] Cray Research
CF90 Commands and Directives Reference Manual
Version SR-3901 3.0.2, 1998, S. 107-128

[7] Silicon Graphics
MIPSpro Auto-Parallelizing Option Programmer's Guide
Technical Publications Library, document number: 007-3572-002
1998, Kapitel 1-3
http://www.techpubs.sgi.com

[8] Silicon Graphics
Fortran 77 Programmer's Guide
Technical Publications Library, document number: 007-0711-060
1994, Kapitel 5
http://www.techpubs.sgi.com

[9] Silicon Graphics
MIPSpro Fortran 77 Programmer's Guide
Technical Publications Library, document number: 007-2361-007
1998, Appendix B
http://www.techpubs.sgi.com

[10] Silicon Graphics
MIPSpro 7 Fortran 90 Commands and Directives Reference Manual
Technical Publications Library, document number: 007-3696-003
April 1999, Kapitel 5
http://www.techpubs.sgi.com

[11] Silicon Graphics
C Language Reference Manual
Technical Publications Library, document number: 007-0701-130
April 1999, Kapitel 10-12:
http://www.techpubs.sgi.com

[12] Silicon Graphics
Origin Servers Technical Report
April 1997, Kapitel 2
http://www.techpubs.sgi.com

[13] Silicon Graphics
SpeedShop User's Guide
Technical Publications Library, document number: 007-3311-006
1999
http://www.techpubs.sgi.com

[14] IBM
XL Fortran for AIX User's Guide
SC09-2719-00, Version 6 Release 1,
Dezember 1998

[15] Ragnhild Blikberg, Tor Sørevik:
Early experiences with OpenMP on the Origin 2000
Fourth European SGI/Cray MPP Workshop
September 10-11, 1998, IPP, Garching, Germany, S. 166 - 177
http://www.rzg.mpg.de/mpp-workshop/papers/ipp-report.html

[16] Paul Graham:
OpenMP A Parallel Programming Model for Shared Memory Architectures
EPCC TEC-WATCH Report, Version 0.3, November 1998
http://www.epcc.ed.ac.uk/epcc-tec/documents.html

[17] Klaas Jan Wierenga:
Survey of Compiler Directives for Shared Memory Programming
EPCC TEC-WATCH Report, Version 1.0, März 1997
http://www.epcc.ed.ac.uk/epcc-tec/documents.html

[18] Rudolf Berrendorf und Michael Gerndt:
SVM-Fortran Reference Manual
Technischer Bericht KFA-ZAM-IB-9510, Version 1.4, April 1995
Forschungszentrum Jülich
http://www.fz-juelich.de

[19] Michael Gerndt:
Sprachunterstützung zur Programmierung von Multiprozessorsystemen mit Shared Virtual Memory
Technischer Bericht KFA-ZAM-IB-9712, Juli 1997
Forschungszentrum Jülich
http://www.fz-juelich.de

[20] Matthias Hloucha:
Thermalization of Large Fluid Systems consisting of Lennard-Jones Mixtures
Technischer Bericht FZJ-ZAM-IB-9902, Februar 1999
Forschungszentrum Jülich
http://www.fz-juelich.de

Danksagung

Die vorliegende Arbeit wurde als Diplomarbeit in Informatik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen erstellt.

Dem Lehrstuhlinhaber, Herrn Prof. Dr. Friedel Hoßfeld, danke ich für die Möglichkeit, diese Arbeit am Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich anfertigen zu können.

Herrn Prof. Dr. Klaus Indermark, dem Inhaber des Lehrstuhls für Informatik II an der RWTH Aachen, danke ich für die Übernahme des Zweitgutachtens.

Besonderer Dank gilt Dr. Rudolf Berrendorf für die ausgezeichnete Betreuung, für seine stete Ansprechbarkeit und für die vielen konstruktiven Diskussionen.

Herrn Dr. Reiner Vogelsang (SGI) danke ich für die freundliche Unterstützung bei Performancemessungen auf SGI und SGI/Cray Systemen, sowie für die Beantwortung meiner Fragen bezüglich der Hardware dieser Systeme.

Für die freundliche Unterstützung bei Performancemessungen auf der IBM R50 bedanke ich mich bei Herrn Klaus Wolkersdorfer.

Den Mitarbeitern des ZAM, insbesondere Hans Georg Esser, Astrid Göke, Sven Knoben, Tamara Knödler, Christoph Kuhnhenne, Fabrice Roth und Felix Wolf danke ich für die freundliche und gute Arbeitsatmosphäre sowie für ihre Hilfsbereitschaft, die meine Arbeit positiv beeinflußt haben.

Besonders danke ich meiner Freundin Marion Scholz, die mich mit viel Verständnis und Geduld durch die Zeit meiner Diplomarbeit begleitet hat und meinen Eltern sowie meiner Schwester, die mich während meiner gesamten Studienzeit tatkräftig unterstützt und mir diese Ausbildung ermöglicht haben.